Programming

Peter Fox

A manual for everyone from novice to expert

This is a tutorial, starting from the very beginning, on how to be a programmer.

- Acquire the skills of writing computer programs
- Find out what makes programmers different
- Learn how to avoid the mistakes ordinary programmers make
- Develop your of creativity and concentration
- See the world afresh in the crystal sharp focus of precision analysis

Whether you're thinking of trying some simple programming, already do a bit, get paid to program or work with programmers this book will open your eyes and put you on the right track.

This is not a tome on Computer Science, and doesn't set out to teach a particular language. The ultimate objective is to show how to become an elite programmer, someone who can 'see through walls', asks the right questions of the right people, produces sleek and robust programs...
... and be well paid, well respected and enjoying a superior lifestyle.

About the author

Early days

Peter Fox has been a freelance jobbing programmer and business analyst since 1980. He started working on programming word processors from machine code to management awareness courses in the early days of office automation, including a fair stint as editor of the BCS's Word Processing and Office Automation specialist group newsletter.

IT revolution

During the 80's and 90's he cautiously exploited the developments in technology as PCs went from isolated single purpose tools - to locally networked departmental groups with a specific purpose and shared data - to fully networked, multi purpose systems. Applications included: A lot of critical programming for various financial services companies, stores, scientific inventory management, many odd databases, ad-hoc technical support and a multitude of small programming and consultancy jobs. In general his preferred approach is to establish a long term advisory and support role so that clients can 'pick up the phone' about urgent technical issues or general strategic matters.

Business analysis

His business insight has brought about significant changes to the quality and efficiency of clients. One instance of this is a thorough investigation of the way in which 'High street' optometry is managed within the National Health Service. Vague dissatisfaction with the procedures and confusion about how to manage professional standards was replaced by professionally designed procedures, an 'its so obvious when you put it like that' quality system, methodically designed formal protocols, and a strategic plan for implementation. The Bad-Good-Best model of competency (for clinical governance in this case) was a result of this work.

The web

Publishing on the web started out as an interesting hobby in 1995, developing as the technology, capacity and ubiquity of the web developed. Currently concentrating on PHP and mySQL, the object is to combine robust system design with efficient code production to exploit the many new ways organisations can operate in the new Internet age.

Trivia

BSc. Degree in engineering. Was chairman of and coach at Tiptree Rollerskating Club for 14 years. Qualified cycling instructor. Fool of Maldon Greenjackets Morris. Songwriter.

CHAPTER HEADINGS

@@@ To be expanded. Needs to be presented as a dynamic progression not a bare catalogue.

Introduction

Objective

I was asked which was the better of two computer programming languages for a beginner to learn. Neither was brilliant, so I did what hackers do - rolled my own. Then I realised it was the wrong question, and I should have read between the lines:

> How does a non-programmer become a programmer?

It soon became clear that the programming languages, development environments, computer science references and years of experience were secondary to:

> By thinking like a programmer.

That is the objective of this book. You can use this book to go from complete novice to elite programmer, to learn how languages work, to discover how to design a program, how to convert ideas into high quality software and what techniques of logic and automation will give efficient and reliable results.

- To begin with, for outright beginners, we'll work with a paper and pencil instruction language to get the feel for 'giving instructions'.
- Then there's a look at the basics of computer languages - a bit of hands-on Javascript and HTML.
- Followed by an introduction to Object Oriented design and programming with exercises that you can follow in whatever language you've chosen.
- Gradually the subjects become more technical and the exercises more detailed giving you the opportunity to learn a little at a time both the knowledge and skill aspects.
- As we go a theme is developed emphasising the importance of applying intellectual effort. The brain is a magnificent tool and I show you how to apply it to developing high quality software.
- Finally we look at the necessary non-technical aspects of being a programmer such as inter-personal skills and self-discipline.

This book is designed for anyone who is not afraid of applied brainwork.

- the absolute beginner
- the person who is doing 'a bit of programming'
- non-computer people who wonder how to make the best use of the best programmers
- career programmers who are wondering what is it that makes the best stand out and shine.

Becoming a programmer

Not only is programming creative, intellectually challenging and fun but people pay you well too!

- A *technician* is somebody who can understand instructions of a technical nature and if things go wrong can blame somebody else.
- *Management* usually takes the responsibility of actually making workable systems work and even defining what "work" means. To make a workable system fail you have to be management. To insist a hopeless system is delivering results you have to be a *suit*.
- An *engineer* is someone who has the responsibility for designing and building a workable system.

As a Real Programmer[1] you will be an engineer. You'll have far more knowledge about the technology than the technicians and a much better grasp of the Right Way to do things than the suits. If, by half-way through this book you've cottoned-on to the concept of 'a real programmer as an engineer' then by the end you'll have the knowledge and outlook (if not yet the experience[2]) to be able to make an impact. This 'impact' might be in personal achievement of being able to write a program to solve Sudoku puzzles, or to collect the data and draw the graph that gets the sales manager sacked[3] for being a useless waste of space - or gives the sales manager the tools so they can achieve their aspirations.[4]

If all you want to do is learn how to program in language 'X' (where 'X' is your particular choice of language) then this book will help you a bit, but not enough to be an expert in 'X'. Go and get "How to program in 'X' in 27½ minutes".

If you have already done a bit of computer programming then *this book is for you* even though you think you know the subject. Hey! Guess what books steam engine buffs read - Yes, that's right: Books on steam engines. This is a specialist subject and there are loads of wrinkles and fundamental concepts that you can miss even if you do the job every day. One of those might change your whole outlook, prevent grief and open the odd oyster or two. You should still browse the early chapters and you might find the exercises are useful refreshment.

But most of all, without prejudice to the above, this book is for people who want to program who have never done this sort of thing before. There's a nice slow start and even though later on the concepts might be complex, you'll be surfing the crest of the wave and it will be easy.

---

1 This term Real Programmer will crop up a lot. The insights and exercises in this book should give you the necessary leg-up to become an elite programmer capable of useful performance far beyond the average code-wallah. (Some people say 5 to 10 *times* better!)

2 Or the knowledge that you have the knowledge - There's a bit of Zen in all this.

3 Dumping rubbish is unpleasant but necessary. First somebody has to identify clearly, backed with figures, what is rubbish. Real programmers tend to be detached from politics and are often ideally placed to supply wood, nails, and drawings for a cross. More at the end.

4 Don't expect adequate recompense for saving bacon and many thousands of pounds. That's why good programmers should get paid large amounts of money for all that they do even if only 2% of their (often extra curricular) work brings in the Big Wins. All Real Programmers can tell you of, literally, five minutes where they cracked the case - often against opposition from slugs in suits! No wonder they're a feisty bunch.

Instructions

How to read this book[5]
- Twice

First time all the way through...
... start to finish
Goodness in every chapter.
Follow up details or key words by Internet searching

Second time as a review picking out the emphasised words and asking yourself how confident are you with them.

Note Don't get part way and give up because this book often gives partial and suspect advice to begin with then finishes it off properly in a later chapter when you're more comfortable with the foundation skills.

There are no icons to say 'Important', 'Tip', or 'Trap'. By the end you'll understand why, but as this is the beginning, the beginning of a career perhaps or a new direction that will show you things in a different light; I'll just say it is up to you to read the words which are quite sufficient (if not excessive) in themselves. This book will take you however long it takes. I can't say how long it will be before you fall-in to real programming, but when you do you'll begin to realise why programmers are a bit apart from ordinary mortals. They have X-ray vision, insatiable curiosity, the arrogance that comes from knowledge, the insight that comes from contemplation and the confidence that comes from having checked a few facts before the meeting.

> It takes an afternoon to do a parachute jump, a week to go solo in a plane, months to get a pilot's licence and years to become an airline pilot. To become a programmer is equivalent to becoming a pilot. It will be years of working at the more advanced ideas in this book before you're competent to take charge of important projects.
>
> The vital thing is to set your sights NOW on being really good at whatever level you aspire to.

Have you ever tried to start a recalcitrant outboard or motor mower? You fiddle and sweat and try again until eventually (you hope) it bursts into life. That's how I feel about YOU. I'm trying to combine the magic spark of intelligence combined with the fuel of curiosity and the oxygen of back-to-front-front-to-back programmer's logic to make YOU light up and start motoring. Somewhere in your progress through this book, probably about chapter 5 or 6, you'll (as we say in Essex) 'fall in' and the slog turns into a sleigh ride.

---

5 Yes, I know. If you can't read then what's the point of giving you instructions in writing! This is an example of the pernickety logic of programmers. Details details details! Possibilities, consequences, consternation. Opportunity, solution, result. Yes! Programmers can provide solutions to opportunities as well as problems.

0. Take aim[6]

In this book I describe how you need to think to be a programmer. It's about sharpening your perception how the world works, wondering how it might be rearranged more conveniently, and finding exactly the right way to implement a little bit of progress.

Suppose you wanted to learn to be an artist, then *part* of the syllabus would be materials, tools, and methods of using them successfully. The *important part* would be how to think like an artist and acquire the basic language of art in order to capture and develop ideas.

Programming, like painting, is a bit of an art and a bit of technology. There are styles and techniques which may or may not go together. Who remembers Monet's rough-hewn granite sculptures or Titian's photographic montages?[7] There are plenty of books in the library that will take you through the steps of learning a particular programming language. If you've been told to learn a particular programming language that's fine - Except "Learn Foo[8] in a day" is as much use as a typing course for an author: Great for typing, but rubbish for writing a novel.

So to summarise: This book will teach you how to be a programmer not how to program using a specific language.

Frequently asked questions

Q: How long will it take?

A: Days to realise this is something that will be worth spending weeks getting started and a lifetime perfecting.

Q: But my friend learnt programming in a few days!

A: No they didn't. What they did was learn part of a programming language and some bad habits.

Q: I've done quite a bit of programming and think I'm fairly good. Why should I bother with this book?

A: You may not have the breadth or precision of intellectual approach or practised the techniques that are developed as the book progresses. It is easy to be adequate in your eyes when missing a lot of tricks, becoming complacent and failing to build on opportunities.

Q: What programming language should I use?

---

6 Programmers start counting from zero. This will be explained later.

7 Ah yes, well : Just like artists who sometimes like to play around with reality, programmers are prone to "what-if" and "wouldn't that be unreal". This is flexibility of thought developed from an acute sense of awareness that information isn't really real.

8 'Foo' is a Metasyntactic variable - Luckily there is a handy glossary at the end - There will be lots of Foo later on so worth having a look.

A: The early chapters this book use *Javascript* which is built-in to your web browser. See appendix A for how to get started with Javascript.

When we come to databases there will be some *SQL*. If you don't already have a database that implements SQL then try *MySQL* which is freely available (and very good).

Later chapters are not language specific although one chapter does use *PHP* a lot for concrete examples. Whatever languages you use at least one should support *objects*.

> If you are starting completely from scratch then the combination of Javascript, MySQL, *Java* and PHP will be ideal. All are freely available.

Because 'visual programming' is difficult to describe I'm going to have to leave you to your own devices. (In my view Windows-only options should be avoided if possible.)

Q: What else do I need?

A: Pencil, paper and plenty of quiet time.

A computer that you have permission to develop programs on and create your private directories. Text editor. Modern web browser.

Access to the Internet to follow up loose ends - in particular computer science matters and investigating development tools.

Q: Are there any particular success factors?

A: Yes - Three

1 Being organised and efficient
2 Making the effort and taking time to develop your skills
3 Believing me when I say "It's not difficult - Just a little strange at first."

1. Begin!

Take a piece of paper and write on it with a pencil.

Beginner

What have you got? Does it do anything, say anything, download or control anything? Answer - No.

Now write again :

I am a beginner.

You've now got something more. A ***statement*** in the form of a sentence with correct ***syntax.*** Whilst it doesn't do a lot it does more than the first exercise. It makes sense to a reader of English. That is, a reader of the English language can understand it.

Now write again :

All beginners start here.

Is this an instruction or simply a statement of fact? (Statements of fact are sometimes called ***assertions***.) English and the way we use language has plenty of scope for ambiguity and different interpretation of the same thing. "Can you write backwards?" can be answered "yes", "no" or "backwards" (or "what do you mean exactly"). This is why computer programming languages are more formal, so that instructions will always be interpreted the same way.

Here is a simple instruction that I want you to follow:

> Draw a three-dimensional view of an ordinary die (some people call one die a dice) as it might look like sitting on a table etc. (draw a Y for the nearest corner and fill in the other edges.) (You should have one diamond sitting on two others.) Now add numbers or spots to the three faces of the cube you can see.

Notice there are three sections in brackets - let's look at these in turn:

1. 'die' and 'dice' : There are sometimes ***different flavours*** of language usage. The same applies to programming languages which develop over time. I've always called a single cube with spots on "a dice" though pedants call it a "die".
2. I gave you some instructions in the second set of brackets for how to do the drawing. Often programmers need to say "use this method". There is a technical term which you need to know called an ***algorithm***. This means 'a method of doing things' and is one of the fundamentals of programming.
3. The third bracket is an ***assertion*** (see above - a statement of fact). Programmers are always checking that things are going as planned. Not only will you be making loads and loads of mistakes and foolish assumptions but the ***Real World*** has a habit of doing its own thing. Sometimes programmers call this a ***sanity check***.

Stop!

You have just learnt more about programming than many people will ever learn in a lifetime. We will go on with the dice (or die) in a moment, but let us review the fundamentals:

- Language : What's written and what's read.
- Language : There are rules for writing and reading.
- Language : Various varieties and sub-variations of rules
- Meaning : Depends on skill of writer and intelligence of reader
- Unintended meaning : A trap.
- Instructions : eg "Turn right at the Church".
- Instructions : Rely on clear meaning eg "Go North at All saints Church"
- Assertions : Statements : eg "You are now at the pub."
- Algorithm : Complete method.

You are now about half way to having a complete mental toolkit for programming.

You may be asking "How does a computer 'know' how to do things". Good question. At the most basic level a computer is a machine engineered in such a way that it can carry out very simple instructions. That's the transistors and all that sort of stuff where the binary 0's and 1's are. Clever designers assembled more and more transistors to get to the stage where there were hundreds of instructions. Just like there are people who enjoy the art of putting together bits of coloured tiles to make a mosaic, so there are programmers who work out the best way to say find the square root of a number using only plus, minus, multiply by two and divide by two. They did this in such a way that the instructions were available in a bundle to be used as required by other programmers. Phew! Thank goodness you don't have to work out all those instructions yourself. And the put-together-lots-of-small-bits-to-make-something-more-sophisticated process continues: The square root program fragment is used by somebody who writes a mathematical equation solving program - which is used in turn by somebody who wants to solve statistical problems which is in turn used to help forecast the weather. Does that answer your question? The answer is: Because somebody told it how to do things.

Not all computers can do the same things. For example there is a computer in your keyboard which knows things like how to wait a while, how to tell if the key H has been pressed and let go, how to switch the Caps Lock light on and how to 'talk' to the main computer. Some computers will give you a web page if you ask in the right way, others will tell you the way to get somewhere if you touch a screen. Then there are those that work out your wages and taxes. The important thing to remember is that we build, adapt and spend a lot of time and money on these machines because they do something useful. When you start programming the useful thing you'll be doing is learning, but one happy day somebody will pay you to get a computer to do something useful.

A programmers job is to understand the useful thing that needs doing well enough to pick the right instructions to make that useful thing happen - preferably using as few instructions as possible with 100% reliability every time! You probably won't have much trouble finding a suitable computer for 'office' uses but some computers are more suited for some things than others. It may be a fantastic tour-de-force to put the

company payroll on an iPod but by and large programmers always take the quickest route from A to Z.[9]

Back to that dice

When I asked you to draw a die/dice I gave you some instructions but didn't bother with what a pencil is, how to sharpen it, which end works best and I don't know enough about your environment to tell you where to find a pencil. In practice this works out alright because I can be fairly certain of the *environment* in which my instructions are being interpreted - I expect you've got better things to do than read this when you're in the middle of a five-a-side archery competition, or if you haven't, then have enough nous to ask for pencil and paper when you get to hospital. Programmers are often tripped up because they make false assumptions about the environment. For example you might be reading this on an aeroplane and not have handy access to pencil and not want to use the margin of the book to write in. We will look at ways to reduce *exceptions* like this to a minimum later.

Language level

In terms of the language of programming we looked at in the previous paragraph 'draw a dice' splits into English/human instructions. Those instructions are individually broken down internally into things like 'find pencil, is it sharp, if not then sharpen it' and further into the method used for sharpening a pencil. You learned about how to hold the sharpener and twist the pencil at school. : I don't have to tell you now. Even if I was minded to do so I could get into trouble because some people are left-handed and others have propelling pencils.

There are *low-level* languages which 'speak computer' and *high-level* languages that 'speak human'. As there are different sorts of computer so there are different low-level languages. As humans like to do different things so different sorts of high-level language have evolved. In the next chapter we'll see how most of us don't need to worry about what goes on at a low-level.

The advantage of working at the highest level possible is that these nitty-gritty bits can be dealt with at some hidden lower levels without any need to worry about how they actually happen. Well sort-of: Want a square root? ... Ask a calculator or spreadsheet. How easy is that? Now can you tell me the square root of -64? Ummm. Programmers always work within *limits*. Understanding where these are in any particular situation is important. How useful is it to take a week to calculate tomorrow's weather forecast?

Data

Just to get this out of the way. *Data* is what is worked on. For example the items in a shopping list or the PIN for a cash card. Music files, picture files and word processed documents are all data. *File?* A load of data which we can identify by name. Uncle Charlies' will, Auntie Maude's horse racing tips, Events in September and so on. Programmers need to pay a great deal of attention to files, so we'll have more than

---

9 Sometimes they take the cleverest from x41 to x5A - or take frugality so far it comes out the other side. This cryptic comment will become clearer on page 999 when we discuss the art of hacking.

zero[10] chapters on the subject.

Five-a-side?

When I referred to "five-a-side archery" you didn't really think I meant exactly that did you? That was a *placeholder* for 'some other activity where pencils are not easily available'. As a programmer you'll be using placeholders all the time. Later on we'll be looking at a shopping list and sending you out shopping with it. When discussing the list it will be much clearer if I can use typical items which you replace with the actual items when you come to do it yourself. For example `www.mysecuresite.com` looks like a placeholder for 'any web site'...

...But on the other hand when I want to describe instructions I will find it clearer to give names to items that are being manipulated in the way that algebra works. In computer programming names are called *variables*. "To get the difference: Subtract small_number from large_number".

We will now finish this opening chapter with a traditional first program. Somehow the first program you write always does one thing : It displays the two word phrase "Hello World" We can do this in Beginner - It isn't very exciting to look at but to get this far you've done very well, learned a lot and might have twigged that programming is more thinking about things in the right way rather than slaving at a screen - A cat may walk on a keyboard but it takes an intelligent ape to get 4% of critical projects to work as planned.[11]

0. With paper and pencil:
1. Write "Hello World"
2. Go to pub for congratulatory beer

---

[10] Programmers often explore different ways of being precise. They may do this just to keep the brain cells active, but sometimes they use it to underline a point or place a 'must come back to this later in case there are details that need looking at' marker.

[11] If I recall the results of one study correctly 45% of real time projects were abandoned, 50% need significant alterations before being considered acceptable and 4% worked first time as advertised.

2. The basic technology of programming

Some of the statements here are only mostly true for the sake of a quick journey through what can easily become Geek-territory. Even if you've never done any programming before you should have no problem with 'giving the right instructions in the right order' but some of the technical terminology might be daunting. Don't worry, there is no test at the end and those matters look a lot less shifty when you're actually getting stared with a 'real' computer language.

Shocking news!
You had to find out sooner or later : A list of computer instructions is called a *program*. Putting the program to work is called *running*, or *executing* it.

How does a computer know what to do with a program? We did this one in the last chapter : Somebody told it (using another program) how. Let's try an example.

0. With pocket calculator:
1. Type in 365.
2. Follow instruction in box called DAYSTOWEEKS.
3. Result: Number of weeks in a year.

DAYSTOWEEKS
0. Press the divide button.
1. Press 7 then equals.

Here is the sort of format a typical programming language might use to express the same thing

```
include(MathFunctions);
days := 365;
weeks := Math.DaysToWeeks(days);
print("There are " . weeks . " in " . days);
```

For now all we need to take from this example is

a The computer reads instructions from top to bottom
b It may know how to do some things if asked nicely. For example `days := 365;` is an instruction saying "until further notice when you see 'days' I want you to use the actual number 365.
c It is universal programming style to say where you're going to put the result, then put `=` or `:=` depending on the language then instructions for how to get the result. So for example `a = b` means put the whatever the value of b is into the storage space known as a.
d We may need to tell the computer to look up special instructions or get hold of a particular resource. In this case how to convert days to weeks. You've already guessed that 'include' is an instruction with what to fetch in brackets.

Don't worry about the exact *syntax* used above - I made it up anyway.

Variations on a theme

When our made-up computer came across the line "`days := 365;`" it had to do something like "Oh a new line...better see if I can make sense of it...Aha! 'something := something' is a pattern I understand...the first something looks like a variable name...I better find a space to put whatever it is going to represent...and make sure I can find it again if necessary by putting it in a list...and here's the second something...It's a number...that's all...so I'll convert 3-6-5 into my numbering system...and put it in the place I've just reserved for it...a semi-colon...that must be the end of this line and I could understand everything." That's a lot of work just to make a note of a number. Computers might be fast but doing this every time we want to run the program looks inefficient. Hey! We're programmers and inefficiency will be hunted down and eliminated. Sure enough since the early days of computers lynch-mobs of programmers have been working at how to address this issue. The result is a program called a *compiler*. (You need to know this term.) A compiler takes program instructions that you can read and turns them into the most efficient set of instructions in the computer's own language. If you recall when we discussed high-level and low-level languages I said that most of us don't need to worry about the low-level. The reason is that a compiler will translate our high-level stuff into something the computer can understand directly.

> For those of you who want to see what a computer program looks like to a computer here is one I wrote to speed up the repeat key:
> B4 03 B0 05 B3 00 CD 16 B4 4C CD 21
> This is just a list of 12 hexadecimal¤ numbers. "B4 03" (or 180 3 in decimal) says "First number is 3" as instruction number 180 puts the next number it finds into the processor's ready-use location. It's a bit like pressing "3 M+" on a calculator. Numbers like this are known as *Machine code* - If you're interested in tweaking electronics then look up *Assembler* as well.

There is another benefit from having a compiler looking at your instructions before you try to run the program for real: It can pick up *some* of your silly typing mistakes ahead of time. Let's suppose your program works for an hour then finally comes across an instruction that it can't understand. There will be a gnashing of teeth and possibly wailing when all that time has been wasted and you have to start over again. Or it whirrs away for an hour and you begin to suspect it has entered an *infinite loop*. (We'll deal with this common error later.)

To cut a long story short, as technology improved another way of running programs was found which was exactly the original plan where each line of instruction is interpreted at a time. Yes it was slower to process, but it was much faster from the human point of view. A typical 1970s program would be punched onto a set of cards, handed over to the computer room, run when there was a convenient slot in the schedule - often overnight, returned with a listing of errors or a wodge of printout where something went wrong, then the mistake traced by hand and the corrected cards resubmitted. Perhaps one of the origins of the 'all hours of the day are equal to us' culture is that programmers could get computer time early in the morning or at weekends. (And nobody expected you to wear a suit and tie at these strange times either.)

The *interpreted* language that revolutionised the 1980s was BASIC. Not only was it small enough to fit onto the micro computers of the day but it was also easy enough for engineers to pick up and see the potential of for doing real work. BASIC came in many

flavours but once learned it was easy to move from a machine with 32K[12] Bytes of memory to the company mini-computer with a hard disc, printer and possibly 512K of memory. Here is an example BASIC program to give you a flavour.

```
10 DIM A$(7)
20 DATA "Mon","Tues","Wednes","Thurs","Fri","Satur","Sun"
30 FOR I = 1 TO 7
40 READ A$(I)
50 NEXT I
60 INPUT "What number day of the week (0 to end)", D
70 IF D = 0 GOTO 100
80 PRINT "Day number "; D; " is called "; A$(D);"day"
90 GOTO 60
100 END
```

BASIC did what it was originally intended, opened the door to programming a computer for millions of people. It also developed, but every development has involved a lot of looking back and it will never be the computer language of the 21st century for reasons you will discover later on.

If you want to know how to interpret this program here's a brief explanation: (This is the throwing-in-the-deep-end-bit. Don't worry the lifeguards are waiting in the next chapter.) The lines are numbered for convenience of being able to jump to a given line. That's what happens in line 90. Modern languages don't use line numbers[13]. Line 10 sets up a *data structure* called an *array*. Both Data structure and Array are very important. Here the array is a list of seven spaces for storing *strings*. Strings?! In computer programming parlance strings are bits of text. Line 20 gives the program seven strings which are put into the array called `A$` one at a time starting from slot number one through to slot number seven.[14] Can you see how `I` gets shuttled round the 30-40-50 *loop*. Line 60 prints a message asking for a number on the screen[15] and waits for the user to type in something and press the Enter (AKA carriage return) key. Line 70 jumps to the end if the user typed in 0. Line 80 prints two *variables* sandwiched between three *constant* strings. The first variable `D` is the number we just typed in, the second is the Dth slot in the `A$` array. If `D` is 6 then `A$(6)` gives "Satur". Sticking strings together is called *concatenation*. The *program logic flow* now *jumps* back to line 60 to see if we want to have another go.

We never get to the end from line 90...will the program ever stop?...oh yes, possibly, from line 70. One of the reasons we don't use GOTO in modern languages is precisely because we can end up in a mess trying to trace the program logic. (The term for a program where the logic flow skips all over the place is called *Spaghetti code*. this is one of the reasons BASIC got a bad name - because it made it so easy to write spaghetti.)

Finally, and vitally, *what could possibly go wrong?*, and how can these problems be

---

12 K is short for 1024 or 'about 1000'. More details in the glossary.

13 But line numbers are great for explaining what's going on where which is why we'll be using them in Beginner.

14 Hey! You say. Didn't you just tell us in chapter 0 programmers start counting at 0? Yes I did, but sometimes we have to conform to the prejudices of the real world.

15 Or as it used to be, literally bashed out on a Teletype terminal with a noisy rattle.

dealt with? For example what happens if somebody types in 66? We may have spotted the potential problem and ***validated*** the input (for example by making sure it is between 1 and 7) or there may be others lurking that we haven't thought about that we need to ***catch*** to at least give a soft landing. The original BASIC had no way of dealing with unexpected ***runtime errors*** apart from just stopping with a cryptic message - This became a serious problem when BASIC was used for real applications.

Review

If you've got this far then well done! I expect you're realising there's a lot of meat on this chicken. Don't worry - There *is* a lot of meat - but you don't have to take it all at one sitting! And anyway if it was really easy then everyone would be doing it and we don't want *that* do we.

What you need to know is:

- Computer languages are designed to be human readable
- A program is a list of instructions which the computer will translate into its own instructions either
  - once-off in a compiler, or
  - as-you-go in an interpreter.
- A written bit of a program is called ***code***. For example I might say "The code for loading the array starts at line 30"
- Different languages have different ***key words***...
- ...and different capabilities, ways of representing variables, organising code and dealing with errors. Some don't use the list of instructions style at all - We'll look at one of these in a minute.
- There are all sorts of ways errors can happen - some of which you'll probably never discover.

Where's the logic gone?

0. With PC connected to the Internet
1. With a web browser
2. Visit a web page
3. View the source

3.1 Try

3.2 A right-mouse-click should give a menu containing "view source"

3.3 Except

3.4 If 3.2 doesn't work then look for View on the top menu. If that fails try following the Help in the top menu.

4. Repeat from 2 (with new web pages) until you've got 3 or 4 pages of source code.
5. Assertion : You've got a few pages of gibberish on the screen.

Let's look at this Beginner code from a programming language point of view.

- 0. and 1. say what resources are needed. A bit like `10 DIM A$(10)` eh? we'll see a lot more of this later.
- 3. has been expanded to give the important instruction in 3.2...
- ...But not all browsers work the same so we have ***trapped*** something going wrong.

- *Source* is the term used for the program as originally written.[16]
- 4. is a *conditional loop*.
- 5. is a sanity check. We need this because the next section depends on this one having achieved its purpose.

Yuk!

What you're looking at is computer instructions. Scroll to the bottom and you should see something like `</body></html>`. You can decode these as follows:

`<...>` = Instruction
`/` = End ...
`body` = Subsection of *HTML* document that contains the bulk of the page
`html` = Block of HTML program code

HTML is the language used to write web pages with. Note: In HTML the instructions in angle brackets are called "tags". Since everyone calls them that we'd better conform.

So if there is a `</body>` (= end of body section) there should be a start of body tag (looking like `<body>`). Search for this. Very close to this you should see the end of the head section (`</head>`). By looking at all your sources you *should* see all the pages conform to this pattern.

There might be all sorts of junk at the top of these pages so go back down to the bottom and have a look around for other tags. You should see a pattern of `<foo>` ... `</foo>`. (Remember "foo" is a metasyntactic variable which 'stands for something' possibly `i` or `div` or `table` or `tr` or `h2` and so on.)

That's enough grovelling through gibberish - how about writing your own simple web page which will be a test bed and technology demonstrator. Trying out something for yourself is a typical programming gambit. What works for somebody else may not be suitable, or simply may not work, for you. Also having a go yourself 'burn's in' some basic knowledge that makes more complex details easier to interpret.

Starting to code

In ye olden days before everyone had access to their own terminal we used to write programs (in pencil so we could erase mistakes) on "coding forms" which had eighty columns marked across the page each allowing a single character to be written. These would then be punched (often by dedicated staff) to give a bundle of punched cards, one card per line. Guess how popular you were if you dropped a bunch of cards on the floor[17]. Then the cards were stacked in a hopper with top and tail cards to tell where the start and end of the job was and who was going to get the bill and next day (literally if you were lucky) you'd have your printout wrapped around your stack of cards. Sometimes there were only certain times when certain programs could be run, so if you missed your window to get things right you might have to wait days. Of course a program running perfectly first time was unusual so you'd change a few cards and go through the whole thing again. All this slog was rewarded by great satisfaction when

---

16 For the record 'Object code' is what a compiler produces from the source code.

17 If you dropped punched paper tape that just got into knots. With PPT you had to rewind it each time after running through a reader, and since you couldn't just swap a card to fix a typing mistake you might cover some holes up and punch more with sticky tape and a hand punch. I've done that.

IT WORKED AT LAST!

So now you can see why programmers despised sloppyness, hand waving and suits and developed maverick working practices characterised by bursts of intense concentration.

Your turn!

In many ways your coding routine is a lot easier, but I'd still recommend trying to cut-off from the distractions of the world while squeezing brain juice into your fingers and onto the screen.

0. With a PC, web browser
1. With a text editor
1.1 A text editor is a basic sort of word processor. As programming languages don't like odd things that word processors add into text to prettyfy you can't use a Word processing program. Instead...
1.2 All PCs have a text editor somewhere, or you can download one to suit. (On Windows look for a program called Notepad.)
2. Finding a text editor was probably the most difficult bit
3. Type in the following.
3.1 You don't need to get the layout exactly right but everything else must be spelled correctly.

```
<html>
<head>
  <title>A first web page</title>
</head>
<body>
  <h1>Hello World</h1>
  This is normal
  <b>This is bold<i>and italic<font color=red>and red</font></i></b>
</body>
</html>
```

4. Save this to your temporary directory as `first.htm`
4.1 If you don't have a directory called `temp` make one
4.2 Everybody needs a `temp` directory to put their transient files in
4.3 You now have a file called `first.htm` in your temporary directory
4.3.1 Go and have a look in the temp directory.
4.3.2 If all you see is "first" and you are on a Windows system then you need to set up your 'explorer' to show file extensions. Set to ON, leave to ON and wonder at the stupidity of people who put up with a computer system that censors vital information. *File extensions* are a vital part of the technique of programming.
5. 'Point' your web browser at `temp/first.htm`.
5.1 On a Windows system double clicking on the file should be sufficient
5.2 Other operating systems may need you to enter something like : file://temp/first.htm into the address bar.
6. You should now see your first web page.
6.1 Compare what you see on the screen with the code. (You can view-

source as you did above to prove there has been no trickery to change your code.)

Round the loop

Hooray! Well done. If that was a bit of a struggle then don't worry, in a moment we will do the same thing all over again and this time it will be a doddle because you've already got the bits you need lined-up and know how to make them work.

Let's look a bit more closely at the code in `first.htm`.

There are no line numbers but perhaps your text editor has an option for showing them. Worth having a look for the option now or getting a better text editor as you'll need them in the future.

We saved a text file then ran it. In a moment we'll improve it and run it again. This is the basic ***coding process***. *There is a lot more to programming than simply coding, for example finding out what needs to be done, finding a way to do it, and testing to prove it really does what you wanted.*

Blocks

Practically all programming languages have ways to organise code in sections called ***blocks***.[18]

> Two common ways to indicate the start and end of blocks are
> `begin ... end`
> and `{ ... }`

HTML illustrates blocks nicely. Remember that `<foo>` marks the start of a block of foo and `</foo>` marks the end. So the `html` block extends from first to last lines and contains two child blocks: `head` and `body`. The `h1` block is a heading-size-1.[19] When the interpreter inside your browser reaches the `<h1>` instruction it says to itself : Make everything bigger from now on. When it reaches `</h1>` it says to itself : Cancel that `<h1>` instruction from now on.

Blocks are like shoes and socks

Try this and see what happens.

0. With pair of socks and shoes
0.1 Start with bare feet
1. Put on socks
2. Put on shoes
3. Walk around
4. Remove socks
4.1 Slight problem!
5. Remove shoes

Now look at the bold-italic-red line of code. Notice how the blocks ***nest*** they don't

---

[18] BASIC, which we saw above, is one that doesn't.

[19] 1 as in biggest, 6 smallest

overlap.

Another way of typing that line of code could be:

```
<b>This is bold
  <i>and italic
    <font color=red>
      and red
    </font>
  </i>
</b>
```

You can see that it is sometimes really handy to ***indent*** code in this way. Indenting is pretty much a universal layout aid - that's why coding is done using mono-spaced (ie m is the same width as l and all other characters including a space).

A useful quirk of the HTML language is that it treats all ***whitespace*** (tabs, spaces, new lines, blank lines) as a single space....
...which is why the browser stuck "this is normal" on the same line as "this is bold...".
Did you expect that? Let's do something about it anyway.

0. Back to the editor
0.1 Edit the first.htm file
1. Add " " after "normal"
1.1 Don't type any quotation marks
1.2 means 'put in a paragraph break'
1.3 In HTML doesn't need a (Later we'll see a 'paragraph' instruction that is a block.)
2. Change the shade of red
2.1 Replace "red" with "#ff6060"
2.2 See the discussion below for what this is about
3. Save the file
4. Look at the file again with the browser
4.1 Possibly just refreshing the page
5. You should see a new line after normal and a muddy red colour

The lure of Geek

What is that #ff6060 all about then? You can see the computer thinks it is muddy red but so what? You're looking at the buzz of programming in a microcosm. By knowing spells that ordinary mortals don't you can achieve things they can't. They respect and fear your powers. Amongst programmers feats of ingenuity and technical prowess are appreciated for their wizardry and cleverness - and called ***hacking***[20].

To an experienced hacker #ff6060 shouts "hexadecimal"[21] In the case of HTML the #

---

[20] Hacking is frequently seen as a Bad Thing, but started as the coolest technical activity and resulted in huge steps forward as people showed what was possible - and upset the people who wanted to sell overpriced junk to mugs. More about this important aspect of programming in a later chapter.

[21] Base 16 numbering with digits 0 to 9 then A to F so F is '15' and FF is 255 - See glossary for more

tells the interpreter to expect some hexadecimal next and the next 6 characters are three lots of two being how-much-red, how-much-green and how-much-blue to mix together to give the required colour. ff0000 is all red, 0000ff is all blue (you can try these yourself and others) ffffff is white and 000000 is black.

Now if I tell you that the whole page can have a colour, you do this by putting `bgcolor=foo` *inside* the opening `<body>` tag, for example `<body bgcolor=#bbffbb>`, then you can have hours of innocent fun editing-saving-refreshing. We'll do some more HTML in a later chapter, but if you can't wait until then have a look at some more sources and surf to a HTML quick reference.

Review

You are a programmer! With your code editing tool you can create a program that a web browser can interpret. You've also scratched the surface of computer geekism with hexadecimal.

You were probably dismayed by the gobbledegook when viewing the source code. Don't worry it dismays most programmers! Looking at somebody else's code, or yours from a while ago, is daunting. (Obviously if you knew a bit more about HTML and Javascript and DOM and weird Microsoft additions then things would be a bit clearer.)

We've seen how HTML is structured by nested blocks which is a basic feature of practically all languages. However most have additional ways of dividing the code into *smaller and more manageable* units.

3. Logic

With HTML the flow started at the top and went on to the end - always. There is no way to say "repeat this bit until..." or "if this then do that". Most computer programming languages do have these sort of features.

If

Fancy a mug of coffee? Let's try it.

```
0.  With kitchen
0.1     Kettle, water, instant coffee, spoon, milk, sugar, mug
1.  If there isn't enough water in the kettle, then put enough in
2.  Switch kettle on
3.  Find mug, coffee and spoon
4.  Use spoon to put some coffee into mug
5.  Wait until kettle boils
5.1     NB. Assume kettle switches itself off automatically
6.  Pour hot water into mug
6.1     If having 'white' then leave room for milk
6.1.1       If 'milky'
6.1.2       then fill to 2cm short of brim
6.1.3       else fill to 1cm short of brim.
7   Possibly add sugar
7.1     If having sugar:
7.1.2       Sweeten to taste
7.1.2.1         add sugar to taste
7.1.2.2         stir
```

Let's dissect this:

0 We identify the resources we might need

1 *IF ... THEN* crops up just about everywhere. This switches the flow of control once depending on a certain condition.

2 The kettle starts heating the water...

3,4 ... We don't have to wait for this get to the boil - we can be doing something else at the same time. (Programmers call the two independent stream of events *threads*. More about this important subject in a later chapter.)

5 Do nothing until... This is a sort of reverse IF ... THEN where something continues to be done if the test is true. Waiting relies on us doing nothing except being awake enough to realise when the conditions are right for us to continue.

5.1 This is a *comment* that explains how the program is operating to somebody who is reading the code. It is generally considered a Good Thing to put plenty of comments in your code. An interpreter or compiler needs to know what bits are

comments and for human eyes only. Each languages uses its own conventions.[22]

6.1.1 IF...

6.1.2 THEN... We dealt with this in 1.

6.1.3 ELSE... This is an alternative to 6.1.2. *IF...THEN...ELSE* also crops up just about everywhere. Of course all it means is 'do this OR that'.

7.1.2.1 and 7.1.2.2 are a block of code (you remember blocks from the previous chapter) which we might call the 'sweeten to taste block'. For illustration let's show that in two styles of programming language:

```
if ( sugar==TRUE ) {
   AddSugar;
   Stir;
}
```

```
if sugar then begin
   AddSugar;
      Stir;
end;
```

Obvious? Best? Correct?

Getting the instructions in the right order is can be much harder than it looks. Often there are good reasons why you prefer a certain sequence. For example how much effort must you put into checking all the conditions are correct and dealing with rare situations and where do you detect errors and how will you handle them? You will shortly get a chance to try this for yourself as you write the *Beginner* instructions for making coffee for a bunch of people.

I know there happens to be a very nasty *bug* lurking around line 5. If you can discover it for yourself than you're very smart and thinking like a programmer thinks. The answer is at the end of the chapter.

Does it work? Could it be improved? Often the best way to find out is to *trial* the program. Then at some stage 'lets-see-what-happens' becomes 'I-guarantee-so-and-so-happens' which you do by *tests*. Trials are experiments where you aim to improve the way a program works. Tests are exercises to convince everyone the program can be relied on to 'do what it says on the box' under all conceivable circumstances.

A huge amount of effort goes into getting the right order of instructions and proving they work. Massive losses (including loss of life) have been incurred by failing to make the effort.

The flowchart

Some people think in pictures and it is often handy to be able to sketch the possible paths a program could take using pencil and paper. So the *Flowchart* was born. It used to be de rigeur to draw these before putting pencil to coding sheet but nowadays they are not used so much, partly because modern programming languages lend themselves to better readability.

Here is the flowchart for making the coffee.

[image]

The diamonds are decisions. The boxes plain actions. There's no place for saying what

---

[22] HTML comments look like <!--- foo -->. More line-by-line languages have symbols to say 'rest of line is a comment' More in the glossary.

resources we need and it doesn't show the "kettle coming to the boil" thread. (If it did it might alert us to that bug I mentioned.) However although it has drawbacks, if you wanted to explain the logic to somebody the 30 seconds spent on a sketch would be a good investment.

Confusion be gone!

Before we go any further we better start giving our *Beginner* programs proper names. In a minute we'll have multiple coffee making programs and we'll get confused. The obvious way in Beginner is to give the program a title on the first line. Just so we know it is a title we won't give it a number and just so the title stands less chance of being mistaken for something else when we're referring to it we'll write it all in capitals and not allow spaces but use underscores instead. eg *COFFEE_FOR_ONE*

Naming conventions like this play a big part in programming. Sometimes they must be observed and sometimes they are a convenience. In general, programming languages don't like you using instruction commands (called *reserved words*) for your own names. It is fairly common for `I` and `J` be used as variable names for counting through loops. We'll deal with conventions later as we come to them.

Loop

What if you are making coffee for more than one person? The program will go something like:

1 Note who wants what
2 Boil kettle etc.
3 Make N mugs of black no-sugar
4 Add milk and sugar working down your list of requirements

I'll start you off on COFFEE_FOR_N[23] and you can have a go at completing it.

```
COFFEE_FOR_N
- Instructions for making the coffee for N people
- Coffee options limited to White/Black, Sugar/No-sugar
- Designed for values of N up to a handful
- Not tested for N more than 4
0.  With kitchen
0.1     Kettle, water, instant coffee, spoon, milk, sugar, tray
0.2     N mugs
1.  With notepad
- to write list of people and preferences
2.  Collect orders
...
```

Your task is to complete this program. You can see the sort of detail that's appropriate. It might help to think of giving instructions to a very dense child. Notice that I've used a dash to indicate a comment. Useful aren't they!

---

[23] See! Due to some peculiarity of the human mind you've already twigged that COFFEE_FOR_N is a Beginner program.

The handy outline I gave above that is sort-of instruction is called *pseudo-code*. It's often a good place to start. It allows you to develop ideas and get the general picture without becomming bogged down in detail.

Review

Don't worry if it all looks like a lot of work and you haven't done any 'real' programming yet. What we've been doing is putting in the foundations and laying the services for a palace of programming. This isn't grovelling in the mud it's levelling the site.

How did you get on with COFFEE_FOR_N? Now you see why programmers use pencils - details can be a little tricky. If you're bold you could *trial* COFFEE_FOR_N which would be doing it yourself or *test* it which would be getting somebody else to do it under your supervision. Probably worth a test if you can persuade somebody to be a guinea pig. We'll do more on testing in a later chapter.

Iteration and storage

If you were writing COFFEE_FOR_JIM,MARY,JOE_AND_ALICE you could write instructions like this:

```
3.  Get requirements
- Everybody always has the same thing
3.1     Jim : Black NS
3.2     Mary : White NS
3.3     Joe : White Sugar
3.4     Alice : White 2-spoons
```

But in COFFEE_FOR_N we don't know who'll be having coffee or even how many. So we need to have a flexible system. In Beginner this is just a notepad which we add names to and put requirements against. This might go like:

```
3   Get requirements
3.1     Find a page of notepad with enough space to list N names and what they
        want.
3.2     Divide the page into three columns called NAMES, MILK and SUGAR
3.3     For each person in room do the following
- Note : Don't forget yourself
3.3.1       Ask if they want coffee : If 'No' then skip to next person.
3.3.2       Get name and write in NAME column
3.3.3       Ask how they like it and fill in MILK and SUGAR columns
```

What we've done is reserved enough storage space to make a note of who wants what then, one line at a time started filling it up. Now you see why N=1000 might not be a success as we run out of paper. There are ways round this but for now we'll just put some arbitrary upper limit on N. In computing terminology a list or table like this is called an *array*. We came across one of these in the BASIC program in chapter 2. On paper the table for COFFEE_FOR_N looks as follows. (I've filled in some of the items with made-up values.)

Index	Name	Milk?	Sugars	Mug

1	Peter	Yes	0	Red
2	Geoffrey	Yes	0	Chipped brown
3	Sally	No	1	Skinny yellow
4	Adam	Yes	2	Green
5				
6				

What's that Index column doing there? Well it isn't actually a column in its own right just a counting device so we can say 'row 3' or the '4th Mug'.

Types

Two things plus two things added arithmetically is four things, but one string (that's a series of characters) plus another string added on the end give one string.

```
2 + 2                // gives 4
"Great B" + "ritain"   // gives "Great Britain"
```

(// is a common way of indicating that the rest of the line is a comment.)
Now what does `2 + "two"` give? An error. We're trying to mix apples and pears.

Binary and all that

As you probably know everything inside a computer is stored as numbers - *bits* in fact. A bit can be 0 or 1. To do useful things we usually work with a number of bits together at a time. Working with 8 (and multiples of 8) at a time is now the established norm. 8 bits together is called a *byte*. With all 8 possible combinations of 0 and 1 that gives 256 different values a byte can have. The clever part is how we look at these bytes. If all we ever want to do is count from 0 to 255 we can do this using one byte. Or perhaps we want to count from -127 ... + 127 which would still fit the limits. We can represent characters using a numbering scheme. Let's say 65 for A, 65 for B, 66 for C and so on.

But what about numbers like 6 million and ½? What about dates? What about 6 trillion trillion and 5.543 times 10^{-12}? (@@@Later we'll investigate silly utility bills - What is one third of £10?) Types are discussed more in the glossary. *For the time being* all you *need to know* is as follows.

Integers : Whole numbers used for counting, but only between limits. They come in various flavours. Typical limits are +/- 32 thousand (*16 bits signed*) and 2000 million. (4 bytes = *32 bits signed*)

Reals or *Floating points* : Much wider range of possible values for the same number of bytes but not absolutely precise. All non-counting maths will use floating point arithmetic. Typical limits are 10^{-45} to 10^{38} with 7 significant digits (*single precision* using 4 bytes) and 10^{-324} to 10^{308} with 15 significant digits (*double precision* using 8 bytes)

Characters and *strings*

The basic rule is one character per byte.[24] Although you can't do maths with strings you can ask "is 'cat' greater than 'dog' ?" and get a reply based on which comes first in the dictionary.

Boolean
True or false.

Other ***built-in types*** vary considerably between languages. For example some may have types specially suited to currency calculations or dates and times or just a limited set of values or blobs. (@@@See the appendix for more on blobs and lots more on built-in types.)

User defined types in various guises are really handy. For example in COFFEE_FOR_N we might have a type that combines Name,Milk,Sugars and Mug in one handy package. We will be investigating this at length in later chapters.

So if we were putting COFFEE_FOR_N onto a computer what types would we use for our table of requirements?.[25] Name looks like a string. Some languages might ask us to say the maximum length we will ever need. Milk looks like a boolean (either we have milk or not).

How does the real world work?

What about the Sugars? Will we always be asked for whole numbers of sugars? This is an extremely important issue which as a programmer you will encounter every day in one form or another. Your job is to model the way the real world works using a computer as efficiently as is reasonably possible. Just because my ***sample data set*** doesn't contain 'and a bit' doesn't mean when the program is released into the real world that somebody won't ask for 'a bit'.

Being able to see in your mind's-eye what might possibly happen when the program is used for real is an essential programming skill.[26]

So let's make Sugars a single precision floating point number.[27]

Mug looks like another string, but if we had pictures could we use them? Possibly, if the language we were using supported blobs[28]. A picture would take up a larger area of

---

24 This will do as a working definition - but there are variations.

25 Supposing the language we were using gave us the option. Some don't, some don't care, others insist.

26 The computer is also part of the real world. If you know its limits you may be able to head-off over-optimistic ideas which are doomed to failure such as "the next month's visits for each salesman can be planned to give the minimum distance to be travelled when each salesman phones-in to ask for it". See Travelling Salesman¤.

27 What Could Possibly Go Wrong? How about a request for sweetener.

28 Binary Large Objects : Loads of bytes that the programming language doesn't know how to interpret but need to be kept as a bundle. Digital photographs for example. See Appendix for more details.

the notepad, would take longer to draw originally but might have other advantages.

Arrays

An *array* is a sequence of slots put aside for storing information. The process of allocating the necessary space for the job is called *dimensioning an array*. Often arrays are of a fixed size, which as we have been discussing recently, is a limit which we need to consider given the likely maximum *ever* number of slots required. (There are alternatives which have adjustable sizes - we'll look at them in a later chapter.)

Here are some dimensioning examples in real programming languages.

BASIC	`DIM $NAME(6),MILK(6)`	$ in front of a name indicates a string. Otherwise a number[29].

Delphi	`var`
	`  name : array[0..5] of string;`
	`  milk : array[0..5] of boolean;`
		Delphi gives you the choice of lower and upper limits.

Java	`String[] name;` `Boolean[] milk;`	Java @@@
PHP	`name = array();` `milk = array();`	PHP's arrays are very laid-back new-age sorts of things. We can put as much of anything we like in them.

Normally we index array elements with an integer. Here are some examples in real programming languages.

BASIC	`$NAME(3)`	This would be 'Sally' using our sample data set.
Delphi	`name[2]`	Also 'Sally'
Java	`name[2]`	Also 'Sally'
PHP	`milk('Sally')`	Some languages are not limited to integer indices.

More points about arrays:

- Typically you can have more than one dimension. For example `DIM $DAY(12,31)` allocates 12 lots of 31 slots.
- Plenty of languages don't have arrays at all
- If there are say 6 items in an array that starts at index zero you can only go up to 5.
- A common error message is Subscript out of range. This means "you have tried to access an array element using an index less than the minimum or more than the maximum. This is probably the most common 'unexpected' run-time error.
- Arrays (and their cousins, lists) are used all the time.
- How do you know if an array element contains something valid or not? Some languages always fill a newly dimensioned array with 0 or " or false or something

---

29 In olden days possibly only integers allowed!

depending on the type of the array, but others don't.

Another coffee...

It's a while since COFFEE_FOR_N. When we left it we'd established a table of what people wanted. What was left unspoken (*but you can't leave things unspoken when you're talking to a computer*) was the number of people having coffee. When we get to the kitchen we will need to know this number for getting enough cups and putting enough water into the kettle. How can we tell?

One way is to make an explicit count. 1..2..3..4. Another way is to count the number of entries in the list. OK then, what's the number of items in my sample data set table: 4 or 6? There are 6 rows but only 4 have been filled in. As humans we can tell instantly that a blank row is to be ignored but a computer can't. Even if a computer finds a blank line and it thinks Oh dear something wrong with this line - ignore it does that signal the end of the list or perhaps just a missed out or scrubbed-out line? *You* tell it!

...another essential programming skill

Let's cover that last bit again. You have to instruct the computer to do something so-many times to such-and-such a collection of data.

In this simple case you could have made a separate note of 'how many' then 'starting from the top for N times do foo' or 'work through the list and for those that make sense do foo'.[30] The first method is explicit the second is empirical. (Empirical means "based on practical experience" (as opposed to logically correct))[31]. There may not be a 'right answer'. The second option looks more reliable: Somebody may have changed their mind and not wanted coffee after all so the list has say three names on it but the second is crossed through. But what if your empirical method is 'keep on down page until you get to the end of the list' but the list is hundreds (just in case) of (empty) lines long? You'll be there all night!

I hope you can see why good programmers are rare. They have this sort of dilemma all the time. For example a method exists that will normally sort a list very quickly, but once in a blue moon the method will take much much longer. What are the chances of that happening?[32] A programmer needs to be able to spot strong and weak ways to describe the information that's available... ...and also the relative cost[33] of using those methods.

Belt and braces

Why not use both the explicit count and the 'step though until no more' methods? That

---

30 You should be comfortable with Foo by now : Foo = something the same way that Widgets are bits in a factory. Don't try to use Foo outside a programming context - everyone will think you are mad.

31 Notice the nested brackets. Remind you of blocks of code? Thank goodness the English language hasn't been perverted by programmers, but from time to time it is really handy to use a coding paradigm.

32 Either very very small or, by Sod's law some effect conspires to arrange the data in just the way you don't want. Random data is almost unheard of.

33 Cost is an abstract concept discussed in the Algorithms appendix.

way if there's a discrepancy then an alarm bell will ring. That's exactly what good programmers do.

- Looking for discrepancies is second nature to programmers.
- *Validation* is the term used to ensure that data is sensible.
- *Assertion* is a technique that says "at such and such a point in the program the following condition should always apply".

(Actually validation and assertion are siblings rather than examples of belt-and-braces, but for now I want to emphasise that programmers are always on the lookout for ways to ensure rubbish is rejected before it can pollute the rest of the program.)

Here is one way we might implement belt-and-braces in COFFEE_FOR_N.

```
5.  Line up N mugs
. . .
8.  For each item in list pour in the hot water
9.  All mugs should now be full
9.1     If not then something has gone wrong
9.2     Find out why there is a discrepancy.
```

Coffee-making review

Did you use a tray to carry the mugs to your guests? If not how did you carry them?

- FX: Sound of extra instructions being added.
- Think bubble : Perhaps I should have done some testing.

If you used a tray did you measure the maximum number of mugs it could hold and feed back some limits to the design of the program? Or perhaps made the program more complex with a "...then make a trip back with those and come back for more" instruction.

This sort of issue is meat and drink to a programmer. There is always some wrinkle you've overlooked. You often spend more time heading-off or dealing with the things that can go wrong rather than the mainstream events.

Did you spot that serious bug in COFFEE_FOR_ONE?

```
5.  Wait until kettle boils
```

What happens if the kettle is switched on but not plugged in. How long will you wait? A computer would wait forever! This is a subject we will return to in a later chapter.

As you can see there are plenty of traps and no single right answers.

4. Control structures

We have briefly seen IF...THEN...ELSE and FOR loops. As switching the flow of control around a computer program is essential technique we'll nail these and their cousins down before going further.

Some languages[34] don't have flow of control switching.

Precise syntax varies between languages; The objective here is to let you immediately recognise what you see when looking at a program and think about choosing the most convenient for your own.

Plain jump - Go to

Some of the code I've shown you has used `GOTO 60` etc. That was easy to understand. WCPGW?[35] You lose track of how your code works. Many languages don't even have a jump instruction and with modern high-level languages you should never need it. It is great for two things:

1. quick and dirty instructional purposes
2. instructions for humans. eg. procedure manuals

But that's all. Don't use it in your programs unless you're working at a very low level or with a language without any other ways of redirecting the flow of program execution.[36]

Testing conditions

We make decisions based on tests. For example:

0. *Look outside and note weather*
1. *If raining then take umbrella*

or

```
if (tray_size < number_of_mugs){
  ... work out a way to carry more ...
}
```

or

```
while (tray_size < mugs_on_tray){      // If there's a space
  AddMugToTray();                      // put next mug onto tray
  if(not AnymoreMugsToTake()) break;   // loop unless no more to do
}
```

Don't worry about the empty brackets at the end of AddMugToTray() for the moment.

Relations

---

34 eg HTML, SQL

35 What Could Possibly Go Wrong? The phrase on the lips of every real Programmer all day long.

36 One reason for eschewing GOTO is that it resulted in spaghetti code with jumps *going* in all directions and with no clear indication where they were *arriving*. You could be at the top of a loop and not know it until you reached the bottom umpteen pages later on in the printed listing.

The basic tests commonly found in programming are:

Relation	Typical symbols
• Equals	= or ==
• Not equals	!= or <>
• Greater than	>
• Less than	<
• Greater or equal	>=
• Less or equal	<=

(Equals gets a whole section to itself in a minute.)

The symbols shown are called *relational operators*. Look up relational operators in the documentation for a programming language to get the details.

Normally these relational operators work with strings as well as numbers. Quite likely they will work with other types and sometimes you'll be able to mix types and others either be prevented by the interpreter or on the road to some strange and unpredictable results.

Not and other booleanisms

Tests can usually be 'reversed' by putting a `not` or `!` before it.

Tests can usually be combined using *boolean operators* `and` and `or`. Sometimes `&&` is used for `and` and `||` for `or`.

Compare this loop with the previous example

```
while((tray_size < mugs_on_tray) and (AnyMoreMugsToTake())){
  AddMugToTray();
}
```

That's a bit easier to understand. Tip: It may not be strictly necessary, but putting brackets round multiple combined conditions can save a lot of head scratching. This is because of *operator precedence*. (In any expression the interpreter will scan for some sorts of operator (+,-,>,! etc are operators) to work with before others. If you put in brackets you can be sure that the sub-clauses are not jumbled up.

Functions

Functions can appear anywhere and are the core of many modern languages. Let's introduce them quickly here.

A function is a block of code with a name. Here is an example:

```
function LargestOfTwo(foo,bar){
// This function returns the larger of two arguments
  if(foo >= bar) return foo else return bar;
}
```

foo and bar here are called *arguments* or *parameters* - terms we'll be using a lot trom now on. You know what `foo` is - anything, it's a place marker. `bar` is another metasyntactic¤ variable. We could have used `Fred` and `Charlie` or `a` and `b` or `n1` and `n2`...

...but then we'd have to substitute in the body of the function as well. In fact foo, bar, baz and any other local are never used like this - metasyntactic variables are always for

human consumption.

Often functions *return* a value. Sometimes that's the whole purpose of the function, and other times all you want is a "Yes-OK I did what you wanted and it seemed to go OK" like this:[37]

```
function SwitchKettleOn(){
// Operates kettle switch, returns true if neon lights up
  KettleSwitch = 1;   // 1=on 0=off
  if(KettleNeon==1)return true else return false;
}
```

Note how there's always a comment to tell us what the function does. Tip: No matter how simple the function is *always* put a comment in.

So how might we actually use the LargestOfTwo function? You can think of it as a black box, which spits out a result when we feed it with two numbers.

```
10 input "Please type three numbers";a,b,c
20 largestab  = LargestOfTwo(a,b)
30 largestabc = LargestOfTwo(largestab,c)
40 print "The biggest number is "; largestabc
```

Setting up a function for later use is called *function definition*. Actually using it is *calling a function*.

So why are we looking at functions now? For two reasons:

- You often find them as part of tests
- Can you see that the flow of control suddenly jumps from the main program, gets buried in the function then resurfaces back in the main program.

Equals

What could be simpler and less prone to errors and programming mistakes that testing if two things are equal? Just about anything! This section is a catalogue of traps.

The missing =

Some languages allow you to test equality with `if( a = b )...` others insist on `if( a==b )...` to test for equality but let you use a single = to mean set a equal to b and have this inside a condition. Here is an example of what can go wrong:

```
A = 99;
if( A=1 ){print("This should never happen but it does");}
```

Here's how the computer thinks: Set variable A to value 99. Do the bit in brackets first...Set variable A to value 1...Now I need the result of what I just did for the IF...last result was 1 and by my rules of logic 1 is true so do the code block...

Ouch! It might seem weird that `A=1` has a 'result' of 1 but that's how many languages work. This is a bigger trap because with some languages = is used quite happily like this so if you mix languages you're more at risk. Tip: One way to reduce the risk is to try to put constant bits of tests at the front. `if(1=A){...` will upset the interpreter as it will know that it can't assign A to 1. Another way is to get into the habit of always using == for equality wherever you are and let those languages that only use a single = tell you it's unnecessary.

Rounding errors

---

[37] There are lots of loose ends in this example - we'll ignore them for now and press on

Here is another trap:

```
10  INPUT "Input money in pounds and pence";Total
20  INPUT "How many people to share £";Total;" between";People
30  EachGets = Total / People
40  AmountLeft = Total
50  REM Now repeatedly subtract each person's bit
60  IF AmountLeft = 0 then STOP
70  AmountLeft = AmountLeft - EachGets
80  PRINT "Pay out £";EachGets;" Amount left is ";AmountLeft
90  GOTO 60
```

What could possibly go wrong with line 60? If we subtract three thirds or four quarters or n n^{th}s we must end up with zero so the program will finish correctly. Bzzzt, wrong!

Let's see what happens with £10.00 and three people.

(30)EachGets ...£3.33 (40)AmountLeft=£10.00 (50)Remark ignored (60) AmountLeft is £10.00 which isn't 0 so continue
(70)AmountLeft = £10.00 - £3.33 which is £6.67 (80) Print etc. (90) Jump (60) AmountLeft is £6.67 which isn't 0 so continue
(70)AmountLeft = £6.67 - £3.33 which is £3.34 (80) Print etc. (90) Jump (60) AmountLeft is £3.34 which isn't 0 so continue
(70)AmountLeft = £3.34 - £3.33 which is £0.01 (80) Print etc. (90) Jump (60) AmountLeft is £0.01 which isn't 0 so continue
(70)AmountLeft = £0.01 - £3.33 which is £-3.32 (80) Print etc. (90) Jump (60) AmountLeft is £-3.32 which isn't 0 so continue
(70)AmountLeft = £-3.32 - £3.33 which is £-6.65 (80) Print etc. (90) Jump .....(and so on for ever)

But you say that only went wrong because of the rounding error caused by not taking the fractions of a penny into account. OK then (as if you can pay somebody .333 of a penny) how many decimal places do you want to go to?

There's worse news : You may have tested this with £33 and 3 people £10000 and 10 people and it worked every time. It might just be a few *pathological* values that you don't know about which trigger the error months later just when you don't want a phone call in the middle of your holidays. Rounding errors are inherent in floating point operations you might even find that 12.234678 times 54.321 is not *exactly* equal to 54.321 times 12.345678.

Good news:

- Integers don't have these problems.
- You can test for >= or <= to be on the safe side
- You can make money from rounding errors[38]

In practice computer programmers stick to integers where possible and are aware of potential pathological cases with floating point numbers.

Equals what?

We are jumping ahead a lot here and may end in deep water. If I ask you "Do you have the same car as Charlie" what *exactly* do I mean? Do I mean

a the same make/model/colour
or b own the same bit of metal

(If I'd asked "do you have the same dentist" then option b isn't so odd.)

In computing we sometimes meet the same thing because we give labels to things and use those labels as if they were the actual things. In our street Mr Jones lives at No.1 His neighbour is called Doris. (Doris lives at No.2) I live at No.3 and have a neighbour I call Mrs Green. You've already guessed that "Doris" and "Mrs Green" are one and the same person, but suppose I met Mr. Jones somewhere one day and he referred to "Doris" I might not know who he was talking about because I'm not on first name terms with her. Or he might say to me "your neighbour" which makes me think of the chap who

---

38 Research "Salami Slicing" for more details.@@@

lives at No.4! If there are 4 houses in our row of cottages there are 4 people and 6 neighbours - some duplication.

If I give you a photocopy of a document then that's the same document (equal in content) but *also* a different document (separate piece of paper). This is always leading to confusion.

Jumping around

This chapter is called Control structures but where are all the structures? Here they come. These are standard patterns of flow control.

Call and return - Sub-routines and functions

We were just talking about functions. Using a function is called ***calling*** it. When the function finishes it returns to the place where it was *called from* to continue.

Here is a BASIC program[39]

```
10  FUNCTION DoubleIt (N)
11    DoubleIt = N + N
12  END FUNCTION
20  A = 7
30  FOR i = 1 TO 5
40    A = DoubleIt(A)
50    PRINT i, A
60  NEXT i
```

Lines 10,11 and 12 ***declare*** a block of code named DoubleIt which takes a single ***argument***. (Calling it N might give us a clue it is a number - later we'll look at arguments in more detail.) 10,11 and 12 are not executed at this stage, just put by ready to be used later.

Line 40, executed 5 times in the loop, calls DoubleIt. As a result A becomes 14 then 28 then 56 and so on.

Here is how the computer operates : (10) A function definition for future reference. I'll just make a note of the name and how many arguments it needs. (11) Aha! This is how the function will be calculating its result - Just add the argument (whatever that is) to itself. (12) End of the DoubleIt definition. (20) Set a variable location up for a number and label it A. (30) Loop using a variable called i as a counter starting at 1 and stopping after 5 (40) Set A to ... DoubleIt? I don't recognise that word as part of my language - Oh it's a user-defined function...Here it is, it needs a number before I can launch it...And here is the number to use in the brackets. MAKE A NOTE to come back here when I've finished messing about with the function.

(10) One number required - I have just got one number. (11) Add the number to itself ... as this is the result I'll get ready to hand it back to the calling program.

(still line 40) and the result from the function is ... a number I can use to store in A

(50) Print etc. and so on round the loop 5 times.

Basic things you need to know about functions. (We will expand these points in a later chapter)

- They are everywhere
- Sometimes they are called ***procedures*** and ***methods***. In olden days they were called ***sub-routines***.
- You may come across things called 'macros' - a wooly term pretty much deprecated by programmers. If they have names like functions and take arguments like functions then that's what they are.
- A function might *do* things rather than be expected to give a result.
- Variables inside functions (N in the example above) are normally separate from those in the main program. That is say N in a function is nothing to do with N in a main program.
- Normally the number and type of arguments supplied must match the definition.

---

39 Adapted from a working version for clarity of explanation

Using functions is de rigeur

You can have a function call another function and another etc. You can even have a function call itself! In fact this is how many programs work. It is natural to think of programs running 'down the page' in one long stream of computer consciousness. This was found to be a Bad Thing because the code became unreadable with a `FOR I = A TO B` on page 1 of the code and 400 lines later the matching `NEXT I`. This caused all sorts of problems. You don't have just one bone in your body but lots all shaped for the particular job they have to do - The same goes for programs. That's how programmers think.

Review

I never knew there was so much in this topic myself until I started unpacking it to write about it. Another way of looking at this is for you to appreciate that it is like a tent which will fold away into a compact packet of knowledge.[40]

- You can jump using GOTO - but don't
- In order to test things you can compare them using relational operators - but beware of 'equals'
- Once you can test things you can switch program control programatically
- Functions are programming in depth rather than length. The main program passes data to the sub-program, hands over control to the sub-program, then eventually the sub program hands back a result to the main program which can now continue.

This chapter has been rather technical - that's because what we're discussing is the techniques used to structure computer languages.[41]

---

40 Obviously not as compact as it was when you first got it. Have you noticed that manufacturers give erection instructions for tents etc but rarely how to fold back into the carrying bag.

41 Many but not all computer languages work like this. It is too soon to discuss alternatives.

If - Then - Else

We've done some of this already. Each language has a slightly different syntax so you'll need to look up the documentation. The important things to remember are:

- the `else` will be optional
- a program with large numbers of if-then-elses might need a re-design. (More in a later chapter.@@@)
- `then` and `else` normally precede a block of code. Some languages make it easy for you to mistakenly apply the `then` to just the rest of the line when you meant to have a few lines taken together.

Computed loop

You have seen a few examples of loops in BASIC which uses FOR...NEXT.

```
10  FOR SUIT = 1 to 4
20    FOR CARD = 1 to 13
30      ...foo...
80    NEXT CARD
90  NEXT SUIT
```

- `SUIT` and `CARD` are *loop counters*.
- 1 and 1 are *initial conditions*.
- 4 and 13 are *terminating conditions*
- Blocks nest (as you know blocks always do)

We've assumed that we want to add 1 to each counter. While that's exceedingly common we might prefer to count down or in odd steps.

What if we want to do the sort of loop that we use in real life to deal cards: "Keep dealing the cards until none are left"? Now we need to test something other than the loop counter.

```
while (AnyCardsLeft()){
  DealTopCard();
}
```

- There is no loop counter
- Test is at the top of the loop

Having the test at the top of the loop means we might never execute the loop if the condition isn't true at the first entrance. Sometimes we always want to execute the loop at least once.

```
repeat {
  PutNextArticleInBag();
} until (BagIsFull());
```

When packing groceries we might not have a bag until we realise at the PutArticleInBag stage we need one. So the test only makes sense after the block.

Some languages use keywords like D0...WHILE, REPEAT...UNTIL and similar to implement these variations. Others stick to a multi-purpose FOR which works as follows:

```
for ( initialise ; test ; update ) { body }
```

Initialise always runs once. Normally it just sets a loop counter to a handy initial value.
Test checks to see if the loop should continue or quit before each circuit.
Update is run at the end of each loop circuit to do anything that needs to be done before the next circuit.

This is how a Java programmer would write the nested loops at the start of this section.

`for ( int suit=0 ; suit < 4 ; suit++){`
`  for ( int card=0; card < 13; card++){`
`  ...foo...`
`  }`
`}`
	• `suit` and `card` are defined as integers and start at 0
	• `++` is Java (and some other languages) for 'add 1'.

The first line reads: Initialise the loop with a new variable called suit which will be type of integer. For the first and all subsequent times do the block of code in the loop if suit is less than 4. At the end of each loop add 1 to suit (the loop counter).

Our hypothetical Java programmer didn't have to start at 0 and test for 'less than 4' but as I've said, programmers tend to start counting at 0 and feel happier using < or > than = or ==. Here are some variations (Again in Java.)

`for ( ; Shepherds==WatchingSheep ; ){`
`  MonitorFlock();`
`}`
	We don't need to be counting.

`for ( OpenConnection() ; ConnectionIsAlive() ; ){`
`  UseConnection();`
`}`
	We can do complex initialisation

A common use of a loop is to step through the items in an array or list[42]. For zero-based lists (and arrays) the first element has index 0 and the last element will be indexed by the number of items in the list *minus one*. So you will often see code that looks something like

```
for i := 0 to count(TheList)-1 ...  // *don't forget the -1*
```

Switch - Case

Sometimes we want to take one of a number of actions depending on a test. Many languages provide a compact way to do it. The two commonest patterns used are SWITCH ... CASE and CASE ... OF.

Here is an example in Delphi (Pascal). Notice how Square and Rectangle both lead to the same result.

```
case Shape of
  Square, Rectangle : AreaText := 'Base times height';
  Triangle :          AreaText := 'Half base times height';
  Circle :            AreaText := 'Pi times radius squared';
else
  AreaText := 'No method defined';
end;
```

The same task in PHP

```
switch ($shape) {
  case SQUARE :
  case RECTANGLE : $areaText = 'Base times height';  break;
  case TRIANGLE  : $areaText = 'Half base times height';  break;
  case CIRCLE    : $areaText = 'Pi times radius squared'; break;
  default : $areaText = 'No method defined';
}
```

---

42 We will distinguish between arrays and lists in the next chapter. For now just think of an array with all elements used.

Switch in many languages contains a very nasty trap. If we left out the `break` statement in the PHP example control would drop down to the next line (and so on). For example a triangle would Set $areaText to 'Half base times height' then set $areaText to 'Pi times radius squared'
which is not what we want. This is such a troublesome trap that some programmers start by writing out the skeleton with breaks before doing anything else. It also means you *must test every choice*.

Sideline - Style

In the olden days having lowercase was a luxury - if available at all. Literally you could be limited to 0..9,A..Z, and a few special characters. No lowercase, no curly brackets. We were poor but we were happy. Then along came visual displays and matrix printers and the fun could start as different schools of programmers evolved convenient conventions for coding styles. Each felt theirs was 'The One True Style' which lead to religious wars.

As humans we're very good at spotting patterns and assigning characteristics to them. This makes looking at screens and screens of code a lot easier if there is some consistent guide to what's what. When using *Beginner* I didn't have to explain that *6.1.1* was a sub section of *6.1* which in turn was a sub section of *6.* as the convention is universally understood. Since you need to be able to come back to your code in a year or a decade's time[43] you ought to make it easy to read. (Better still, do you really want to be messing about with 'that old stuff' in 5 years time? - Pass it on to some junior programmer with a nonchalant "the code is well documented" and get on with exciting new stuff.)

We have already come across some common style conventions that make code easier to understand.

- Indenting blocks of code[44]
- Using `i` or `I` (and to some extent `j`/`J`) as integer loop counters
- Using 'foo' for 'anything in explanations

Here is a selection of stylistic conveniences which you may see and possibly feel useful. Obviously it will be handy to aim to follow the style adopted by others using the same language if there is a distinctive set of unwritten rules.

Case

- Some languages are case sensitive others aren't.
- Some are pretty strict about the various uses of case.
- Mostly key words can be upper or lower or mixed case. `FoR`, `seLEct`, Mixing `<H2>...</h2>`, and `FUNction` are likely to be acceptable.

Naming conventions

ALLUPPERCASE tends to get reserved for *constants*. (Constants are variables that don't change.)

---

43 On of my customers still uses a large program I wrote 15 years ago. Apart from the people it is the oldest thing in their office.

44 There are still holy wars about *exactly* how to indent and match the beginning and end markers. Stick to one method.

There is a style of variable naming that you either love or hate. You give variable names a prefix according to their type. For example szName would immediately indicate the variable was a zero-terminated-string. (See next chapter.@@@Appx?) This style of usage tends to be favoured more by those who program in C and use system-level APIs.[45]

Java, with it's quite definite naming scheme for classes and methods (just think 'type' and 'function' for now) has had quite an influence in a wider sphere. For example in Java you give a class an upper case letter to start with and methods start with a lowercase letter and end with brackets whether they have any arguments or not. Something that starts lowercase without brackets at the end will be a variable. Here is a Java snippet:

```
Account account = new Account(1234567,'Jim Smith');
account.setBalance(123.45);
```

This reads as : account is a variable of type Account. Create it using 1234567 as the account number argument and Jim Smith as the name argument. With the account variable do the function setBalance[46] using 123.45 as the money argument.

Notice how, because spaces are generally not allowed in names we find alternatives. One which we've used in the naming of Beginner programs is to replace space by an underscore. Another is to lose the spaces and capitalise the first letter of each word. (thisIsHowYouDoIt.)

It is a good idea to give functions names with the action bit at the front - getBalance() rather than balanceGet(),

To conclude this excursion into coding style:

- Some rules is rules - and some aint!
- All aids to communication should be used
- Try to keep to a standard - Find out what the 'local' ones are.
- Look up the manual for specific languages for different styles of comments. Many languages have a program that can look at your code and produce documentation based on the program part and your *strategically placed* comments.

Other ways program flow is switched

We are getting a bit ahead of ourselves here but never mind; we'll deal with them properly later.

Error handling

Let us suppose that your program needs to be connected to the Internet but the connection drops. Deep in the guts of your computer an alarm bell will ring - how does your program handle this kick-in-the-teeth? Nicely we hope - or at least safely. It is considered very bad form to let the unexpected or the rare-put-possible untoward event to crash your program or cause the computer to lock-up. So you tell the part of the program you want to protect how to catch errors. You may not know precisely where they come from and the errors themselves may happen (be 'thrown') at any time. It's a

---

45 If you have to ask what a system level API is you don't need to know. We'll cover it in due course.

46 The setBalance function would have been defined for type Account. In other words any variable of type Account 'knows' how to set it's balance. More in later chapters.

bit like having a safety net - you're not expecting tightrope walkers to fall off - and you'll catch them falling at any time. So the jump out of the normal course is unpredictable, unplanned - but insured against.

Polling and interrupts

When you keep trying to phone somebody but they never seem to be at home - that's *polling*.

When somebody phones you - that's an *interrupt*.

Polling means you have to keep testing - which might mean you can't be doing other things because you're too busy testing. On the other hand you can be getting on with something useful until interrupted by the phone ringing. In computer terms this means that continually polling can lock up a computer, so it is really handy if you can arrange to press ahead but catch (the same as for errors) interrupts. All the same things apply but this time you might say "I'll just finish so and so before dealing with that interrupt"[47]

Review

Repetition can depend on simple counting or testing.

Testing needs to be done at the right place, and fail-safe.

Depending on some condition do this or that will use *if...then...else*. When you have a number of options to chose from you can use *select* or *case* to switch between them.

Sub-routines (functions, methods, procedures are all names for the same thing) are planned diversions from linear flow. They allow you to break up a program into manageable pieces that can be developed and tested independently and then assembled.

If you've done a bit of programming before this chapter will have been a breeze. These basics are important because being fluent in fundamentals means you can apply your brains to the more challenging stuff.

---

47 There are some interrupts you can't ignore - Non-Maskable is the technical term. Hungry cats for starters!

5. First steps in coding

You have already put you hands on a keyboard to write Hello World in HTML. Some would say that HTML isn't a 'proper' programming language and they do have a point, but as we'll see down the road, web pages are a patchwork of HTML stitched together and overlaid with 'real' programming.

In this chapter we'll start you off on your chosen programming languages 'YCPL' just so you become familiar with how to bridge the gap between "I want to do foo" to seeing a result on the screen.

In later chapters we'll return to the thoughtful bit of programming - Then you'll be able to try out concepts we've discussed using the skills learnt in this chapter.

Development environment

Here's how programming works

PROGRAMMING_OVERALL_METHOD

1. *Idea*
2. *Research and design*
3. *Write the code*
4. *Run the code*

4.1 *Get the code to compile and without the computer complaining*

4.2 *Put the program to work*

5. *Repeat 3 and 4 until results are satisfactory*

When you wrote Hello World in HTML you used a text editor for stage 3 and a web browser for stage 4. Luckily all the knowing how to run HTML is contained in your browser 'for free' so to speak. With other programming languages you'll need to be telling the computer how to interpret your code. Also there are programmer productivity aids to make editing code easier. Many of these are personal choice and I expect you to experiment a bit before finding the one that suits you and your pocket. An Integrated Development Environment or *IDE* is a sort of programmer's workbench where your reference books, bits of work in progress, tools, test instruments and notes can be kept in one place. As you know, having the things you use all the time ready to hand is a great benefit to getting a job done quickly and efficiently. As you mature so will your workbench.

How to get started?

Good question. Whatever programming language you intend to get stuck into will require you to set up some way of at the very least passing some code to a program that does something with it. Unless you've already got this installed and working this is going to take an hour out of your life.[48]

---

[48] It is relative : My first program took 3 weeks to get returned from the data centre.

What programming languages is the best? or What programming language should I use? Sorry, I have no more idea than what you should wear this evening or where you should go on holiday. Have a look at the appendix on languages.

The ultimate objective is to be able to run a program that displays the magic words "Hello World". I only wish I could be more helpful but like modern furniture it needs a lot of self assembly - and far too often there are bits missing.[49] For starters you don't need all the development tools a professional would use so try to keep things as simple as possible.

There is a cheat which is available to most people without any downloading of IDEs and so on if all you want to do is follow the examples in this section: See the Appendix - How to code with Javascript.

HW_IN_CODE_1

----- Install and run programming language -----

0. With your chosen language

1. Install necessary bits

- Details vary from language to language and also IDE to IDE

- Don't forget the manual

2. Write a program to display "Hello World"

2.1 Read the manual for instruction - It should have it in there

2.2 Run to get a result

2.3 Save the program

2.3.1 You probably want to create a special directory to play in

2.3.2 And implement a logical naming scheme

3. Relax and celebrate : That's the tricky bit over.

2.3 is interesting. Firstly some programming languages require you to put certain files in certain places. After a while you should be thinking of adapting the configuration to allow you to have a number of project zones. Secondly some will expect you to observe file naming conventions. It's a really good idea to give your programs names that will let you retrieve them later. Whatever you do, don't use spaces in file names.

HW_IN_CODE_2

----- Make sure you can use numbers -----

0. With the Hello World program you created in HW_IN_CODE_1

0.1 Make sure you can load it from file

1. Alter the code to display the results of 1.1+2.2+3.3

---

[49] I recently tried to install 3 IDEs and 4 frameworks (See glossary or wait until later) with 28% success rate after a lot of trying. Now if somebody with my experience has these problems it doesn't bode well for beginners.

1.1 Something like `sum = 1.1+2.2+3.3`
1.2 So the full result is "Hello 6.6 World"
2. Save the program under another name
3. Make sure you can still run your first program.

In 1.2 I've set you a tricky task: To mingle numbers with strings. If it isn't 'obvious' then you need to find a way to convert a number to a string. Some languages are very forgiving, other strict.

HW_IN_CODE_3
---- Using the documentation ----
0. With the original HW program
1. Alter the result to : "Hello World", then on the next line "The time is " followed by the current time.
1.1 How to start a new line?
1.2 How to find the current time?
1.3 How to format 1.2 in 'hh:mm' style?
- From now on I'll assume you'll manage your program files
- without specific instructions

If you found the HW_IN_CODE series a struggle - do not worry. You've learnt a lot of important matters on the way and they won't be troubling you again. Take it from me that it is only the unfamiliarity of the task that makes it a struggle and in a day's time you'll scoff at the simplicity of it all.

GOODBYE_COMPUTER
---- There are other things in life ----
0. With computer
1. Switch off
2. Quickly scan the next paragraph
- Go and chill-out with a glass of your favourite tipple

In the rest of this chapter we'll be doing simple programming exercises. They are carefully in sequence and you should do them all. The objective is to get you familiar with your programming environment and introduce you to some of the intellectual and practical skills that programmers use every day.

5.1 Average

This exercise is an opportunity to use arrays, integers and floating point numbers. Finally we look at how to find a maximum and minimum of a list.

EX_5.1_BY_HAND
0. With pencil and paper
1. Write down a handful of integer numbers in a list
2. Count number in list
3. Add up numbers in list

4. Divide sum by count to get average
5. Report result

How are we going to convert that to a computer program?

- You already know about arrays - They need to be defined, filled and indexed
- You already know how to browse the manual

I have written some programs as illustrations.

- You will need to adapt for your particular system. All these programs have been tested but don't be surprised if you get error messages when you try to run them. the reason for giving you these listings is so you can understand the underlying structure and possibly pick up the coding style appropriate for your chosen language.
- Notice how the variable used to add up the total is explicitly set to zero. Often you can expect this to be done for you - but experience shows it is always best to do it yourself.
- Different styles of names have been used for example `sum` and `SUM`, there is no real significance in this.

BASIC

```
10   REM Prepare array with any old data
20   COUNT = 8
30   DIM A(8)
40   FOR I = 1 to 8
50     A(I) = I
51     REM BASIC is unusual in using round brackets for arrays
60   NEXT I
70   REM Array is now ready for use
80   SUM = 0
90   FOR I = 1 TO COUNT
100    SUM = SUM + A(I)
110  NEXT I
120  AVERAGE = SUM / COUNT
130  PRINT "Sum ";SUM; "   Count "; COUNT; "   Average ";AVERAGE
```

Delphi/Pascal

```
function ComputeAverage:single;
{ returns a single precision average of the first 8 counting numbers }
const
  COUNT = 8;  // Constants often capitalised as a matter of *style*
var
  a : array[1..COUNT] of integer;   // all variables must be declared
  i, sum : integer;                 // before being used
begin
  // Initialise
  for i := 1 to count do a[i] := i;
  sum := 0;
  // total
  for i := 1 to COUNT do sum := sum + a[i];
  // result
  result := sum / COUNT;
end;
```

- Display results in a string format using something like `format('%3.3f',[av]);`

- Check `sum / COUNT` works correctly. Delphi can be snotty about dividing with integers. (It won't! - Look up typecasting.@@@)
- `{ ... }` can be used in Delphi as comments. In all other programming languages known to man { ... } are vital program block delimiters.
- Delphi requires all variables to be specified before the code can begin.

PHP

```
/* Initialise array */
$a = array(1, 3, 66, 7.2, -5.55, 8, 9, 10, 11);
$count = count($a);  // count is a built-in PHP function
$sum = 0;
// do the totalling ...
foreach($a as $n){  // foreach is part of PHP
  $sum += $n;
}
$average = $sum / $count;
print(" $count items in array total $sum. Average is $average");
```

- /* ...*/ are commonly used to indicate comments
- All variables in PHP start with $
- We've initialised the array as the same time we dimensioned it.
- PHP is very laid back about what we put in arrays - we can mix and match types to our heart's content.
- Notice that we have *found out* how big the array is
- foreach is a very useful ***enumerator***. (More on enumerators in a later chapter.@@@) It says let's call each element $n and hand that to the following block - and repeat for all elements.
- += is a shorthand for foo = foo + bar that is quite common in programming languages.
- PHP is laid-back about printing - but notice the *double* quotes.

Javascript

```
// ----- fill array with any old data -----
COUNT = 8
a = new Array(COUNT)
for (i=0;i<COUNT;i++){   // js arrays start at 0
  a[i] = i + 1           // when i=0 1st element value is 1
}
sum = 0

// ----- step through array adding up -----
for (i=0;i<COUNT;i++){
  sum = sum + a[i]
}

// ----- results -----
average = sum / COUNT
document.write("<br>Average of " +
COUNT + " items is " + average)
```

- Trap: Javascript statements don't end with `;`
- In the last line the `.write()`

Use of objects and methods in a nutshell

```
Susan = new Person(24,Blue,Blonde)
Terry = new Person(26,Brown,Black)
```

`Susan.getAge()` gives 24

`Terry.getEyeColour()` gives Brown

`Susan.setSpouse(Terry)` marries them

`Terry.isMarriedTo(Susan)` gives true

function (notice the leading period) is applied to the document which has been pre-defined for us.
- Javascript is very forgiving, but sometimes too trusting, about mixing numbers and strings
- COUNT in upper case tells us immediately it is a constant.

Java

@@@

Modifying 5.1

Edit the code to add a maximum and minimum and show these as well.

1. Initialise two variables
2. Test and set in the analysis loop. This would follow the pattern:
   ```
   if a[i] > max then max = a[i]
   ```
   adapted to your particular language.
3. Report

What could possibly go wrong?

- The loop counters don't start at the right place (0 instead of 1 or vice versa)
- The loop counters don't finish at the right place.
  This type of programming error (called *off-by-one*) is one of the most common and will dog you every day you program. Like a restless poltergeist it will press the wrong key from time to time and cause horrible grief. Be strong and resist! Most programmers find double checking beats a lucky rabbit's foot.
- What happens if the number of items is zero. Do the loops (not) work ok? What happens when we divide sum by zero? Empty arrays and zeroes happen all the time - often that's part of the plan. However there may be unusual circumstances where it is even more important than usual to avoid this trap. Suppose our program was monitoring communications. If zero bytes were received in a period that's probably when our diagnostic reports would be most useful - but then the program crashes with a stupid[50] 'divide by zero' message.
- Some of these languages treat `Sum` and `sum` as different entities. discover a naming style that suits you and the programming community you inhabit and try your very best to keep to it at all times.

Review

Let's take time out a moment to see what this chapter has been about so far.

- You've found a way of writing instructions and getting the computer to swallow them.
- You've thought through a 'how to make it happen' procedure and implemented it as a computer program.
- You've seen a number of programming styles which are almost different dialects than languages. Overall they are very similar, but the details are interestingly different. (As we continue differences will become more interesting.)
- You've experienced some of the practical difficulties. If it is any consolation, my examples had to be err... 'tweaked' to get them to a state I was happy with. Old age is my excuse - but in reality a lot of programming is about spotting and dealing with bugs.

---

50 Stupid as in "stupid programmer"

Now we'll change the emphasis from coding to designing the program method. I'm still expecting you to get all programs to run as described because you need to get a few hundred lines of code under your belt before you start pick the right sort of approach every time.

Bear with it - Every mistake you make now should save repeating it in the future - or at least you'll recognise it with a 'Duh! not again'. That's what learning is all about. If you can't hack this section then give up programming. Actually, typing code will soon become second nature - it is the thinking of the best overall plan that will sort the men from the boys.

5.2 Sorting

Being able to sort a list is often very useful. There is more to this than the aesthetics of having your address book in alphabetical order. Many times there are good business reasons[51], sometimes a method will only work if all the same kind of thing are lined up together. A lot of languages will sort for you or you can get sort functions to add-in off the shelf. Doing sorting well is tricky for two reasons : Firstly there isn't always one 'best' method (but there are too many easy to program but excruciatingly slow methods). Secondly it is easy to make mistakes when programming.

For this exercise we will take as our task sorting the days of the week alphabetically. What we will discover is that we need to think beyond the way we'd use paper and pencil because computers are a lot better at juggling data items in memory than humans.

```
EX_5.2_BY_HAND
0.  With pencil and paper
1.  Write the names of the week in a list down the page
1.1     Starting at Monday
2   Draw a second column next to the days of the week
2.1     Call this 'sorted column'
3.  Find day of week that appears earliest alphabetically
3.1     Look for first day name not crossed off as 'candidate'
3.1.1       Make a note of row number - 'candidate index'
3.1.2       For each following row do the following
3.1.2.1         See if the candidate day is later than this day
3.1.2.2         If yes then this day becomes the candidate
3.1.3       Repeat 3.1 until end of list is reached
-   The first time through this loop the day should be Friday
4.  Move candidate to the sorted column
4.1     Write day name in sorted column
4.2     Cross it off the first list
5.  Continue 3 and 4 until sorted column has 7 items in it
5.1     Keep count of items in sorted column
```

---

[51] We will talk about 'Business logic' in another chapter@@@.

5.2 If that count is less than 7 then loop back to 3

6. Report sorted column

There's no real need to actually do this by hand so long as you're satisfied that it would work. This is a repeated 'find the lowest' (which we did at the end of exercise 5.1) with the added complication of adding it to a new list and somehow deleting it from the original list.

Algorithms

We will deal with algorithms in a lot more depth in a later chapter@@@ really?. The overview is this:

- An algorithm is a posh name for method or procedure
- Some algorithms are 'better' than others. 'Better' depends on what you mean by it: Fast, compact, reliable, works for large amounts of data, only uses a small amount of memory, can be proved to only need certain resources as a worst case.
- Algorithms are usually defined in pseudo-mathematical or logic terms for purposes of proving they work...
- ...then translated into real code which can be strictly monitored

A sort algorithm suitable for a computer

Overall idea : If we started at one end of a list of items and worked our way to the other all the time carrying with us the largest so far then we would end up with the largest ready to dump at the end. Now we can ignore this end one in future sorts and do the same for the list again but this time for one fewer items.

list of N items
list of N-1 items + highest
list of N-2 items + 2nd highest + highest
list of N-3 items + 3rd highest + 2nd highest + highest
etc. until
1 item followed by ordered remainder

As an algorithm that's pretty convincing from a will it work and is it guaranteed to work under all possible conditions point of view...
...So let's try the second stage : putting it into code:

```
const
   N = 7;  // No of items
var
  i,j : integer;
  dow : array[0..N-1] of string;
  tmp : string;
begin

  dow[0] := 'Monday';
  dow[1] := 'Tuesday';
  dow[2] := 'Wednesday';
  dow[3] := 'Thursday';
  dow[4] := 'Friday';
  dow[5] := 'Saturday';
  dow[6] := 'Sunday';

  for i := 0 to (N-2) do begin
    for j := i+1 to (N-1) do begin
      if dow[i]>dow[j] then begin
        tmp := dow[i];
        dow[i] := dow[j];
```

```
        dow[j] := tmp;
      end;
    end;
  end;

  // report these by outputting to a Tmemo
  // (ie print 1 line at a time)
  for i := 0 to N-1 do begin
    m.lines.add(dow[i]);
  end;
end;
```

The first part of exercise 5.2 is to code this in your particular programming language. When you've got the typing mistakes and other silly things out of the system - what do you get? Is the list alphabetical as promised? It should be.

Notice how the computer program could sort the list in-situ without needing another array.

What could possibly go wrong?

Apart from the usual off-by-one and typing errors?

Err... How about: The code given above as a template doesn't do what the algorithm given said it should do! In fact when you look at it can you understand how it works and if it is just lucky-so-far or if it too will give a guaranteed result.

Try fiddling with the data given to the program to see if it always gives a sorted list. Don't forget to alter the number of items in the list because there might be an issue with odd or even numbers of items. OK so it *looks like* (after a handful of tests) that the code delivers the goods - But is "looks like" good enough? in the words of the old song: No No A thousand times No!

We will address this issue properly in a later chapter - But the essential lesson is that you can only prove the correctness of an *algorithm* and *from there* validate your code. In this example we have what purports to be a sort program but not the algorithm to go with it. (We've got a *different* algorithm.)

> Tough stuff this algorithms business eh? Congratulations - most programmers wouldn't know how to validate their code if it sat up, wagged its tail and begged. At least you now know it goes high up your agenda.

5.3 Digital roots

If you add up all the individual digits of a number (adding this up in turn if necessary until you get a single digit) the result is called the digital root. For example the digital root of 789 is 6. (7+8+9=24 ... 2+4 = 6) I haven't actually come across a situation where this has been needed but it will serve a very useful pedagogical purpose.

EX_5.3_BY_HAND

0. *With pencil and paper*
1. *Write down number*

2. Add up all digits
3. If < 10 then report result, Finish
4. Repeat procedure using this number.

The important bit of this is line 4. The procedure says "re-do using your recent result as your argument". This snake-eating-its-own-tail is called ***recursion***. The reason we ought to look at this is that it can make for neat programming, but needs to be handled with care. A simple example is if we wanted to search all the files on a disc. We start at the root¤ and list the files and sub-directories. Then repeat for each sub-directory (which of course lists all its sub-directories. Another use is where we split one problem into two smaller problems and then those problems are subdivided again and so on until we've got down to size that is easy to solve. For example suppose we want to sort a list. We could put all the large items at one end and all the small items at the other then split into two sub-lists and repeat for the large (into very large and not so large) and small (not so small and very small) portions and then again (eg very small into teeny-weeny and microscopic). Each time we're using the same method in finer and finer detail[52].

If we had a function called AddDigits() how could we code this?

```
function DigitalRoot(WhateverNumber:integer):integer;
var
  n: integer;
begin
  n := WhateverNumber;
  while(n>9){
    n := AddDigits(n);
  }
  result := n;
end;
```

- The function definition tells us it takes one integer parameter (WhateverNumber) and returns an integer to the calling program.

What we have here *is not* recursion, just repetition. We also have *two* functions: `AddDigits()` and `DigitalRoot()` But nevertheless we can see how it works and be confident that it works.

Here is an all-in one function which *does* use recursion.

```
function DigitalRoot(const aNumber:integer):integer;
var
  dr, n : integer;
begin
  // *Instrumentation goes here to report argument*

  // this bit does the adding of the digits
  dr := 0;
  n := aNumber;
  while n > 0 do begin
    dr := dr + (n mod 10);
    n := n div 10;
  end;
```

---

52 See the algorithms section for more.@@@

```
    // this is where we decide if we've finished
    // or need to crunch the number a bit more
    if dr > 9 then result := DigitalRoot(dr)
              else result := dr;

    // *Instrumentation goes here to report result*
  end;
```

- `mod` is the modulus operator. Modulus gives 'the remainder when divided by'. So for example `8 mod 3` is 2, `9 mod 3` is 0, `10 mod 3` is 1, `11 mod 3` is 2. Anything 'mod 10' gives the last digit.
- `div` is Delphi's integer division operator. Basically any fractions are thrown away. For example `8 div 3` is 2. You may think that's a rather senseless thing to do, but let's see...

In Javascript the adding of the digits loop looks like this:

```
while(n > 0){
  digit = n % 10
  dr = dr + digit
  n = (n - digit) / 10
}
```

Why is this is a lot more complex than the Delphi/Pascal version? The reason is that Javascript doesn't know that `n` must always be an integer and doesn't do integer division. Let's *trace* the code with n = 345.

digit is 345 modulus 10 which gives 5
Add 5 to dr
345 - 5 gives 340 ... divide by 10 gives 34

Which is now the next integer-a-like `n` to work with. Another way of doing this is

```
while(n>0){
  dr = dr + (n % 10) // note brackets
  n = Math.floor(n / 10)
}
```

Which relies on a built-in function that drops the fractional parts of a number.

- The `*Instrumentation*` markers show where you can add some reporting to follow the program flow. *Instrumentation* is a term used to describe special code which reports on the workings of a program. In this case we will want to print a line which says "Argument is :"+aNumber at the start and "Result is :"+result at the end. (Details depend on YCPL.) Sometimes you remove the instrumentation completely, other times comment-out to leave a clue to anyone looking at the code later what was worrying you, or provide a switching mechanism along the lines of `if(areWeInDebugMode){... do instrumentation ... }` We will look at debugging later on.(@@@ Where?)

- *Before* running the program with instrumentation, jot down what you expect to see. After running with instrumentation were you surprised? This is why recursion is tricky, because the flow of control isn't always obvious.

What could possibly go wrong?

- If we make some error in the return condition we could recurse for ever.
- If we make some other logic error almost anything could happen. Don't even think about poke-n-hope programming with recursion. You must be rigorous. That's why the study of algorithms is important.
- In the Javascript code for adding the digits we could come a cropper with rounding

errors if n - digit / 10 isn't *exactly* an integer due to the accuracy limitations of binary arithmetic...

... so why don't we do belt and braces an use the `Math.floor()` function as well? Because the result of dividing by 10 might be a minute fraction *under* and `floor()` will see this and think we want one *whole number* less.

These mathematical errors may never happen during testing. Perhaps we test using 10,000 numbers and everything works fine, but would you fly in a computer controlled aeroplane where there is "less than 1 in 10,000 chance" of a program crash. (And that's only in one tiny fragment of code.)

5.4 More dice

One of the useful things that computers can do is perform simulations to let us understand how systems work. For example after trying 18 times to get through to the Inland Revenue telephone call centre, I though what's the chance of that - they must have far too few staff. I should think that's obvious, but before going much further we need to put some figures to say something like : "Need 80% more staff to give a 80% chance of being answered on two attempts".

Think of the economics. Let's suppose you spend 2 man-hours on programming and the rest of the day collecting some actual data and improving the model. Your results will say how many people to employ week-in week-out. So that's one good reason why good computer people should be paid well. It is also a reason why suits hate programmers because the programmers can point to the knots in the system, often exposing embarrassing false assumptions. Managers don't want to get involved with cans of worms like that.

The object of this exercise is to do a simulation and look into how to display results. Also, now you can read code as well as write it, this is an opportunity to see what is involved in translating code.

The subject of the simulation is to see what the probability of getting a particular score with two dice is. All we'll do is throw the dice 1000 times and count the number of times we get each score.

I've written you a ready-to-run Javascript program. Type it into a text editor : typing practice is a good thing - you can see how accurate you are then save the file as `ex54.htm` and look at this file in your web browser. (See Javascript appendix for more details.) Or you can re-write it in your chosen language; not word for word or even line by line, but section by section after twigging what's going on in that section.

- The last section (with the <table border=1> in it) is HTML specific, but if you ever think you might be displaying tabular information on a web page you should check out the nesting of the blocks (`td` inside `tr` inside `table`) which is an extremely common web page programming task.

```
<html>
<head>
  <script LANGUAGE="JavaScript">
    // ******************
    // ***** Heading *****
```

```
// ******************
document.write("<h2>Exercise 5.4 - Display</h2>")
document.write("<hr>")

// ---- define parameters of experiment ----
NOOFDICE = 2
NOOFTHROWS = 1000


// return a random number 1 to 6 inclusive
// ---------------------------------------
function ADiceThrow(){
  return 1 + Math.floor(Math.random()*6)
}

// Hack to align numbers. Return a two digit
// string version of the integer parameter
// ---------------------------------------
function TwoDigitInt2Str(i){
  s = "___"+i
  sl = s.length
  return s.substring(sl-2,sl)
}

BAR = "#####################################################"

// ---- initialise array to keep hits in ----
maxScore = NOOFDICE*6
hits = new Array(maxScore+1)
for(i=0;i<=maxScore;i++){hits[i]=0}

// ---- run the experiment ----
document.write("Testing " + NOOFTHROWS + " throws with "  (cont. next line )
+ NOOFDICE + " dice<br>")
document.write("<hr>")
for(t=0;t<NOOFTHROWS;t++){
  score = 0
  for(d=1;d<=NOOFDICE;d++){
    score = score + ADiceThrow()
  }
  hits[score] += 1
}

// ---- report the results in three ways ----
for(i=NOOFDICE;i<=maxScore;i++){
  document.write("Score of " + i + " thrown " + hits[i] (cont. next line )
+ " times (" + (hits[i]*100)/NOOFTHROWS + "%)<br>")
}
document.write("<hr>")


for(i=NOOFDICE;i<=maxScore;i++){
  s = TwoDigitInt2Str(i) + "|"
  percent = (100 * hits[i])/NOOFTHROWS
  s = s + BAR.substring(0,percent)
  document.write(s + "<br>")
}

document.write("<hr>")
document.write("<table border=1>")
for(i=NOOFDICE;i<=maxScore;i++){
```

```
      percent = (100 * hits[i])/NOOFTHROWS
      b = BAR.substring(0,percent*(NOOFDICE-1))
      document.write("<tr><td>"+i+"</td><td>" + b + "</td></tr>")
    }
    document.write("</table>")



    // ****************************************
    // ***** flag end of program reached OK *****
    // ****************************************
    today = new Date()
    document.write("<p><fontcolor=green><small><i>[" (cont. next line)
    +today.toLocaleString()+"]</i></small></font>")
  </script>


</head>
<body>
<!-- all the work goes on in the javascript in the HEAD -->
</body>
</html>
```

When you've got the program to work after a fashion you can look at the code above again and notice the following practices. (From top to bottom)

- The parameters we were initially given were 2 dice thrown 1000 times. Now isn't the first thing you want to fiddle with these parameters? Of course. Wouldn't it have been easy for somebody who didn't think like a programmer thinks to have written the program with fixed numbers. Doing that is called *hardcoding* which is often a Bad Thing[53].

Programmers always think how a program might be re-used and developed.

I've lost track of the times I've said to a client: "Are you really *really* sure this is all you want to do?" to be told "yes" - only for events to require just the extensions I envisaged. But I'd allowed for that anyway and spent a fraction more time designing-in extendability so there wasn't much grief. Users are often the least qualified to understand what they could do given new tools. Never mind - If we didn't have suits all the flip chart makers would be out of jobs.

- Why such simple functions?
  Surely it would have been less bother to put this code in the main body?
  - Throwing the dice is a bit of a tricky thing to get right. In Javascript the random() function returns a number between 0 and 1 which we've got to convert into 1,2,3,4,5 or 6 and be absolutely sure that every score gets the same percentage of hits.
  - Where you want to do something a little fiddly which is a distraction from the body of code it is handy to stick it into a function so that when trying to read the main code we can concentrate on one level at a time.
- random() functions in different languages work in slightly different ways. You must

---

53 Strangely you never hear of 'softcoding'. If you do that's a Suit-alert.

read the documentation. One interesting feature of many languages is that (in a very complex way) the next number depends on the current number. This means that it is sometimes possible to repeat the simulation with exactly the same *pseudo random* numbers.

- `TwoDigitInt2Str()` is a funny thing. For drawing a bar chart if the numbers on the left are sometimes one digit and sometimes two this distorts the alignment. This is an attempt to deal with that problem by always returning two characters by padding single digits with a leading underscore[54]. What a mess! Luckily most languages have built-in functions for tidy formatting. By the way, the '2' instead of 'two' in the function name is often seen but not a very good idea.
- Note the `hits` array is explicitly set to zero - generally good practice
- Reporting something at the start (and possibly some progress counter) is a good idea when you may be waiting a while for a result.
- `foo += bar` is shorthand for `foo = foo + bar`
- In this program we report the results in three ways (split by `<hr>` which is HTML for horizontal line)
  - Firstly simply listing the figures
  - Secondly in a crude bar chart. Nowadays many languages give you better graphical drawing facilities, but as you see this is quite effective.
  - Thirdly the particular features of the language (actually HTML) are exploited to improve the presentation. Replace `border=1` with `border=0` for extra prettiness. A cool feature added to the this bar chart is magnifying the bar scale so we can see more detail when there are more dice (and hence a wider spread of scores and so smaller percentages).

The effectiveness of a program often depends on the quality of the *user interface*. We'll look at this in more detail later, but for now you might want to see how you can produce a cool display using your chosen language. (But be warned, that could be quite a task for a beginner - don't get bogged down in complexities just now.) One thing you could try is indicating a horizontal scale. (One cheeky way is to replace every tenth `#` in `BAR` with a `*`. Often keeping things simple means you can concentrate on the guts of the task rather than spending ages just to put some pretty ticks in.)

What could possibly go wrong?

- The `ADiceThrow()` function could be biassed. How can you check this out? How about 1 dice(die) and 200000 throws? Hey! Parameterising the number of dice and throws was cool wasn't it.
- The `BAR` string used to draw the chart might not be long enough.

> Just so you don't feel too bad about spending ages trying to get a program to work, you may be interested to know that it took me ages to get `TwoDigitInt2Str()` working. A real hitting head against a brick wall situation. I was using substring() the wrong way. Actually, my annoyed response when I finally read the documentation was "for goodness why can't it work like every other language's substring function!". Grrr!

[54] Underscore because when rendered in HTML a space would get 'lost'. This is a horrible hack not a recommended procedure.

5.5 ISBN

This is a simple final exercise. By now you should have all the skills and knowledge necessary to crack this in five minutes.

You'll have seen the ten digit numbers and barcode on books called the ISBN.[55] The last character is a *checksum* derived (like we did the digital root) from the first 9 digits. The object is to trap typing mistakes so that a mismatch between the *computed* and *given* checksum will raise an error alert.

The method used to calculate the checksum wants to try to catch common typing errors such as transposition("ba" instead of "ab") so a simple digital root style of adding all the digits won't help very much as 12345 gives the same result as 54321. There are lots of different checksum methods, the one the ISBN uses is:

```
The sum : (digit 1 * 10) + ( digit 2 * 9 ) + (digit 3 * 8 ) + ...
(digit 9 * 2)  + checkdigit must be divisible exactly by 11
```

There is a wrinkle : Sometimes the check digit has to be 10! X is used to represent this.

- Already you'll have seen that the checkdigit part of the sum is just an extension of the rule used for all the other digits (ie digit 10 * 1)

You're on your own to code this one but here are some points to note:

- If your language conveniently allows input[56] then write your program to take a user input and report if the number is OK.
- What if somebody puts in a wrong-length ISBN?
- The way to test is? ... To take some real ISBNs from actual books. Is one book enough? 10? 100? Are you testing to trap mistakes as well as good ones? We'll look at these questions in more detail later.
- Probably the best way to represent the ISBN is as a string rather than a number. This will give you a chance to read the manual on :
  - How to examine the nth character of a string. Tricky due to possible zero-base-indexing but quite a frequent job and worth getting out of the way
  - How to find the first occurrence of one string in another

  Using these you can do something like

```
for each character in ISBN
  find how many places into "0123456789X"
  multiply this index by the appropriate weighting and add to total
If total mod 11 is 0 then report OK
```

What could possibly go wrong?

You should be getting the hang of this now. WCPGW is usually a matter of 'certain inputs might cause a problem.' Go back and look at your test data. Have you tried really bizarre inputs. What if the input is 2000 characters long? No really, this is important. You'd be amazed at the number and seriousness of the security breaches

---

55 International Standard Book Number

56 Javascript can be hacked and is relatively simple *when you know how* but is not worth bothering with here.

this has caused.

Chapter 5 conclusion

Phew! What a chapter that was. Your brains are probably melted from overload. Future chapters will not be so code-intensive. Just take a moment to go over the whole chapter again. You'll see that what was difficult is now easy and that shows you've learnt the skills and taken in the knowledge.

- You've got the hang of the practicalities of coding, and no doubt experienced the 'joys' of debugging. One of the things you may have found is the level of concentration required can be quite intense. How people program in open-plan offices full of distractions is a mystery to me.
- As time goes by you'll develop all aspects of your programming environment, both on the computer, on your bookshelves and in your office. Don't be afraid to experiment. If you program through the night (a lot of people prefer it because it is quiet and there are fewer distractions) you may want to change your display so it shows bright letters on a dark background. There are all sorts of things to try: A more comfortable chair, a heater for your feet, the right spectacles, trackball instead of mouse, two screens, and that's without even considering programming tools and IDEs.[57]

> I work from home and found that things were happening outside which I couldn't quite see because my chair was lower than the window. Getting up to scan the scene was a big time waster. Solution: Raise computer bench and chair by 18 inches so now I can glance out to see the activity outside, wave to the neighbours when I see them passing and so on. (And of course just gaze out of the window putting off work a bit longer.)

- You've discovered that programming languages seem to have a basic common core which means you can pick up what is going on fairly easily.[58] To get a feel for their relative strengths and details takes experience - but don't be afraid to have a peek at other languages to see what they have to offer, who uses them and why some people have strong opinions for and against.
- On top of the strict discipline of programming language syntax you have seen that you have to take a rigorous approach to stitching together instructions. Remember that sort program (5.2) which didn't even use the specified method! You will never write perfect programs but a calculated approach will save you hundreds of hours of grief, not to mention the benefits of giving the general public software they can rely on. Your whole programming life (and probably all of it once you click with the concept) is about reality checking an unreal world inside a computer[59].
- If a programmer was asked to make a machine for bottling lemonade they would immediately think "Hey, this could and should be adapted for other fizzy drinks ... I'll start by building a generic fizzy-drink-bottling machine and tweak it for the specific

---

[57] And a proper shelf on top of the warm monitor for the cat.

[58] At least those we've looked at so far

[59] I've cut out an exercise in reality checking because enough is enough and by this time you've had more than enough - Hey! that's a reality check... really? Yes, you're real, this book is just information, these words are just made up in the hope they make sense in some wider picture. they only *describe* the real world they are not actually the real world.

case of lemonade." (Programmers enjoy solving problems but hate work; that's why, if for a small extra investment they can deliver twice or ten times as much they go home happy.)

- Once you understood the concept of what the problem was you could then use your knowledge to implement it in many different ways. There is often no one Right Way to write a program - Which is why programmers need to read books like this to give them the breadth of outlook in order to exploit a range of techniques.
- WCPGW? On the one hand a programmer is thinking in theoretical terms about the general case and how flexible their program should be - that's fine... On the other they have to deliver a specific solution that will work every time. That requires mental agility[60]. By the way, WCPGW is something you get from experience and reading about the experiences of others. Don't expect to ever be able to smoke out every gremlin.

Important

- Details matter to programmers. So does cleverness. So does delivering what was asked for. A programmer should be a master of practicalities, precision and performance... which is why they don't get on with suits. (What are the characteristics of suits? Assumptions, vagueness and excuses.[61]) Take it from me[62] that the only way to deal with suits is to stick to facts, never *ever* compromise on the facts, don't give into low level or high level bullying and you'll start to get respect - or if not respect, be tolerated[63]. (You won't get promotion ... *especially* when you save everybody else's bacon.)

Finally: Well done to get this far. I'm sure many readers have given up. If you're still with me then you have got over the worst The rest of this book still requires mental capacity but isn't so intense as this last chapter.

How Real Programmers think.
That's what this book is about....

...Should I have made that nice warm shelf over my monitor large enough for *both* cats? (WCPGW - You guessed - I Didn't - There is one sulking and one glaring as I write.... Fur may yet fly...)

6. Data structures

60 For what it is worth it is my opinion that some people are better at mental agility than others, but it can be developed, then decays with age.

61 And bullying. You can't bully a computer ... which is why suits look for some other target... guess who?

62 Have a guess: Do I speak from experience? Could I Bore for Britain on this subject?

63 If you are not tolerated and you know your stuff then you are dealing with, shall we say, a bounder. A few months after you leave they depart under a black cloud. Bounder-suits (as opposed to bumbler-suits) have antennae that will instantly detect the one person who is always doing reality checks.

You know what's coming: It's going to be technical with lots of diagrams. No and No.

By the end of this chapter you'll have a clear grasp of what data structures are... but you actually know about them already. Probably you don't know you know!

In the next chapter will deal with the way data is represented internally in objects. In this chapter we'll consider how 'things' can be related.

A trip to the supermarket

You write out a shopping list[1] which may be structured with similar things together or be a list of shops to visit each with sub-lists[2]. So you drive to the shops, take your turn at getting into the car park[3] and find a suitable free parking space[4]. Now you take a basket from the top of the pile (or a trolley from the 'back' of the nested line) [5] and enter the supermarket. Firstly you head for the fruit-and-veg section, in particular the fruit sub-section and find the apples and then the Granny Smiths[6]. Now for the remainder of the things on your list you pick a suitable aisle[7] and proceed along it until you get to the items you want[8]. Now you wait in line at the checkout and put your items on the conveyor belt [9]. Now you pile your items into bags[10] probably finding that you need more bags than you brought with you.[11] You find your car in the car park[12], put your bags in the boot[13] and go home. At home you ask one of Mummy's Little Helpers to unpack the bags and put the items in suitable cupboards, shelves etc.[14]

Discuss

When you're making up a shopping list using pencil and paper you are not strictly limited to sticking the next item you think of at the end of the list, but when you take a supermarket trolley you can't take one from the middle of a line. In the supermarket you can pick an aisle at random but once you're in it you end up going the whole length of the aisle to find one item. These are real-world examples of organising the storage and processing of things. Inside a computer we use exactly the same methods.

[1] Lists

We have come across arrays already. Isn't that a list you ask? Well err.. sort-of. An array is a number of slots for putting things in. Often, for reasons of efficiency, you fix the number of slots. This is fine for many situations but gets very messy when you want to insert and delete from the middle. The next item in the list is always 'one more in the index'. This is a great advantage if your data fits nicely into this pattern because all you have to do is start at the first item and keep adding one to the index until you reach the end. But when writing a shopping list you're always thinking of things almost at random which don't want to go in the list at random. 'Take the library books back' doesn't want to go between cornflakes and crisps.

Each item in a list keeps a note of its neighbours in the list. The first item points to the second, second to third and so on. So how does this help - surely it's just a line at the end of the day? So it is, but if we want to delete say the 6th item this is how it happens in an array:

```
move item 7 into place 6, move item 8 into place 7 and so on  .... move
```

item 333 into place 332, make a note that there are only 332 live items in the array.

For a list all we have to do is

set the pointer of item 5 to point to where item 6 pointed (which in effect bypasses 6[64])

To insert a new item into an array means shuffling all the later items up one. To add an item to a list we have to break the chain of pointers at the right place. Suppose we're adding after item 4:

Copy where item 4 points to to the new item (Item '4.5' now points to 5). Set item 4 to point to the new item. (The chain is now: 3 points to 4, 4 points to 4.5, 4.5 points to 5.)

Lists come in various flavours and are used in various ways which we'll discover shortly.

[2] Lists of lists

Your shopping list might start with places to visit as headings. Library, bank, butcher, dry cleaners, supermarket and so on. Each of these may have its own list of things. So each of the top-level list items has a *pointer* to the next top-level list item (as just described) and also a *pointer* to the head of a list of 'child' items.

- Lists of lists are extremely common and useful
- They can be lists of lists of lists of lists ...
- Items at the same level are often called 'siblings'
- Items at the next lower level are generally called 'children'. (And vice versa 'parent')
- All the list-goodness applies to lists of lists, plus two very useful characteristics of the hierarchical structure:
  - You can work your way down the levels of a list getting 'more detailed at each stage' (and vice versa, ignore details when you want to work at a high level.)
  - Every child has a parent which gives it context.

Programmers are always thinking hierarchically. See [6].

I've been a bit vague about these pointers and how they actually 'point'. Don't worry the nitty gritty details will eventually be covered.

[3] Queue

A queue is a list which has new items added at the 'back' and always gives up the one on the 'front'. Here is some code for you to grovel through. It implements a queue using a *circular buffer*. This means when we get to the end of the space allocated for storing items we start again at the front. The really important part of this is that we don't actually move the items just the pointers for the head and tail. Each time we take an item of the head we increment the head pointer so it points at the new head item. (This code is ready-to-run if you want to try it out.)

```
<html>
<head>
  <script LANGUAGE="JavaScript">
    // ******************
    // ***** Heading *****
    // ******************
```

64 How objects are actually created, destroyed and allocated space is a subject for a later chapter.

```
document.write("<h2>Queue illustration</h2>")
document.write("<hr>")

/*
  1. We need somewhere to store the items.  An array
     is ideal.  WCPGW?  Too many items.  As it stands
     this program doesn't handle this problem
  2. Real world objects like people waiting at a
     checkout move.  We can use the magic of pointers
     to keep track of head and tail so we DON'T EVER
     MOVE the items in the array
  3. We need a way to tell the queue is empty
  4. When the pointers reach the end of the array they
     'wrap' back to the zeroth position ( "% 20" does that)
  5. DisplayQueue is an extra feature for illustration.
     (The funny for loop works by sending the index i
     up and round the loop until it reaches the end, but
     the 'have we reached the end' test is in the body of
     the loop not the head.)
*/


// ----ititialise----
queue = new Array(20)
frontItemPointer = -1   // flag for not-valid
backItemPointer = -1    // flag for not-valid



// ----functions for manipulating queue----
function AddItem(aString){
  if (backItemPointer == -1) {
    frontItemPointer = 0
    backItemPointer = 0
  }else{
    backItemPointer = (backItemPointer+1) % 20   // 20->0
  }
  queue[backItemPointer] = aString
  document.write("<b>"+aString + "</b> added at back<br>")
}

function GetItem(){
  if ( frontItemPointer == -1 ){
    s = ""
  }else{
    s = queue[frontItemPointer]
    if ( frontItemPointer == backItemPointer ){
      frontItemPointer = -1
      backItemPointer = -1
    }else{
      frontItemPointer = (frontItemPointer + 1 ) % 20 // 20->0
    }
  }
  document.write("<b>"+s + "</b> taken from the front<br>");
  return s
}

function DisplayQueue(){
  if (frontItemPointer == -1){
    document.write("Queue is empty")
  }else{
```

```
        for(i = frontItemPointer;;i = (i + 1) % 20){
          document.write(" ["+i+"]=" + queue[i])
          if (i == backItemPointer){break}
        }
      }
      document.write("<hr>")
    }


    // ---- exercise the functions ----
    DisplayQueue();
    AddItem("one");
    AddItem("two");
    AddItem("three");
    DisplayQueue();
    s = GetItem()
    DisplayQueue();
    AddItem("four");
    DisplayQueue();
    s = GetItem()
    s = GetItem()
    DisplayQueue();
    AddItem("five");
    AddItem("six");
    DisplayQueue();
    for (i=1;i<10;i++){
      AddItem("I"+i)
      GetItem();
    }
    DisplayQueue();
    for (i=10;i<17;i++){
      AddItem("I"+i)
      GetItem();
    }
    DisplayQueue();

    // ****************************************
    // ***** flag end of program reached OK *****
    // ****************************************
    today = new Date()
    document.write("<p><font color=green><small><i>[")
    document.write(today.toLocaleString()+"]</i></small></font>")
  </script>


</head>
<body>
<!-- all the work goes on in the javascript in the HEAD -->
</body>
</html>
```

- What are the options for handling the 'quart into pint pot' WCPGW? Use a list of course! Don't try this at home - *yet*. I'm keeping some magic back until you're ready to handle it. So stay in the nest a little bit longer.[65]

---

65 The magic is objects and automatic storage allocation. Bear with me. Code Bunnies can't fly.

[4] Random access storage

You can put you car in any space you like in the car park.

- If there are any spaces.
- There may be some reserved for special uses

You may have to hunt for a space. When there are only a few spaces left it is more difficult to find one you prefer.

Computer memory, both the working RAM and hard disc are organised so you can park things pretty much where you like. Fortunately the details are handled for you by the electronic version of a parking attendant who does the actual parking and fetching for you. Some languages give you more power over the parking attendant than others.

Memory allocation will often be done completely transparently, all you have to worry about is not hogging memory unnecessarily. Many languages clear up after you, so freeing the memory for the next job. This is called *garbage collection*. Failing to free memory allocated to your program results in a *memory leak.* After each run of the program (or function) the computer has less memory free until ...

See [12] for one way of finding your car again.

[5] Stacks

Consider the pile of baskets at the door. You can only take the top one or return an empty on the top. This is called a *stack* and is extremely significant in computing. (A stack is pretty much the same as a queue except adding and removing take place at the same end.)

First-in-first-out : FIFO : A queue Last-in-first-out : LIFO : A stack

You use a stack when you want to defer something you're dealing with while you work on something else. Suppose you are working on coding a cool program when the phone rings. You halt coding and answer the phone. Then the postman rings the door bell so you ask whoever phoned you to wait a moment and answer the door. Then return to the phone call. Then when the phone call is finished return to coding. As each task is interrupted it gets put on the stack (*pushed* is the term used in computing) then when an interrupting task finishes the stack is *popped* to return to whatever it was that was interrupted. Does 'return' ring any bells? I should hope so.

You will probably come across stacks when doing involuntary debugging. The interpreter might say something like

```
divide by zero in foo()
exception in bar()
exception in baz()
program halted in main
```

This is telling you that the main program called function baz() which called function bar() which called foo() where the actual problem is. Let's see how a computer might process the following code:

```
if ( FunctionFoo() > FunctionBar() ) { ...
```

Let's work on the IF condition. The first part is a function which I need to evaluate before I can continue. Make a note of where I've got to so far ON THE STACK and jump to the start of FunctionFoo(). Do the work in FunctionFoo()... Aha Finished. Now where was I? Oh yes I made a note of that on the stack. Next is greater than. Now another function. Make a note of where I've got to on the stack. Jump to the start of FunctionBar()....Finished FunctionBar() so pop the stack to get back to trying to evaluate the condition....

Do you remember in chapter 5 when looking at digital roots we had a function that

called itself. No problem... unless it never returns. The *call stack* would get larger and larger until it *overflowed*.

[6] Keyed access - dictionaries

Time after time we want to find something by name. The general form is:

```
key ---> value
```

- Key is often a string.
- Value can be anything.
- Physical storage can be anything from an array to a database

The two core operations are

- *Add this value and label it with this key*
- *Fetch the value that matches this key*

In the supermarket a programmer might think of

`/FruitAndVeg/Apples/GrannySmith` or `FruitAndVeg.Apples.GrannySmith`

Hierarchical keys are common in everyday programming. 'Inside' many dictionary type programs there will be a hierarchical structure, perhaps like your address book that has one page for the people beginning with A and then all the A-names listed. We are firmly in list-of-lists territory here. A computer might have the top level list for the first character of the name, then lots of second level lists for the second character of the name. This is a good thing because instead of having to plod through 100s of names until a match is found the all you need to do is plod through 27 letters then you have 1/27th of the size of list to slog through.[66] All of this normally goes on behind the scenes and you probably don't need to worry about heaps, binary-trees and similar exotic, but extremely important in their own way, data structures.

[7] and [8] Random -v- serial access

Supermarket aisles tend to have big signs over them saying what sorts of things you can find along them. Aha! dictionary look-up which we've just discussed. But that just gets you to the end of the aisle. Now you have to plod all the way down it until you reach the thing you want. Sound familiar? That's what we were just discussing in [6]. Should there be lots of very short aisles or a few long ones to save the cross connection space? It's a matter of compromise. Finding the best compromise is called *optimisation*. Optimisation depends on the relative value of your resources. You might be desperate to have something that works very fast, or using hardly any memory. It won't come as a surprise that a lot of work has gone into the 'best' ways of doing things.

[9] More queues

Fortunately there are normally multiple checkouts working in parallel. The same thing can be implemented in computing. For example when you call a web page from a large web site your request will get passed to one of a number of web servers to be processed.

[10] and [11] In the bag!

Sometimes it is convenient to stuff things into a bag without any structure. Just bung it in until it fills up then start another. Storage often comes in physical or administrative units which you can do with what you like.

- Too many small bags are a nuisance
- Too much in one bag makes it difficult to find things. (This is the ladies handbag

How do you structure your directories? Neither one big one or millions of almost empty ones.

66 Actually it is more subtle than this - Some of the details will be revealed in the Algorithms chapter@@@appendix

problem. There is so much in there with so many pockets that nothing can be found without great palava.)

[12] Hashing

When you left your car in the car park you didn't make a note "15th space in the 4th row". Instead you remembered roughly where it was. When you come to find it agin you head for the approximate area, then you should be able to recognise it immediately. This is called ***hashing***. There might be a slight problem if there's another car like yours in the same vicinity but with a bit of investigation that can be sorted out. It is a very practical method that's stood the test of time. The same goes for the computer version, only in this case the hash program doesn't put the item to be stored just anywhere, but at some place derived from the item itself. It's as if a car park with ten rows has a system whereby the first digit of the registration number is used to pick the row and the a rainbow colour system used for how far along the row, red at the near end and blue at the far end. Of course this works for both putting and fetching. Although hashing is used a lot behind the scenes an everyday programmer is unlikely to use it much - but when it is the right tool for the job it will knock the socks off other methods.

There are some traps with hashing, but as it is such a cool technique perhaps we should do some pseudo code. Suppose I have a thousands of records of people who I know their names and dates of birth.[67] One way of keeping my records would be to have a sorted list of names, but each time I need a name I have to work my way through the list of names until I get to the right one. This might mean a dozen tests even with sorted lists of lists.[68] Here's how we could hash. Let's take the digital root of the date of birth to give us 1 to 9 and add it to ten times the last letter of the name (Where value is here calculated as A=1 B=2 etc). So `Mary Carter d.o.b. 15/3/47` has the hash (R2),182, `Sam Holland d.o.b. 7/7/50` has the hash (D1),41. The largest hash I can get with this system is 269 and the smallest 11. To store up to 250 people I could use the following method.

```
Allocate an array with 280 vacant slots

To add somebody:
   Calculate the hash
   Until an empty slot is reached
      Add 1 to the slot counter
   Put into the place we've reached

To locate somebody:
   Calculate the hash
   Until an empty slot is reached
      Test to see if the occupant of the slot is the one we want
      If yes then finish
   Finding an empty slot means the person isn't in the list
```

Ah yes I, ahem, forgot to mention ***hash collisions***. What happens when now we come to add `Tom Raymond d.o.b. 23/6/62` which hashes to (D1) 41? This is the same hash as Sam Holland. This isn't a fatal flaw because when we dive into the array to add Tom and find Sam already sitting there at position 41 we just try a little bit further up the

---

67 Such as in a genealogy program

68 Small lists and lots of levels is an efficient search tree and would be practical here.

array. With any luck we'll only have to try a few until we find a space. The same applies to fetching. The great thing about hashing is that it is very fast. Do a quick sum on the key and you go to either exactly the right place or very nearly the right place.

- Notice that this system can't give hashes less than 11 or any hash ending in 0. Do I care about this 'waste of space', shouldn't I calculate the hash as *nine* times the letter plus the digital root? No not really. Hashing depends on there being spaces so these might come in handy.
- Can you guess what we do when looking further up the array and 'fall off the end'? You've seen it with the queue example we looked at above - ***wrap*** to the bottom and carry on.

WCPGW?

- Not enough spaces. This results in an item being stored many slots up the array from the initial hash point.
- Uneven hashing. In our example there won't be many in the ten 'Z' slots from 260 to 269 but there are lots of names ending in R so the 180s will overflow into the 190s and possibly beyond. (That doesn't stop S and T being used it just means they have more competition.)
- What happens if somebody is deleted? A blank slot is left. That's fine for adding another but can break the search procedure which stops at the first blank. So we need to be a bit cleverer.

```
BIT_ROT_AND_EVEN_HASHING
- Bit Rot is a mythical process where old programs don't seem
- to work any more. Purpose 1 of this exercise is to check that
- BR hasn't set in to your digital root program.
- The second purpose is to see if the digital root of dates has an even
- distribution. You've done a simulation using dice so this should be a cinch.
0. Reusing code from digital root and dice programs
1. Confirm the digital root function works
2. Either
2.1     With all dates
2.1.1       ie for y=00 to 99, for m = 1 to 12, for d=1 to 30/31
- NB Write a 'how many days in month' function and put it safe
- You'll need it hundreds of times in the future
2.2     Or with random dates
2.2.1       ie y=random()*100, m= random()*12 etc
-  NB no zeroes in months or days!
3. Count the frequencies of digital roots 1..9
4. Report
```

Final notes on hashing

- There are many ways to skin-this cat, for example you might hash your way to the head of a list then work through the list.
- Hashing is no good whatsoever for sorting items or doing something like "I want the people born in 1953" You need to know what you're looking for and the ***hash table***

will tell you whether it is in there or not.
- The efficiency of hashing comes from being spot-on first time or at least very close which means an even distribution of hashes and spaces.

[13] Containers

When you put your bags of shopping into the boot of the car you are nesting containers. Other things might go in the boot, but perhaps not an elephant(size limitation) or a bucket of water without the bucket (unsuitable container.) Somewhere, somehow information gets put into containers of a logical type (arrays for example) which are eventually stored in physical containers (disc drives, RAM chips) for example.

In the 80's and 90's there were enormous strides taken to tackle the size and suitability issues. Chopping information up and later reassembling it is now done as a matter of routine. In Ye Olden Days any data larger than 360K bytes had to be split using a proprietary program across floppy discs[69]. Nowadays data is split into *packets* for sending over networks.

> Enlightenment: Why is data sent over networks split into packets rather than being sent in a continuos stream? Because the streams would mingle and you'd never be able to untangle them.

There is another aspect to suitability: Privacy and security. As a Real Programmer you'll need to study the techniques and be a champion of stopping up all the leaks. Only a suit would gaily put 196,000 real personnel records onto a laptop and only realise after the event that it could get stolen.[70] Security will be discussed in detail in a later chapter. You lock the boot of your car don't you?

[14] Tell me what you are little thing

When Mummy's Little Helpers come to help unpacking the shopping they need to recognise that a cauliflower doesn't get put away with the cornflakes and the milk goes in one place and the bleach in another. This task would be impossible if there were no labels on the tins and packets. Sometimes the item will say "store me in the freezer". Other items rely on them being what they are and you knowing that say grapes go in the fruit bowl.

You could go on a computer programming course and never have anybody tell you that just like all real things are made of atoms but when joined together in a certain way become a mushroom or a walrus, so bits (in the 0's and 1s sense) are joined together in different ways to make different information things and we need to handle the 'thing' as a 'thing' not as a collection of atoms or bits. Trying to wordprocess a video file or spellcheck a photograph is pointless and it won't work.

This is why files tend to tell us what sort of file they are by their file extension. Not only do they tell us but they tell the programs that work with them as well. That is only one example of how a bit of information is able to tell us what it is. Many things contain a

---

69 As recently as 1994 I was working for a large financial firm who couldn't simply transfer data from mainframe to LAN except in a batch operation using optical discs. Because of suit-isms one particular job took ten days instead of 20 hours.

70 Sadly a true (but far from rare) case. Search for Fidelity and HP.

signature: HTML files start with `<HTML>`[71] All my word processing files start with `FF 57 50 43`[72]

As we'll find out in the next chapter some can say "best before foo" and "here is how to cook me".

Review

What we've covered in this chapter is an introduction to the ways of organising data. It is more important that you understand the various possibilities and their strengths rather than trying to program these yourself. Nowadays the majority of programming will pass by most of this most of the time... except that efficient data storage and retrieval is fundamental to good programming. Whenever you have a bunch of things to manage you have to chose a system... but so many tools are ready and waiting for you to use that you normally don't try to re-invent the wheel - just fit the right wheel to the right vehicle.

Physical data storage

As you know data is stored using different physical media. Also there are virtual ways of storing data such as in a cache, 'somewhere' on a network, in a database. There are two things a programmer needs to be aware of when dealing with various forms of storage: 'Cost' of accessing it and accessibility.

Cost

By cost we don't usually mean money, but time and memory. For example if you wanted to sort a pack of cards you could do it very quickly by laying them out in a pattern on a large table. If you didn't have a table you'd need to juggle back and forth between two handfuls and a knee. The really slow way would be to fetch two cards from a box, compare them, and stick them back with a marker, then take another two out, look at them and so on. The slow way to sort files in a filing cabinet is to sort the files themselves, the fast way is to go through the files once making a list of the names then sort the list of names to make an index. Similar considerations apply to electronic data. When we look at algorithms we'll see there are some methods that get a result faster or with less shuffling of data than others. Although using an inefficient method can be *very* inefficient, (orders of magnitude) on the top of every programmer's agenda is how long does it take to get at a bit of data. Because having a look on a hard disc takes so much longer than having a look in random access memory a programmer will be careful to access data on hard disc in the most efficient manner possible. *Cacheing* is one way. At the supermarket you go and fetch all the items, *then* go to get them checked-out. Imagine the situation where you went to a till with one item, had that checked-out, then wandered off round the shelves for the next item, had that checked-out, then went and found the next item. This isn't just inefficient but so seriously inefficient that it makes the supermarket concept unworkable. The same applies to taking items out to the car - you wouldn't do that one item at a time. Neither would you access data over a network like that.

This difference between a practical system and non-starter is quite typical of some

---

[71] Actually that's a lie - Have a look at some source.

[72] That's 4 bytes in hexadecimal. It's the signature of a Word Perfect file.

applications. As a programmer you will always be thinking of the efficiency even if *most* of the time it is a red herring due to huge memories and fast processors. But note that in server applications where there may be dozens of copies of your program running together you need to keep a strict watch the possibility of running out of time and memory resources.

Accessibility

If you want to find the ace of diamonds in a pack of cards that is really easy if they are laid out in front of you. (Cost=1). If they are in a deck you have to work your way down looking at each one in turn (Cost=26 on average). The first method is ***random access*** the second ***serial***. In Ye Olden Days all data was stored on magnetic tape. Imagine taking some items out of stock using a stock system: Read through tape to get to right spot - see if any/enough in stock - rewind a bit - rewrite new stock amount - rewind a bit - check the tape has recorded the correct information. [73] That's why the processing was saved up until the evening when the ins and outs would be sorted into the order they were on the main stock tape so it could run through in sequence. That's the explanation for why the computer would say "there's 10 in stock" but the shelf was empty.[74]

Nowadays when data is on a disc we can often jump to any spot we like to save this palava.[75] Data flowing across a network, or between cooperating programs is a different matter. Not only can we not even 'rewind' but we generally have no control of how long and possibly in what order the data is going to arrive! When your web browser asks for a page from a web server it hopes, eventually, to get the whole of the HTML document which it looks at and then asks for all the other bits and bobs that it needs to fill out the page such as style sheets and images. It will send a host of requests for these items off to the server then wait with no knowledge of which will come next or how long it will take. If all you wanted was the 1st image you've got to ask for the whole page and may have to wait for the whole thing to download.

Files

You know what a file is. Basically a long stream of bytes written to a disc. Those bytes might be organised to represent a picture, bank account, error log, personal preferences, seat reservations or anything.

Although behind the scenes the computer is probably treating all files in the same way, there are two practical ways for accessing files - our friends *serial* (called ***sequential***) and *random access*. The main advantage of reading or writing (By the way normally only one of these at a time) to a sequential file is that you don't need to keep track of where you are. If you like "you are where you are". Often sequential files are arranged as text, one line at a time.[76] Suppose you want to write an error message to a log: All you have to do is say `stick this new line onto end of error log file.` To do this with a random access file you'd need to know how to find the right place to put it and make sure the data you were writing wasn't too big for the ***record***.

---

73 Look up ISAM - Indexed Sequential Access Method for more about how the 'right spot' was determined.

74 You'd have thought this was history now...

75 And the data is often cached transparently for us by the operating system.

76 See glossary entry on New Line for details and traps

Record?

If you want you can access individual bytes of a random access file. It is much more usual though to store groups of bytes. Such groups are called records. You say something like *fetch a stock record from position 123456*. You might have defined a stock record as say An unsigned 32-bit integer for an ID, then a 16-bit integer for the quantity in stock, then 30 characters for description then four bytes for single precision real for the unit price. 4+2+30+4 = a block of 40 bytes. All records have this *fixed length* of 40 bytes.

If all you had in this file were stock records then the first would start at position 0 (*offset* is the usual term for position here), the second would be found at offset 40 and the n^{th} at position 40 * (n-1). If you had an array in memory of which stock ID was which 'n' you could access the required record in one hit. Or perhaps you hash the stock ID to get 'n' thinking that should be near enough.[77]

Mixing records

In a sequential file you will often get lots of different records joined together. The way this works is by indicating at the 'head' of a record what sort of thing it is and how long it is[78]. The 'how long am I' bit can then be used to tell where the next record starts by the simple expedient of adding it to the head position (ie offset) of the current one. Obviously to get to the 77^{th} record you have to start at the top and do 77 of these skips. This may seem a drawback but in practice, for the sorts of thing these files are used for it isn't.[79]

Random access files are often made up of various sections and sub-sections. Somewhere at the top might be a table of offsets to the various parts. (You can only say what these offsets are when you've written those sections.)

The most general type of random file access requires an index. This is a compact set of records which match the key being wanted with the offset. Then there is no need to stick to fixed lengths to do the how-do-I-find-it sums.

Abstracting I/O

It is worth noting that you may not be dealing with real files as just described at all. One way to 'cheat' is to use a database program which keeps track of everything for you. *I/O* is the all-encompassing term 'input/output' which refers to any data flow.

> In Ye Olden Days I/O would include as a matter of course keys pressed and characters typed on a teletype and later VDU console. These would be sent down a wire character by character. Nowadays the console might be built-in to the operating system. Look up console¤

---

77 Which it might well be. When the operating system reads a bit of a file it will probably end up caching much more than you access. So the next 'read of the file' is really a 'look in RAM'.

78 The How-long-am-I is often missing if it isn't needed but it is useful for picturing the structure.

79 'Serialising' comes later when you know about objects.

Streams

A very important concept is a *stream*. In simple terms, this presents the programmer with a serial source or sink for bytes to be read from or written to regardless of whatever the actual storage being used is. This applies to networks, discs, and communications between cooperating programs.

This is important because a great deal of input and output (and internal processing) can be made to look like a stream. This standard interface makes it easy to isolate the program from that strange and uncertain place: - The rest of the computer, the rest of the network and the real world.[80]

- FileStream? What's that? It's a Stream. A Stream is a 'pipe of bytes'. A Stream is a very common abstraction used to interface with 'data storage'

@@@ More on streams??? (elsewhere?)

Conclusion

If you thought computing was all about number crunching then you might still be right (for certain applications) but now you know that these numbers are organised in certain ways to make manipulating them efficient. Next time you go to the supermarket stop to think what would happen if say trollies didn't *stack* inside each other, how much space would that take? What would happen if there were no labels on the tins? What would happen if the shelves were arranged in one long aisle and you were picking your list in the order on your paper?[81] And how many mistakes would you make if you had to remember precisely where you parked your car. Just think : Supermarkets would be impossible without the shopping trolley.

You'll need to spend some time understanding the limitations of physical storage as well as the logical aspects in order to be a competent programmer. For most people this means a little bit of experience and a lot of vigilance - knowing where major 'costs' can occur should put you on the alert for gross inefficiencies. (Ignore the minor ones.)

I've been rather coy about data, bytes, records and silent on objects. Stand by for more eye-opening stuff as we look at ...

---

80 Real world as in keyboard and screen.

81 If you think that's ridiculous, consider how a fork-lift operator in a large warehouse picks pallets from many long aisles, stacked 10 shelves high. One of the first uses of business computers was to convert a 'mixed lorry load' into a set of instructions for a fork-lift driver to fetch items in as short a journey as possible.

7. Data gets intelligent

Introduction

In the previous chapter I was a bit vague, referring to 'things' and using tins of beans as a metaphor with the clear implication that 'thing' could mean anything. In this chapter we'll nail down these 'things' in a computer context - and see that they can still mean anything. We'll be talking about *objects*.

You may have heard the term 'object oriented programming' (which everyone shortens to *OOP* or just OO) and you may know somebody who can quote the three holy tenets of OO. :Persistence. Inheritance. Polymorphism. That's all very good stuff - but I want you to *think* 'objects' not spout the mantra.

Before proceeding, I expect some of you are asking "what about databases and web pages and client-server and bots and frameworks and image manipulation and file formats and applets and games" and so on. Don't worry we will get to 'useful' and 'commercial' all in good time. This chapter will be as fundamental to your programming as perspective is to a painter: It is obvious, there all the time, and there are ways to make the illusion work seamlessly.

Tiny objects

The smallest computer object we can have is a bit: 0 or 1 in binary, a high or low voltage, the presence or absence of an electrical current, a tick or a cross, a left or right and so on. That single one-of-two bit can be

- indicated in many ways and
- interpreted in many ways.

> OO experts will be going "pshaw!" and starting to foam at the mouth here. Well thinking of all things as objects is the name of the game. Later it will emerge that some objects are whizzier than others.

Most programmers will be insulated from the need to worry about how bits are indicated (but don't relax, indication will become very important shortly) and it is the interpretation by a programming language which 'knows' how to interpret this as True and False which we then use in our program for pass/fail, up/down and so on.

The next smallest object in common use is the *byte* or *character*[82]. A byte is a group of 8 bits. Bytes are the currency of computing. Data is banked on your disc in bytes (typically 512 at a time), jingled in RAM (often 4 at a time), exchanged over the Internet (in any amounts), stolen, minted, defaced and lost behind the sofa.[83] For the record there are 256 possible combinations of 8 bits which are normally interpreted as 0...255

---

82 Pshaw! Quite right: A character is not a byte as will be explained in a couple of paragraphs time.

83 A lot of information seems to evaporate. Some people blame Gremlins, but in out heart of hearts we know we didn't look after it carefully enough.

(or 0...FF in hexadecimal, or 0...ff case doesn't matter with hex. Sometimes you will see hex numbers flagged as being hex by a preceding '0x'. so 0x10 would be 160 decimal.) Of the 8 bits one, called the least-significant-bit, will indicate 1's, the next 2's, the next 4's and so on to the MSB (Most significant bit) which indicates 128's. Sometimes there are variations on this theme which will be pointed out to you in any documentation. It might say "flipping bit 0 will alter the read-only flag". That's the LSB you'll have to switch from a 0 to a 1 or 1 to 0 while leaving all the other bits alone! There is an appendix on this subject.@@@

There is a standard way to interpret a number between 0 and 127 as a character of text. This is ***ASCII***.[84] 127 can fit into 7 bits, with the 8th originally used for parity. (See the glossary for more on parity and its legacy.) As well as the alphabet in upper and lower case, numerals and some common symbols such as the ones we've seen in code, characters 0 to 31 were given special meanings mostly for controlling communications. These are called ***control codes***. The ASCII control characters you need to know are:

Name		Dec	Hex	Escaped
NUL	Null	0	0	
Tab		9	9	\t
CR	Carriage return	13	0d	\r
LF	Line feed	10	0a	\n
ESC	Escape	27	1b	

- NUL is extremely important. A lot of strings, sometimes called Unix-style, indicate their end by a NUL stuck on after the last 'real' character. This is no problem so long as you remember that when a string says it is 6 characters long it actually needs 7 bytes to store it.
- When reading documentation you may also see CRLF which *probably* means CR followed by LF.[85]
- There is no such character as EOL/End of line/***newline***. This very important marker, which is used quite a lot, might be CR, or LF or CRLF.
- Space is 32 decimal or 0x20
- ***Whitespace*** refers to any number of consecutive space, tab, CR and LF (and normally all control) characters. Whitespace is often important when parsing[86] data.
- The Escaped column shows common shorthand used in many programming languages to allow you to indicate these control characters easily. For example

```
print("Tab\t\t9\t9\t\\t");
```

should give you the relevant line from the above table. ("\" in this context is known as the ***escape character***. How then do you get a backslash? Normally by doubling it.)

---

84 Nobody cares what the letters in this acronym stand for. This may be a surprise to some of you but having a universal standard for which number meant A (65 decimal by the way) and so on was such a good idea that IBM kept on using its own, different version. Now history repeats itself with Microsoft trying to promote its own err.. 'standards' for Open Document Format.

85 In UNIX-world a end of line is traditionally indicated by LF. Elsewhere by CR and LF. This can cause confusion, particularly if you get a standard routine to do reading and writing of lines that get swapped with other systems.

86 Parsing means splitting up into tokens(mostly words). Whitespace is frequently used as one of the splitting methods. Thus fireSPACEalarm reads the same as fireTABalarm and fireSPACESCRLFPACETABSPACEalarm.

- The ESC control code is one of those communications symbols. Nowadays you are unlikely to come across it, but in Ye Olden Days we'd send things like `NormalESC(s1SItalicESC(s0SNormal` (where `ESC` is the single escape character) to get print "Normal*Italic*Normal" on the printer.[87]

More characters

Having only 96 usable characters is a bit of a limitation. The 8th bit was pressed into use in various ways, sometimes to draw lines to make boxes to line forms on character-based screens and sometimes for accented and novelty characters such as ©,È and è not to mention £ and ¢. This is fine for many English-centric uses but is a bit hopeless if you want Russians or Japanese to be able to use your program in their own language. So the only way out is to use more than one byte for a character. Now you see why 'character' usually means 'byte interpreted as a bit of text' but sometimes means '*one or more* bytes interpreted as a bit of text'. More modern programming languages have support for *Unicode*, older ones don't. Unicode can represent anything including weird mathematical symbols.

Larger built-in data items

We have already discussed types (Chapter 2?@@@) and discovered different types needed different numbers of bytes to contain the information. I could send you a file of 100 32-bit integers using exactly 400 bytes. (Or of course 100 bytes if I knew all my numbers were small enough and I was mean about file size.) What happens when you read the first number: Read first four bytes as a number is probably built-in to your language. With any luck my program with its WriteANumberToFile() function uses the same convention for physically writing bits and bytes to the file as yours does for reading. But it may not be the case. There is a standard @@@ but it's not much good stamping your little feet if the data source isn't using it or your programming language doesn't support it.

Real numbers work in the same way. If you know that the next n bytes are a binary representation of a certain sort of floating point number just call the appropriate built-in function.

Strings come in three varieties:

- Fixed length where somebody has specified there will always be n characters assigned to a particular string
- Null-terminated of variable length which you have to read a byte at a time until you reach the NUL (zero)
- Specified length where one or two bytes at the front tell you how long the string is and therefore how many bytes to read.

---

87 I used to be able to read most HP printer escape codes. A ten character string of a bit of this and tweak that and adjust the other was normal! For this book I tried to work out how I'd do a superscript but it was so horribly complex I gave up. Be safe, don't waste your life like me boys and girls - stick to ready made printer drivers.

The really dumb amongst you can try mixing these types and see what happens.

> ***Hex Editor***
> How do you find out what the byte and character values are within a file? Use a Hex Editor. This will normally list 16 bytes per row both in hex and also as a character. I can recommend you looks at some binary files (don't change anything!) to get a feel for how this tool works. Later you can use it to see what happens when you write something to files of your own.

Lines

There is a really easy way to deal with textual data within sequential (often referred to as *text*) files. Write one thing (or set of things) per line. This format is used a lot by *configuration files* and there is no reason why for non-binary data you shouldn't use it yourself if you want a format that is easy for other programs or humans to read. Most (but by no means all) programming languages have a function that will read a whole line (ie up to the next newline[88]). This is a really great format for logs and reports, configuration files, some data files, and not least program source files. Actually a lot more file formats such as email and HTML use it as well. See if your chosen programming language has readln() and writeln() functions for starters. If not there may be other ways of doing the same thing.

- Lines of text can only contain err... text. (No control characters anyway)
- Lines of text can be of any length.[89]
- In all the text files I've encountered one byte=one character.

Here is an example from one of my files

```
#      Background bitmaps
# ============================================================
# This file lists scans from maps with background images
# Field 1  : Level      Smallest number gets drawn first.
#                       Larger numbers will paint over smaller ones
# Field 2  : Filename   BMP file
# Field 3  : Map ref    South West corner
# Field 4  : Map ref    North East corner
#            The map refs are formatted as grid square followed
#            by two Km values eg TL 56 70  or TR 5.8 77.2
#            NB Do not use commas to split these three parts up
#               ---------------------------------------------
#  Example   10,BackGnd.bmp,TL 20 20,TL 40 40
10,     bg001.bmp,     TL 70 00,       TL 90 20
50,     bg002.bmp,     TL 71 15,       TL 75 20
50,     terling.bmp,   TL 76 14,       TL 80 18
40,     cressing.bmp,  TL 76 19,       TL 80 23
50,     ayroding.bmp,  TL 57 14,       TL 61 18
50,     sailing.bmp,   TL 70 23,       TL 74 27
51,     send.bmp,      TL 80 18,       TL 82 21
60,     cretem.bmp,    TL 78 17,       TL 80 20
```

---

88 Just to remind you (I know it was only a page ago) that newline is a concept that may be physically represented in up to three different ways. Every programmer worth their salt will check that their built-in readln() actually works with the data they're given as the first thing they do with the data.

89 Somebody will tell me about a language where strings are limited to 254 characters or some similar limitation. In which case I'm wrong - But if you're writing such long lines you've probably got the wrong file format.

- # is one of the common conventions for a comment
- There isn't a bit of binary in this file. The program that reads this (It is written by a human - so that's a good reason for avoiding binary) has to read character 1 then character 0 and turn that into 10. That's normally very easy.
- A bunch of data items is called a *record*.
- The parts of a record are called *fields*.
- The way to load this data into a program is Read line. Split into fields using comma as a marker. Try to interpret each of the four strings in whatever way is appropriate. If all fields could be interpreted OK then report thumbs-up!

Fields and records are key concepts. A record is a self-contained, pre-defined collection of fields. Each field is a data item. (And to make it a little more confusing) A field could be a record in it's own right. In fact the example given illustrates this where fields three and four, which are map references, are comprised of three elements grid square, Easting and northing with whitespace as a *field delimiter* or separator.

You will often come across configuration files where each data record takes the form `foo=bar`. In this case there are two fields delimited by the `=`. (Normally the first is referred to as the key and the second the value.)

Binary records

If my computer is talking to yours directly, or you're storing data for future use then all this conversion from text to binary and vice versa is a bit of a bore. No problem, just specify a bunch of fields in a group and treat them as one unit. Here is Delphi code that illustrates how map references might be packaged.

```
MapRef = record packed
  Square : string[2];  //  3 bytes (2data+1length)
  East   : single;     //  4 bytes
  North  : single;     //  4 bytes
end;                   // 11 total
```

Suppose I wanted to emulate the previous text example in binary I can nest my MapRef type inside a BackgroundBitmapSpec as follows. (Recall how in the previous chapter we had items in containers inside other containers - tins of beans in bags in the boot - here is the electronic version.)

```
BackgroundBitmapSpec = record packed
  Level : byte;              // 1 byte
  BitmapFile : string[60];   // 61 bytes
  TopLeft : MapRef;          // 11 bytes
  BotRight : MapRef;         // 11 bytes
end;                         // 84 total
```

- Delphi aligns fields on even bytes unless you explicitly tell it to squash them together. If `packed` was left off then `Level` would occupy 1 byte plus a spacer to bring the start of `BitmapFile` to an even byte. Why should we bothered? Because if we are exchanging data and we publish the specification of which bytes in a record represent which parts of the record we could forget these spacers.
- Fixed length strings alert!
- A typical method to indicate 'part of' or 'field' is to use a period. So the Square field of the BotRight field of a variable called bbs would be
  `bbs.BotRight.Square`
- Records are really complex custom built data types and are used in the same way.

```
var
  bitmapSpec : array[0..6] of BackgroundBitmapSpec; // dimension
. . .
```

```
for i := 0 to 6 do begin
  read(myDataFile,bitmapSpec[i]);  // read next 84 bytes
  if not fileexists(bitmapSpec[i].BitmapFile) then ....(error)
end;
```

- This last bit of code assumes there are always exactly 7 records. Unlikely! A very common way to deal with this is to write the number of records then the records themselves. When reading the first thing yo do is discover how many records are following and allocate and loop accordingly.

In a short while we'll talk about types that know how to do things for themselves. (Objects). Two common function which may be built-in to your language are WriteMyselfToFile and MakeMyselfFromFile[90].

Review

So we can make our own types and have discovered how to make them *persistent*. Persistence means we can store them and recover them later. We can still use built-in types and text files of course but the future is *objects*. When working with binary files use a Hex Editor¤ to check that everything is in exactly the right place. File formats change without anyone telling you and your programming language may write in a slightly different way to that used by somebody else to read.

Objects

Problem: Many languages do not support objects. If in doubt look up 'constructor' in your documentation. If it doesn't appear then now might be a good time to look for one that does. A lot of programming can be coded without objects but unless you are fluent in 'object-think' you won't know what is and isn't suitable. On the other hand, Java is a mainstream example that forces you to use objects.

> Usually you get a special chapter on objects. The concept is extremely important and we'll be getting you immersed in them in subsequent chapters but the principles are simple enough to grasp quickly.

Recall the groceries being unpacked when you got it home from the supermarket. All the packets and tins had labels which told you things like weight, fat content, how to store, how to cook and so on.

Terminology note

So far we have been talking about *types* and *fields*. At some stage we've got to convert to the exact same things by other names: Types become "objects" and "classes" (Mostly interchangeable terms). Fields are still "fields" plus "properties" (which for now we'll say are the same as fields) and a new component "methods" which are functions specific to an object. Watch out as we segway¤ into this new tech-speak.

The code given below is pseudo-code. You won't see any actual code like this but you should be able to translate this into the object fields of your preferred language.

---

90 Not usually given these names. Also not necessarily a file - could be a stream.¤

Inheritance

We could define a type called TinOfBeans with fields Weight, Barcode, Nutrition, and Contents. We could also define a type called PacketOfPasta with fields Weight, Barcode, Nutrition, and Contents and a type called Pizza with fields Weight, Barcode, Nutrition, and Contents. What a bore! Instead we could define one type to cover all of them

```
GroceryItem
  .weight : single
  .barcode : string
  .nutrition : BlahBlahText
  .contents : string
```

So far so simple, we're just setting up a general type to cover all grocery items. What about tins and bottles. They say what material they are made of which isn't allowed for in GroceryItem. We could either add that to our general type and put up with it being irrelevant in many cases or, *this is the important bit,* develop a more specialised type from the general type. The specialised type will ***inherit*** all the general type's fields and add some more of its own.

```
Container inherits from GroceryItem
  .material : MaterialType
  .recycleInstructions : string
```

Container objects now have these two *additional* fields.

Similarly

```
FrozenFood inherits from GroceryItem
  .storageInstructions : BlahBlahText
  .howToCookFromFrozen : BlahBlahText
```

and

```
Tin inherits from Container
  .doesTopHaveRingPull : Boolean;
```

WCPGW? We've forgotten 'Best before date'. Watch this magic. All we have to do is add a `.bestBefore : Date` to `GroceryItem` and all the classes *and all the inheriting classes* will have it too.

Functions of objects are also inherited.

Constructors

Our `GroceryItem` is an abstract concept - a ***class*** of things not a particular ***instance***. To create an actual instance of an object we need to *construct* it. The function used to do this is called a ***constructor***. Once constructed we can set the properties.

```
tinOfBeans = new Tin()
tinOfBeans.material = Steel
tinOfBeans.bestBefore = "May 2006"
tinOfBeans.doesHaveRingPull = False
```

- One widely used naming convention is to use lowercase to start instance variable names and uppercase to start class names. Here we can tell `Tin` is a class and `tinOfBeans` is an instance at a glance.

There are two styles of naming constructors. Some languages give you a ready-made function name such as Create() to use, while others recognise the class name being used as a function. Here are examples

Java	`Tin tin = new Tin();`
Delphi	`tin := Tin.Create();`
PHP	`tin = new Tin();`

A constructor is a function and as you recall functions can take arguments. Let's define a constructor function for GroceryItem. Here is the sort of thing you might see:[91]

```
function GroceryItem(Grams,ID,Contents){  // constructor
  weight = Grams;
  barcode = ID;
  this.contents = Contents;
}
```

- My personal convention is to Capitalise function arguments.
- We don't normally need the leading dot (NB assuming dots are used - some have other symbols to indicate 'component of object') inside the function when referring to an object's fields.
- The last line untangles a potential confusion between 'contents' the objects field and 'Contents' the function argument. `this` is a common way of saying 'this instance of the object'. Sometimes `self` is used for the same thing.

Since a constructor is a function of an object[92] and functions are inherited we can use the GroceryItem constructor to create a Tin instance. In fact we may never want to create the skeleton GroceryItem at all, always ***sub-classing*** it to make useful objects. A class that never has instances per-se is called ***abstract***.

Lets see how this bit of code works

```
tin = new Tin(100,"123456789","Baked beans")
```

new tells me to construct a new instance of a class. It will be a Tin object. What's that? Do I have a constructor for it? No...But it may have inherited a constructor function from the parent class. Oh dear no constructor function for Container...But it may have inherited a constructor from its parent. Yes! Here it is and it wants three arguments (which I've just been given - that's OK) Set the weight field to 100, Set the barcode field to "123456789" and set the contents field to "Baked beans". Wrap everything up as a Tin object and finish.

Just as we added to the fields of GroceryItem to develop a Tin so we can add additional layers of functionality

```
// constructor for Container
function Container(Weight,Barcode,Contents,Material){
  parent.(Weight,Barcode,Contents);  // Grocery item bit
  this.material = Material;          // special bit
}

// constructor for Tin
function Tin(Weight,Barcode,Contents,Material,HasRingPull){
  parent.(Weight,Barcode,Contents,Material);  // Container bit
  this.doesTopHaveRingPull = HasRingPull;     // special bit
}
```

Now we can construct a more comprehensive Tin

```
tin = new Tin(100,"123456789","Baked beans", Steel, False)
```

where a chain of constructors called. Tin calls Container which calls GroceryItem.

---

91 Remember this isn't working code - it is generalised and simplified. Your preferred language will have its own syntax.

92 Actually a function of a class but we won't worry about the distinction for a while yet

Methods

> Functions of objects are called *methods*.

Methods mostly work on instances. (As opposed to constructors that worked on the class - to create an instance.) Is that tin of beans past it's best before date? Since best before dates are not unique to Tins...or Containers...but are part of GroceryItems that looks like the place to put a function which compares today's date with the best before date. Here are a couple of functions ...

```
function HowManyDaysLeft(){
  // Assume DaysDifference and Today() are built-in functions
  return DaysDifference(this.bestBefore,Today());
}
function StillOK(){
  return (HowManyDaysLeft() >= 0)
}
```

...but how do we make these functions part of GroceryItem? HowManyDaysLeft() needs the bestBefore field of a *particular instance* of an object and so it can't quite the same as the functions we've seen so far which have been available for any bit of code to use providing they gave all the required arguments. The answer is these functions are defined inside the class.

Record

```
Name of type{
  any data fields go here
}
```

Object

```
Name of class{
  any data fields go here
  any functions go here
}
```

Let's pick these two functions apart:

- Both are functions of an object. To save keep calling them "functions of an object" we just say *method*. (*We can then use function to mean a freelance sub-program*.)
- In HowManyDaysLeft we access the best before date using the code `this.bestBefore`. The "this" is unnecessary, but included to show you what is going on. Obviously bestBefore refers to the data field of a particular object. Err...Which particular object? Answer: The one the function was applied to.
  ```
  if (not myTinOfBeans.StillOK()){ throw myTinOfBeans away
  if (not anotherTinOfBeans.StillOK()){ throw another tin away
  ```
- In the StillOK method we call the HowManyDaysLeft method. Could we have put a `this.` in front of the call? Yes, but in practice there tends to be less opportunity for confusion between method arguments and method names.

Private data

Are you old enough to *insert legal restriction here*? Let me look at your record... Ah yes I see you were born on the 4th of April 1961, so that's OK. Hold on! Isn't that a bit personal? I don't need your date of birth all I need to know is are you old enough. Now look at the two GroceryItem methods. StillOK() doesn't leak the best before date which is exactly the sort of thing we want in the are-you-old-enough case. Data and methods

can be given *restricted visibility* to implement this. Details vary between languages.[93]

We still need the separate HowManyDaysLeft() method for other purposes. (Perhaps we want to sort items so that things that need using up soon are at the top.)

You will probably find that your methods tend to fall into two categories:
- Those that are based around 'how the object works'
- Those that make the object useful to the wider world

We'll see more of this later. For now don't be surprised to see method Foo() calling method Bar() which calls method Buz(). You can see objects with long lists of methods to give extra bells and whistles for external functionality that package a few core methods.

Method variations

Suppose we have an object which is some list, a list of things to do perhaps we might want to implement a method called AddItem(). Now what arguments should that method take? We could start with a string:

```
method AddItem(string : OneString){  // takes a single string
  ... appropriate code goes here
}
```

That's fine, but what about adding a whole array of strings in one go

```
method AddItem(string[] : ArrayOfStrings){  // takes an array
  ... appropriate code goes here
}
```

When there are two (or more) variants with the same method name it is called *overloading*. This might not seem much of a big deal with this simple example, after all we could have called the second version AddArray(), but sometimes it is very convenient to keep the number of method names down to a manageable number.

There is another style of method variation called *overriding*. A method is declared in a parent class which is fine for the general case but can be replaced by a more specific method - with the same name, so nothing needs to be changed. In our groceries example we might have a generic method Open() which is reimplemented for bags, packets, bottles and tins. If say Tin doesn't supply its specialised version then the parent's will be used.

Review

We've got some more work to do before all the basics of objects have been dealt with, but how far have we got?
- An object is a super-record with methods as well as data
- A class is the definition (where the methods live)
- An instance is a particular data record which can be manipulated by the methods of whatever class the object is.
- Classes can inherit data definitions and method definitions.
- Instances are created using a constructor method

---

93 Confusingly different languages may mean different things with the same keyword. It always pays to study and get clear in your mind each language's implementation of visibility specifiers.

- Data fields within an instance are specific to that instance
- Methods are *defined in the class* but *operate on instances*.
- Methods are handed down from parent class to child classes but may be overridden.
- Methods and data fields may be hidden from view

In the next part we'll continue looking at objects, but first a little bit of why they are so important.

Object oriented programming

OOP required a new way of designing programming languages and a revolution in thought. Even though a programmer would be writing the guts of a program in roughly the same way as in a functional language the stitched together of the parts is entirely different. In addition it changes the way programs are designed, the economics of programming, the way in which separate modules are developed and reused. The two snags are:

- Humans are not very good at juggling abstract objects
- Processors still need old-style lists of instructions

There are type of language where OOP is not relevant: Scripting(eg shell), definition (eg HTML, SQL) and special purpose languages.[94] Also it is often tempting to hack¤ some code together without the overhead of working out how classes interlock.

Components

So far it may not have become apparent that any program of any size consists of components, probably many that operate behind the scenes. Even with HTML you may get Cascading style sheets and Javascript scripts which are vital to the overall presentation of the page in a browser. You may be the one to program all of these components, but in the main you'll be using ready-made bits off-the-shelf, or bits that have been entrusted to somebody else to build and test while you get on with your part. Here are the important questions :

- Do you need to know the internal workings of those components?
  No.
- Do you need to know how to use them: What they do and how to make them do it?
  Yes.

Recall that I said you'd find your methods will tend to separate into those that did the workings and those that interfaced with the rest of the program. See the same split?

Design

It should be easy to see that if your program is used for a application where there are objects in real life that you can easily model these as electronic objects. Patient, doctor (both sub-class of Person), appointment, medical record item, medical history, medication, treatment, investigation (all three being subclass of action) and so on. But there are less concrete objects: Risk, cost-benefit, patient's attitude. Also lots of small helper objects : phone call, blood-pressure reading, contra-indication alert. Plus background objects : Electricity bill, pharmacy stocks, qualifications and so on. Strewth! What a nightmare - however is it going to be possible to computerise this lot - just for a patient to see their doctor about an ingrowing toenail? If one thing is

---

94 It is comparatively easy to develop a special purpose language (example coming up much later @@@) but building in object features is extremely complex.

connected to another thing which is connected to others and others where will it end?

The answer is to define a limit - draw a boundary. But within that boundary? That's why computer systems can be expensive, because when you begin to look at what's involved, under the surface there's a lot more going on than at first appears. The OO approach allows you to sketch on a piece of paper firstly the objects, secondly the relations between them and thirdly the inner workings. (Thirdly being what the programmer does a lot of.) It is often possible to 'zoom in' and 'zoom-out'. For example from a patient's point of view they might just have symptoms, treatment and outcome with some methods like go-to-the-doctor. Patient's history and other properties and methods might be internal to the patient object.

Design is a subject that will get a more extensive treatment in a later chapter. For now make sure you've got plenty of paper and some coloured pencils to draw the components of a system in exactly the same way you'd draw the components of a bit of machinery. Time spent on design is time well spent.

Programming Interface

Almost always called an API[95] the specification for how to use an object is what the user needs to know, and if you're a programmer making use of a wheel that's already been invented then you are the user. How many times have you tried to make something work but been frustrated by lack of instructions? The API is the knobs and buttons and user guide for what happens if you press them for an object. Inside there might be a maze of technological wizardry but the user doesn't need to know that, just for example how to turn that bloody noise off!

All of the above are different ways of saying that objects have an inside and an outside, a mechanism and a function. Depending on whether you are programming or designing will affect which side you are on... ...but you have to connect the inside and outside.

More on the technology of objects

Let's suppose you have a client who designs swimming pools. Part of the process will be calculating the volume of water. You want to be able to say something like aPool.HowMuchWater() to get the volume. Until you got involved with objects you'd start by writing a function something like this:

```
function HowMuchWater(Length,Breadth,Depth){
  return Length * Breadth * Depth;
}
```

and submit your bill for prompt consideration. WCPGW? Your client is astonished that you were not psychic and failed to divine that all her pools were not rectangular.[96] OK so now you're on your guard and you find there are circular, oval, rectangular and odd-shape and the rectangular ones have a shallow and deep ends but the others might have funny shaped bottoms. Groan!

---

95 A = Application

96 This really is typical of the level of information you get from clients, large and small, unless you really poke around and ask all those 'irrelevant' questions.

Lets start with the basic mechanism: The volume will be the area times the average depth. This will be the same for all pools. (So this looks like being a method of Pool.) Now what about sub-classes of pool? We can only *work out* the average depth in the case of the rectangular with deep and shallow ends, the rest we need to be told the average depth. What about the area? Circular pools: pi-r-squared, elliptical: Length times Breadth times pi/4. Rectangular : Length times breadth. Odd shape : needs to be given.

With this insight into how pools differ we can construct an ***object hierarchy*** as follows:

Pool
- Rectangular
  - EvenlySloped
  - Elliptical
- Circular
- OddShape

- Elliptical as a sub-class of rectangular!! Why not? We need length and breadth.
- All rectangular pools have length and breadth
- An evenly sloped pool can calculate its average depth
- A circular pool has a single radius dimension ...
  ... but we may need to give the user a diameter property to avoid confusion.
- An odd shaped pool is beyond simple maths so we'll say the data we need (for HowMuchWater() must be supplied.

All pools need a HowMuchWater() method. This implies they need (either methods or data) for area and average depth, but hold on! A Pool object doesn't know these things. Only the sub-classes have that information. It's like we're supposed to be inheriting from our children.

Here is how the conundrum is solved: The Pool object proudly declares it will provide the HowMuchWater() method but sneakily insist the sub-classes do the actual work to deliver the data for the Area times Average depth formula.

- Parents can insist children must implement a method for themselves
- Children can ***override*** a parent's method...
- ...Either completely or augmenting it.

Here is a skeleton to illustrate, notes at the end.

```
class Pool{
  method HowMuchWater(){
    return Area() * AverageDepth();
  }
  abstract method Area();
  abstract method AverageDepth();
}
class Rectangular inherits from Pool{
  field length;
  field breadth;
  field avDepth;
  method Area(){
    return length * breadth;
  }
  method AverageDepth(){
    return avDepth;
```

```
  }
}
class EvenlySloped inherits from Rectangular{
  field shallowEndDepth;
  field deepEndDepth;
  method AverageDepth(){
    avDepth = (shallowEndDepth + deepEndDepth) /2;
    return parent.AverageDepth();
  }
}
class Elliptical inherits from Rectangular{
  method Area(){
    return (length * breadth) * ( PI / 4);
  }
}
class Circular inherits from Pool{
  field avDepth;
  field radius;
  method Area(){
    return PI * radius * radius;
  }
  method AverageDepth(){
    return avDepth;
  }
}
class Oddshape inherits from Pool{
  field avDepth;
  field area;
  method Area(){
    return area;
  }
  method AverageDepth(){
    return avDepth;
  }
}
```

- Methods to create the objects and get the necessary base data have been omitted.
- The 'language' in the code is my specially fudged one for clarity. You should be looking at the documentation for YCPL and spotting the similarities and differences.
- `class Foo inherits from Bar`. This is revision. Notice where the block extends to. It encompasses the fields and methods for class. Note the initial upper case letter naming convention.
- Pool doesn't have any data fields defined. It promises to be able to deliver what is asked for and that's all somebody using a Pool object (or sub-class of Pool) needs to know. Any Pool object will be able to respond to the method call HowMuchWater().
- In fact we can never have any objects of Pool, only of sub-classes because some of the methods have not been defined, they're ***abstract methods***. They've been *declared* but not *defined*.
- An object of class Rectangular can be instantiated because it supplies the required definitions. It also has the supporting data fields.
- Class EvenlySloped re-defines the AverageDepth() method which was originally defined in Rectangular (its parent class). It is ***overriding*** the parent's method....
- ...But then it calls the original method. (Walk-though coming up.)
- Class EvenlySloped used the parent's Area() method.
- Class Elliptical overrides the parent's Area() method. (Unlike when

EvenlySloped.AverageDepth() also called the parent's method, here the method is complete in itself.)

- Class Circular is similar to Rectangular. It can provide what is demanded of it, namely the Area() and AverageDepth() methods, but its internal workings are completely different to Rectangular.
- The OddShape class looks like a lot of work for a little result. Why bother with those methods? Couldn't we just re-implement the HowMuchWater() method, overriding the version in Pool to use the data fields directly? No. Firstly because we won't have implemented the abstract methods declared in the parent. Second even if we could, how are we to know that some other method won't need Area() and AverageDepth() - possibly as the program develops at a later date.

@@@ could we have put avDepth etc in Pool obj?

Let's do a walk-through. Suppose we've

Created an object of class `EvenlySloped`

Called it `es`

Somehow given it the necessary dimensions

So now what happens when we execute the code

```
vol = es.HowMuchWater(); ?
```

es is an object of class EvenlySloped. Where is the definition of HowMuchWater()? Not in EvenlySloped... Not in Rectangular either... Ah here it is in Pool. So let's execute the code:

We need the Area() method. Here it is... ...but No! Area() is an abstract method so we've go to go looking for it in the child classes. Is it in EvenlySloped? No. Is it in Rectangular? Yes. So execute the Area() method:

We need the length field : (es inherited it from Rectangular)

And the breadth field : (es inherited it from Rectangular)

So do the sum and return the result.

We need the AverageDepth() method. Here it is... ...but No! Area() is an abstract method so we've go to go looking for it in the child classes. Is it in EvenlySloped? Yes. So execute it:

We need shallowEndDepth and deepEndDepth from es. OK

Do the sum and save result in es's avDepth field (inherited from Rectangular)

Now find the parent's AverageDepth() method. OK so execute it.

Easy! Just return es's avDepth field

And return the result just given to us by the parent's AverageDepth() method.

Now we have the necessary results from Area() and AverageDepth() we can do the sum ... and return the answer to the caller.

Phew! That was a lot of to-ing and fro-ing. The question in the back of your mind, having seen the code which seems a lot for a little and the walk-through which seems a maze of 'wrong department - you need to ask elsewhere' buck passing, is "Is this really justified?"

The short answer is yes. It works well in practice. The reason it works well in practice is that to use it all you need to do is call that one method and all the details are sorted out. When you get into a car all you have to do is turn the ignition key to start it. One method that works for all cars. In times past there were special starter buttons, crank handles, running on petrol then changing to paraffin, plus a bit of shouting and praying. Because automobile engineers spent the time improving the machinery you can now reliably start any car. Also you can easily explain to somebody else how to do it.

Information hiding

Nowadays most of the guts of a car are hidden away so you can't tinker with them.[97] Admittedly the complexity of modern engine systems makes tinkering very unlikely to succeed in fact almost certainly something will be put out of step with everything else adding yet another problem.

The same applies to the data inside objects. We want to keep itchy fingers and prying eyes out of our internal workings. There are two aspects to this:

Firstly as we've seen you shouldn't need to know how the object goes about its internal business if all you're doing is using it.

Secondly we may want to take definite steps to control access.

Let's consider how we might validate dimensions given to our swimming pool program. Remember that these dimensions will be used to work out the area of tiling, amount of excavation, heat losses and other costs. If there is a loopy input to one of the fundamental dimensions then there could be expensive trouble all down the line. (A bit of WCPGW in practice there.)

One way of setting say the average depth is by coding like this

```
myPoolInstance.avDepth := 0.95;
```

WCPGW: What units are we using? What are sensible values? We could possibly ask the calling program to check these but (a) that's only a *please* and (b) it is one more thing for the user to consider that the object itself should know. Do you remember those pointers in data structures that pointed to the next item and so on? What would happen if these were tinkered with? Gloom! So it is vitally important that some mechanisms are in a case marked "No user serviceable parts inside".

The other way of doing this is to have a method for setting the average depth which does the necessary protection work.

```
method SetAvDepth(DepthInMetres){
  if ((DepthInMetres > 0.5) and (DepthInMetres<1.4)){
    avDepth := DepthInMetres;
    return TRUE;
  }else{
    return FALSE;
  }
}
```

So the object itself validates what it will and won't accept.

To finish this off we have to prevent outside code accessing avDepth.

- This is done by flagging the field with a suitable keyword
- Keywords depend on language ("private", "public" and "protected" are common.)
- Typically different amounts of hiding are provided for. You can restrict a field to a single class, or that class and associated or make it generally available. The exact details vary from one language to the next.

Same method - different arguments

Finally it ought to be made clear that most languages allow you to specify methods that take different arguments but still use the same name. For example we might have a

---

97 Which can be frustrating if it doesn't work and you can't get in to find out why or try bypassing, changing, testing or boosting something.

method HowMuchWater() and another HowMuchWater(Units:String). The documentation would tell us that HowMuchWater() with no arguments assumes cubic metres, but we could use the alternative form with an argument of "gals", "m3", "ltr" or "ft3" to get a result in gallons, cubic metres, litres or cubic feet. (Have you guessed what the definition of the no-argument form is? `HowMuchWater("m3");`

Review

Another chapter comes to an end. We started with the storage of compound data fields as a package. Then the breakthrough of adding functions to types to make them classes with methods, inheritance, information hiding and *polymorphism*. (That's a jargon word for different ways of executing the same method depending on what sub-class is involved. HowMuchWater() is an example.@@@ Check this@@@)

There is a barrier to getting stuck-in to objects which you may be feeling at this very moment. There is a lot of thought, organisation and foundation building to be done before you can run a program. This 'bottleneck' of 'hope this is right' and the long wait before 'the lights start blinking'[98] soon becomes easier to bear after suffering the 'millstone' of 'what's going on here - and why does it sometimes go wrong' that plagues programs written from the hip.

---

98 For some light computer lore going back to at least 1959 look up "blinkenlights".

8. Progress review

I am hoping that you are keeping up your practical studies using YCPL. You can whisk all the way to the end of this book without touching a keyboard but you'd be better off re-reading what we've done already. Knowing what you know now will make it all so much easier and the details (and fudges) will become clearer.

In the next chapter we will write a whole application using the programming knowledge gained from this book and the language knowledge you've acquired from elsewhere. This should be a useful exercise in the slog of programming to get a working application. The less you're looking forward to head down programming, consulting the manual and debugging the more valuable it will be. If you're already a whizz at doing OOP in your chosen language then simply reading it may be sufficient.

After the close up on code we'll step back to look at databases. Then a bit more on the essentials of good programming practice that I've been missing out. Then some wider issues surrounding programming and programmers.

```
AND_RELAX
- If you didn't see this coming when you went to the supermarket
- all those aeons ago then you're not yet thinking like a programmer.
- As well as WCPGW think: "What could possibly be better".
0.  With kitchen
1.  glass = kitchen.cupboard.Find(Glass)
2.  corkscrew = kitchen.drawer.Find(Corkscrew)
3.  bottle = kitchen.winerack.Find(Bottle)
4.  corkscrew.Open(bottle)
5.  bottle.Pour(glass)
```

Cheers![99]

[99] If you were a suit you wouldn't need this. But as a programmer with your nose to the grindstone, crystal sharp vision and strict adherence to logic you occasionally like to sit back and let the details dissolve into a warm rose-tinted glow.

9. Let us code

This is a hands-on chapter. You will need:

- To be familiar with your development environment
- Be familiar with the basic coding syntax of YCPL
- Have written some example programs in YCPL and got them to work
- Be using a language that supports OOP
- Be using a language that supports reading and writing to files
- Be using a language that supports interactive on-screen applications

And to have read and 'be happy' with all the preceding chapters.

We will develop a diary application which uses the minimum of display and interaction. At the very least you need to be able to display a list on the screen (diary dates overview), get some input (add/edit item), provide controls for selecting a diary item and overall actions such as save to file.

One of the reasons for having this chapter is to get you familiar with common practical programming features. So far we've skipped all mention of user input and display and fudged the storage of data. You need to know how to do these things. It may be difficult to learn for YCPL given the documentation you have to hand, in which case get better study materials! Don't expect to learn this list of requirements in just an hour. Sorry, but if it was easy then *everyone* would be doing it. You can't be a programmer without slogging through lines of code and examples that look straightforward in the book but complain when you try to get them to work. It will be hard but that's because your fingers and brain are not used to it. One day soon it will get a lot easier and you'll be able to concentrate on the interesting bits of programming.

Code in this chapter will be written in Fudge. Fudge is a language for humans that looks close enough to a computer programming language to make conversion into YCPL straightforward. We've already seen it in chapter 7.[100]

Design

The normal way to design things is to discuss what you want to do, get a feel for the nature of the beast, where the critical points of failure and bonuses of goodness might be, and before you know it you'll have a bunch of component parts. This is the top-down approach which works from the general to the specific.

> Designing is like those matchstick puzzles where you have to get a shape by moving only three matches. You can stare hard and look at the 'obvious' way without success. At some stage you need to back-off and preferably discuss what you're trying to do with somebody. Sleep on it to play with the possibilities.

---

100 I'm making Fudge up as I go along.

You'll also be thinking of technological issues which may need resolving. For example what will the screen look like, how will you verify users, will it run fast enough, what method should you use to store the data, how will it integrate with X, what happens if we want other-language versions? This is the bottom-up approach which aims to provide the critical modules (or at least prove they are feasible, and how much they will cost) from which the application will be built.

Somewhere in the middle these approaches meet.

Then you review the design, preferably after a few days of ignoring it. Try and get an experienced person to review it with you - the more experienced the better. More on this in a later chapter.

Application specification

The diary will display a list of dates for which we have 'something on'. It will distinguish between weekdays, weekends and public holidays and be able to name days of week and names of public holidays. As well as the typical reminder and holiday entries it will have a facility to indicate shift work. For example 4 days on 2am-10am, rest day 4 days on 10am-6pm rest day 4 days on 6pm-2am etc. Diary entries can be created, edited and deleted. A facility to spot clashes (eg 'Doctor 11am' when doing the daytime shift) would be useful, as would being able to automatically fill in a set of days shift pattern.

DIARY_TOP_DOWN_DESIGN

0. With lots of paper
0.1 Different coloured pens/pencils may be helpful
- There will be lots of sketching, re-drawing etc.
- This is normal
1. Draw boxes for the components in the diary
1.1 these will become your objects
1.2 give the boxes names
2 Experiment with an object hierarchy (sub-classing anyone?)
3. List the data fields inside each object
4. List the methods this object will implement
5. Make side notes about anything you think about

DIARY_BOTTOM_UP_DESIGN

0. With notepad, development environment and manuals
1. Sketch the user interface as you imagine it would appear
2. List the display and interaction logic
2.1 Sketch the various activities in bubbles
2.2 Joined by links marked edit, delete etc
2.3 Review 1.

3. List what file operations will occur
4. List any particular bells and whistles you envisage
5. For each of the items listed in 1. to 4.
- Looking at Risk (5.1) and Resources (5.2)
5.1 Will that be easy, hard or don't know to implement
5.2 Will that take a lot or a little time implement
6. For each of the items marked as hard or taking a lot of time
6.1 Is there an alternative, easier, simpler approach?
7. For each of the items marked as don't know
7.1 Experiment with some actual programming to discover how to do it.
7.2 If it still looks difficult look at 6.1
7.3 If it is impossible then change the requirement or change language
8. Collect your conclusions into a short, easy to read list with the key functionality listed.
- The object of 8. is to form a plan of work for your many minions (ahem)
- to go off and beaver away at ready for you to use in your top-down
- inspired program.

See I told you this chapter would be hands-on and difficult. This *is* programming. It isn't the sort you get in *Learn YCPL in 3 days*. It will take months before you're fluent at design and even then it's littered with many unknowns, complexity and really interesting possibilities that won't quite lie down. Personalities and technique come into it a lot. Above all, give yourself time and always sleep-on-it.

In my opinion a lot of poor design stems from two things:
1 : Skimped design. (F/note@@@ Watch out for spending more time on fancy design tools than basic ideas.)
2 : Lack of practice. Get into the habit of reaching for a bit of paper every time a task appears on the horizon. In the early days, say the first year, do that even if you know the task isn't yours - you'll be able to compare and learn.

My top-down design

I have left some bits out so we can see how logical making alterations to the code is for 'oh I didn't think of that' and 'that would be a good idea'. I have added some bits in we won't be implementing - but please feel free to enrich the application yourself.

Choices

DIARY contains:
an *ordered list* of DAYs
a file or filename for reading and writing to
possibly some configuration for colours, layout, date and time formats etc.

We'll stop in the middle of DIARY because there is another way:

DIARY is a sub-class of *ordered list*
The list items are DAYs
Extend by adding fields for
a file or filename for reading and writing to
possibly some configuration for colours, layout, date and time formats etc.

Is one method better than the other? It depends. One way to decide is to ask are you trying to make 'a better version of the class' or just using it as a component.

- Perhaps in this case the first method is simpler and more appropriate.
- However DIARY might be a sub-class of some other object which is good at showing itself on the screen. That could come in really handy.

Ready to reuse components

Suppose you already have an ordered list class, or a list that is sortable. If that class knows how to read and write itself to file you already have a lot of the functionality for the diary object. It is fairly likely that YCPL does have such a class ready to hand or you can make one easily. If you have to make one then it will be really useful in the future for all sorts of things.

> Code reuse is one of the primary objectives of all programming.
>
> One of the requirements to make this work is for code to be self-contained. OO code meets this criteria quite well. (Another is can you find the stuff when you need it? We will cover this sometime.)

Data picture so far

DIARY : Mostly a list "DAYS" with configuration bits
DAYS : List containing DAY objects
DAY : Data fields for date, and special day name

How do we connect the ENTRY objects (eg Time, duration, note) into the system?

- We could have a separate list of ENTRYs if each had a date, then process that list (say in date order) to give the diary listing.
- Or we could put them inside DAY as required so that each day is a self-contained object

There isn't a correct answer. For example if we were linking this to a database with an events table probably the first solution is best, but if we want our data to be contained like Russian Dolls inside our objects then the second. As we are doing our own storage we will use the second, list of lists, approach.

DAY : Data fields for date and special day name
Sortable list of 0 or many EVENTs

Methods

DIARY : Load and save from/to specified file
Display list of events (with range of dates)
Provide user controls to (initiate) select, add, and edit ENTRYs
Possibly provide a way to alter configuration settings

Notice 'display list of events' doesn't say how. The mechanism is going to be hidden from the user and will be delegated as follows:

DAYS : Load and save to/from file
Add and delete DAYs

Sort
Enumerate¤.

Nothing about displaying a list? Err...well that is sort of covered by enumeration. It will work out all right when we get down to code.

DAY : Load and save from/to file
Display (self and/or EVENTs)
Edit public holiday name
Add / remove EVENT
Provide facilities for user to add, edit and delete EVENTs
Tell date
Tell if weekend, weekday or public holiday
Tell if any EVENTs
Tell if any clashes of EVENTs

This is getting a bit more like it. Obviously DAY is going to be a major component

EVENT : Load and save from/to file
Display
Tell start, finish and duration

That's enough top-down detail for now.

- The ordered list class gets used twice
- File I/O (Input/Output - Reading and writing - Loading and saving) keeps on getting delegated. We will investigate this in a moment.

My bottom-up design

1 What is the display infrastructure?
2 File details need to be clarified
3 Inputting data and editing items is going to be a bit tricky
4 Dates are always slippery : We have to add 8 hours onto 6pm...
5 Is an ordered list class ready to hand?

Some of these I can only make a stab at because your system is likely to be different from mine. Nevertheless let's have a go and see where we get.

1 What is the display infrastructure

The simplest (and most universal) we can think of is a list of dates/entries down the page. If possible we want a delete/edit/add button or selecting ability for the items. We need some general controls.

- Console
  (A console interface is a glass teletype¤ Output is sequential lines of text. Input is typing at a keyboard.)
  - We can print a list of dates easily, but could get stuck if the number of lines goes off the top of the screen.
  - We will have to give a list of commands such as Save, New entry etc.
  - ...and perhaps number the entries so we can say something like Delete 16

- Web page
  (A web page is not very interactive but has improved layout and facilities.)
  - We can easily list events in a table. No worries about losing entries off the top of the screen as web browsers let us scroll.

- We can put Delete and Edit buttons/links against each entry
- Overall controls will need to be buttons/links
- Getting some inputs will need to be done using forms.

- GUI
  (A Graphical User Interface allows control of the presentation and interactive capture of mouse and keyboard.)
  - GUIs are often set up by a visual programming environment. This would speed the work of organising, painting controls and capturing events. However the simplicity of the application should allow us to do these by-hand quite easily.
  - Separate child windows for adding and editing entries.
  - Input and selection should be straightforward if using predefined controls.

2 File details need to be clarified

The way we'll use the file is to read in all the data at the start, run the program then save any changes at the end.

Do we need this file to be
- Human readable?
- Machine readable?
- Accessed by multiple applications?
- 'Live'? (Contents could change by outside influence when we're using it.)

These issues are a summary of quite a long detailed list which we'll come back to in a later chapter.@@@?

Object substitution
Suppose your diary project was to be implemented on a number of systems you could write a Display object to get the program working on your development system and then have alternative versions to swap-in. The majority of your program is blissfully unaware of any changes as it still uses the same API to talk to the display. Or your part of a project will be using an object which somebody else is in the process of developing. In the meantime you could write a fake version that is enough for you to do some testing until the real version is available.
- You can see the importance of defining, documenting and sticking to the API.

Human readability would be a bonus, especially for a first application where it can be really helpful to see what's going on and tweak the file by hand. But since you can't join the programmers air-force without doing some loop-the-loops we'll say machine only binary. We don't need any other application to read or write our data. (And if we did we might be able to reuse our objects to do that.)

- If the object class in YCPL supports serialisation¤ you're laughing. If not you're going to have to write read-me and write-me methods for your objects.

We could base our data file name on the program name or give it a fixed name or allow the name to be specified on the command line¤. Let's do the latter, but say that if no command line argument is supplied we'll use "diary.dat".

3 Inputting data and editing items is going to be a bit tricky

Why? Because user interfaces vary so much as just discussed in 1.

And also because we need to be able to pick items to edit.

And also we need to be able to validate input and report validation problems.

And also because we will be doing different tasks at different times. Sometimes browsing the list of events, others editing a particular event, others asking for a particular selection of events. (eg When am I going to the opera?)

You need to be able to

- have a way of selecting an item to edit/delete
- to add/edit an item's details
- report data input errors
- take commands like save changes

One way of clarifying what will be happening is to list (or draw boxes) the separate functional aspects of the program. (In this case List events, add/edit item, delete item, load/save data, sort items, control display.) How do these fit on a virtual screen? For example 'load data' may be automatic ... put perhaps needs a progress bar... or a dialogue box to ask which data file to open. You will see that the functions begin to separate into logical groupings. You've seen this happen in practice when for example you interrupt your spreadsheet figure-work to play with a graph.

Just because you have a number of separate functions doesn't mean they have to go onto separate 'pages'. For example your diary might have an events list on one part of the screen, a menu and a details display/edit area at the top of the list. There is an art to getting the right balance between flipping between screens and having too many different functions on one page. (Programmers are supposed to be able to deal with complexity and want a work bench with everything just a single click[101] away. Don't take the clutter of an IDE as a model for general applications.)

Now, in a different colour, indicate where actions happen. For example "Save and exit" against 'Finish' button. As you do this you'll probably be thinking about the controls you'll be providing for user interaction as you go. Don't be afraid to remove or add controls. (If, after a while you still can't decide you probably need to step back and have a fresh think.)

There is a clever way of writing in the actions that result from an interaction. What would the pop-up hint say? What could be simpler.

In some applications where there are complex procedures set in motion you probably need to either refer to (or write) a full description of what happens. This ought to have been done at the top-down stage though so if you

> Design is closely associated with systems analysis. The top-down needs good analytical skills, the bottom up needs people who can create useable user interfaces and take the time to check that the details of what's proposed match real-life experience. (Beware - They never do.)

101 **Better still just a single keystroke.**

find yourself doing this that's an alert to revisit the top-down design phase.

4 Dates are always slippery : We have to add 8 hours onto 6pm...

I've written an appendix @@@ on the subject of dates and times. You should read that then consult the documentation for YCPL. Most systems bundle date and time in together so 'all you have to do' is add (1440 * 8) seconds to work out the end of an 8 hour shift given the start date/time. In all events you need to experiment with adding X time to Y date/time to get Z date/time.

Now you'll want to display the date and/or time. Generally you'll find there are ready to use functions for this which need a little fiddling with. You'll need to track down the documentation and try a few experiments.

5 Is an ordered list class ready to hand

Yes. If it isn't built-in you'll be able to snatch one from the Internet or use the same functionality from say built-in sortable, extendable arrays.

> Roll your own list.
> That's a good exercise because it is so simple, so basic and such a good exercise in how bugs grow. For now we can use something ready made, but working with lists of various sorts will be necessary if you're going to have the confidence to stitch together complex data structures...
> ...and to be aware of what's possible a few years down the road when you suddenly get something out of the ordinary.

Review

So we've looked at the overall concept and broken it down into the assemblies that we need to make it work. We started with ideas like "list the dates" then refined them into "a date lister object which does X and contains Y and delegates Z to Y" (Top-down)

Then we investigated the practicalities of sourcing the parts and would they be suitable and do we have to modify them. This involved looking quite closely at how the program would interact (with a user in this case) and ensuring we have the competence to glue everything together.[102] (Bottom-up)

That's taken a long time!

By now we could have coded the whole thing! Well actually you'd have typed a lot of code and some of it may have done something. Whether the options were fully investigated, whether you used the right object model, whether you selected the right way to deal with dates and many other 'whethers' are a matter of speculation. It is very

> Collect a good set of tools for the design job. Something like:
> • Paper, coloured pens
> • A filing system
> • Time to look at alternatives
> • Confidence to question assumptions and investigate details
> (More later.)

[102] We might also be working on sample data or test cases in order to prove the system. More later.

tempting to wallow in cutting some code. The pressure to see something on the screen is enormous. But you're not in a sprint, and if your colleagues have rushed off in a lather of code just ignore them. They will be the ones puffing and panting at the end or who fail to finish the job to a reasonable standard while you sweep up on the outside in full stress-free control, knowing where you're going and how far it is to the finishing tape.

At the end of the design phase you should be able to
- explain what your proposed system will do
- explain how it does it
- identify a list of tasks to be done

And, to the sound of arms being twisted very hard up backs, how much effort it will take.[103]

So let's build the application

Because you are working with YCPL in your environment it is not possible for me to give you precise instructions. However we have a set of components that we can all build which means there is a framework for a plan of work. In this case I'll work from the outside inwards. In all cases I'll make the very basic functionality work before going on to details. If you feel the need to adapt and enhance please do so.

The syntax is whatever seems easiest to follow. Normally the code would be commented to explain the object and its methods but as you know what's going I'll let you add those yourself. YCPL probably has some conventions for this which are worth reading about.

Diary object

DIARY contains:
    an *ordered list* of DAYs being the
    a file or filename for reading and writing to
    possibly some configuration for colours, layout, date and time formats etc.

```
class Diary
    fields
      days : DayList ;    // will list the days
      fileName : string;  // won't be read/written to file

    constructor Diary(FileName:string){
      Diary();
      LoadFromFile(FileName);
    }

    constructor Diary(){
      // [possibly call inherited constructor here]
      this.fileName = 'diary.dat';
      LoadFromFile();
    }
```

---

103 When we look again at design in a few chapters time we'll add a bit to this.

```
    method LoadFromFile(){
      if (FileExists(this.filename)){
       fileStream = new FileStream(this.filename);
        days = new DayList(fileStream);
      }else{
        days = new DayList();
      }
    }

    method LoadFromFile(FileName : string){
      this.filename = FileName;
      LoadFromFile();
    }

    method SaveToFile(FileName : string){
      this.filename = FileName;
      SaveToFile();
    }


    method SaveToFile(){
     fileStream = new FileStream(this.filename);
      days.SaveToFile(fileStream);
    }

    method Display(){
      days.Display();
    }
}
```

Let's halt there and have a few words about the code.

- Reminder : My naming conventions are to start with upper case for
  - Class names eg DayList
  - Function arguments

  These are not the same as Java where the convention is to use capitalisation only for class names.
- YCPL may or may not be case sensitive. Even if it isn't, make an effort to work as if it is. Not only will it save you grief if at some stage you encounter a case sensitive language but it will help you almost sub-consciously recognise the significance of names.
- The constructor Diary() and methods LoadFromFile() and SaveToFile() are *overloaded*. This is a very common feature of OO code. (Some OO languages practically insist you have a no-argument constructor.) Notice also how the versions with arguments call their no-argument versions. This is usual. You tend to call the inherited constructor and alternate constructors before fleshing out the object.
- "this." is given for clarity. You need to be clear about the rules YCPL uses to identify object field names and others.
- How does the Daylist object know where to display itself? Good question. It may be that there is some method accessible to DayList objects (and their components) that is provided by the system and taken for granted or which they can call on at will by 'reaching out of their box' to access some part of the environment. In Java graphical display has to happen via a Graphics object which would be available to the main program and is passed to all components that want to draw on it. Also in

Java, StdOut is accessible throughout all parts of the program. Thus with Java the programmer has a major choice to make: Cheap-n-cheerful command line or all-singing GUI with additional overhead. In the GUI style the method calls to components would be passing the object to display with *and also* need to keep a note of it for itself and to be able to pass to its components in turn.[104]

- Hold on! LoadFromFile() doesn't really do anything except call the constructor of the days list. This is correct. There is no other data that the Diary needs to load so it gets the days list to do its own loading. When you think about it which object knows more about loading the list of days - the days list of course, so that's where the method goes. This hand-me-down method style is extremely common in OO code. We'll see it used in saving and displaying also.
- FileStream is an assumed version of a stream opened on a file for reading. Streams were introduced in chapter @@@ and you should have had a look in the documentation for YCPL to get the details.
- So WCPGW with the LoadFromFile(no argument) method?
  The file might exist but not be readable (For various reasons such as we don't have permission, it is a directory or it is broken in some other way.) so that The FileStream construction process fails. How can we deal with this?
  (a) Some way of testing and taking avoiding action?
  Or
  (b) Trapping the error if it arises and taking remedial action?
  The general answer is "a mixture". 'Prevention is better than cure' but very 'expensive'. We try to foresee and test for error or abnormal conditions, but sometimes we must deal on the fly with ***runtime errors***. The classic runtime error is divide by zero. The "sine of the angle between Tuesday's high tide and the length of the rhubarb patch minus the number of Aunt Augusta's bunions giving her gyp" is unpredictable and could be zero - so we might elect to ***handle the exception***, should it ever arise, by ***trapping*** it.

  YCPL will have its own way of handling run-time errors. Possibly some construction that looks like

```
try
 something
except
 handle exception
```

  This is called ***exception handling*** and it is *vital* you know how it works, how to deal with awkward spanners in the works of your code from a 'how do we handle this' point of view - and also how to generate a sensible error message for human consumption if that is required. Run-time errors tend to be called ***exceptions***.
- WCPGW with the file stream? Something! There is always the small risk of not being able to access (or send/store) data.
  - Does this matter? Probably yes. Suppose for example that your program writes its data to a removable disc drive before shutting down. You system may be being shut down and in your rush to go off to the pub you take out your disc...
    ... and your program being closed down by the system waits, and waits, and waits. Or it gives-up and reports an error message that is displayed for a

---

104 See the review for an important note. ??@@@

millisecond on the screen. Either you get very thirsty or you lose your changes without being aware of it.
    - Can we catch it? If we want to.[105] Probably by more `try...except`
- W else CPGR with the file stream? What LoadFromFile() does is open a file and line it up ready for reading. Then we get the days list to read itself from this file stream. Job done? No. We haven't closed the file/file stream after we've finished with it. Is this a problem? Quite possibly. It is standard practice to make sure that every resource you program borrows for its own use is eventually freed so that it becomes available to the rest of the system again.

> The terminology releasing resources is:
> - Objects and memory are ***freed***
> - Files and network connections are ***closed***

  So we add in a line at the end of our LoadFromFile() method will close the file stream.
- WCPGR with closing the file stream at the end of the method? What happens if there is an error in the middle of the days list being read? The program crashes and never gets a chance to close the file. Bearing in mind we are probably trying to solve one problem, we don't want another piled on top. A snowball turns into an avalanche. In this situation there might have been some transient failure but then the operating system has booked-out the data file to the program that crashed (***locked*** it) and so it isn't available until a reboot.[106]

  Modern languages have a construction like

```
try
 do something
finally
 tidy up
```

Doesn't it just get harder than you ever dreamed of? Especially if you have done some computer programming already you may be feeling the strain of all this WCPGR to the n^{th} degree. Well, yes it is a bit of a strain thinking of everything up-front... ...but it will soon become second nature to put in safety-nets.[107]

Just like athletes don't fall over or give-up because when you can't see them they're training, training, training so if your programs don't fall over the reason is because you too are working out of sight on the less glamorous graft.

In practice it isn't too bad. It beats dealing with strange faults over the phone for which you're probably not getting paid, when you haven't looked at the program for a year, and

---

105 At least with a modern language.

106 A reboot *might* clear a file lock, but some times you're on the phone for ages trying to get some poor user who has never seen a command line to delete certain locking files without doing collateral damage.

107 Also you should start to recognise the mistake you make every time Doh! - Putting in traps to detect your blind spots is a salutary lesson.

when the customer is not equipped to solve the problem even if they could describe it to you.[108]

```
method LoadFromFile(){
  if (FileExists(this.filename)){
   fileStream = new FileStream(this.filename);
    try
      days = new DayList(fileStream);
    except
      raise Exception("Prob with" + this.filename);
    finally
      filestream.Close();
  }else{
    days = new DayList();
  }
}
```

Look again at this code for the LoadFromFile() method.

- `try` says protect this block of code by assigning exception handlers to it.
- `except` and `finally` define *exception handlers* which *catch* exceptions
- `raise` *throws* an exception. (YCPL might use the keyword `throw`.) What's this: Creating more exceptions! Actually we are catching one and throwing another and so passing the error 'up the stack' to for the caller to catch. If we didn't do this some file error message would be passed up, now, (if we can't deal with it) we can tailor the message.

Exceptions propagate up the *calling stack* with each function passing the error back to the caller and so on, until the main program is reached in which case there is generally some unpleasant technical message called a stack dump or just "error 123 at 123456". As a programmer you can't stop something going wrong, but you should be able to make it a soft-landing - one which doesn't involve you in technical support calls.[109]

Back to the keyboard

Your plan of work should be in two phases:

1 Put exception handling into Diary. You will need to look up the subject in the manual for YCPL.
2 Develop a Diary with absolute minimum functionality (for the time being) so we can exercise what we've got and weed out those bugs.

The main program

Somewhere you'll have to write a program that calls Diary. You know how to write this program because you've had practice, not least in chapter @@@. The guts of this will

---

108 True story : "Your program won't print" says a customer from a national engineering research centre to me. Eventually it turns out, after checking version numbers, operating system, doing a walk through and other frustrations that the printer isn't connected to the mains, there is no light on, they don't know where the on-off switch is. Important note: Most users have the technical knowledge of a cuttlefish - if that. There will be some words of wisdom at the very end on this subject. Q:H@@@ow many Essex policemen does it take to switch off a TV? A: More than two.

109 More about the very important subject of exceptions in a later chapter.

probably be (as written in Fudge)

```
d = new Diary();
d.Display();
d.SaveToFile();
```

Some programming languages insist you write this in a separate program file and 'know' where to find it, others let you write write separate code files if you wish but need to be told to include the separate file by name, others allow you to mix main programs and objects together. Whatever happens you need a filing system, it needs to be rational, reliable, inclusive and backed up. See the filing system appendix for more on organising a code base.

Will it run?

Just for the fun of it what happens if you try and run your main program. We know it can't possibly do anything because we haven't got a working Diary object. You should get to the stage where the Daylist object can't be found. We did this exercise so you can begin to associate the various errors being reported with the various problems with your code.

Daylist

DAYS : List containing DAY objects

Let us make a Daylist object.

```
class Daylist inherits from SortableList{
  // No no-arg constructor - use parent's

  constructor DayList(Fstream :  FileStream){
    SortableList();  // call parent constructor
    //@@@ reading file will go here
  }

  method SaveToFile(Fstream :  FileStream){
    //@@@ writing will go here
  }

  method Display(){
    //@@@ display will go here
  }
}
```

Hey what a swiz! None of the really useful stuff! Have you seen the hucksters operating the three card trick? What could be easier than finding the queen out of *three* cards as dealt in front of you? Answer: Finding the queen when there's only *one* card! What we're doing is taking one step at a time in the same way - in a sort of find the bug game. If there's a bug then it must be under the new card because we've previously eliminated all the other cards.

- @@@¤ is a convenient tag to tell us we must come back here to finish off.

- SortableList is an imaginary class. With any luck you should have something matching this built-in to YCPL or easily available.

Add some temporary "I'm here!" message code to these three methods:

- If you are using StdIn/StdOut you might want to put some 'StdOut' message at the head of each method.
- If you're using a GUI likewise a place-marker progress message.

So incorporate this spavined DaysList into your main program and run it. In a perfect world the program runs, possibly displays three messages from the methods in DisplayList and finishes without errors. Back in the real world it is up to you to weed out the typing errors, other mistakes, get a grip on tricky bits hidden in corners of the manual and at last have a working program.

By the way, the box on the right is a reality check for experienced programmers. Programming *always* takes longer than you think. Don't worry about it: Five minutes spent at this stage will work out to fifty minutes of debugging saved and five hours less of dealing with problems when the users try it out.

Hurrah!

You now have a working program as a scaffold on which to build more functionality. An experienced programmer would probably have coded all the objects in one go before sticking them together because they would be making fewer mistakes and also not be phased by the complexity of the interactions of the methods.

Finishing off DaysList

The load and save methods are obviously 'mirror-images'. Whatever format we chose to save needs to be compatible with loading.

Persistence shortcut

Persistence is tech-speak for making a permanent copy. This is what we are doing, and of course is such a common requirement that you'd have thought it would be built-in to all objects as a matter of course. We'll you might be surprised that many OO languages are a bit hesitant about it.

The process of converting the internal representation of an object into something that can be written to a file is called *serialisation*. To read an object from a file you'd unserialise it. By the way, in general, serialised objects are not portable between

Note for experienced programmers. If you have been reading without coding so far here is an exercise for you.

1. *Write down* an estimate the time it will take you to get this program working to this level, including how much of that time will be spent from notional 'finished the code' to actually getting it to work.
2. Do it.
3. How realistic were your estimates?
4. Write down where you were rusty, forgetful or sloppy.
5. Pin up your self assessment somewhere where you can see it for the next week or two and reflect upon it.

different programming languages or even dialects.[110] Use it for private storage of working data rather than publicly shared information.

There is a nifty wrinkle to serialising which is when an object serialises itself, as well as saving things like the class name and ordinary variables, it then serialises any component objects and so on. Unserialising works in reverse. [111]

Have a look in your documentation now because this will be an ideal use for serialisation.

Home made serialising.

We'll look at the saving method before loading method.

```
method SaveToFile(Fstream : FileStream){
  Fstream.WriteInteger(this.itemCount);
   for(i=0; i<this.itemCount; i++){
    d = this.GetItem(i);  // Day object
    d.SaveToFile(Fstream);
  }
}
```

- I've assumed a file stream knows how to write a given object to itself. In YCPL this will probably be a little more complicated.
- YCPL will probably want telling that d is a Day object.

The joy of object inheritance

The code

```
    d = this.GetItem(i);  // Day object
```

is quite interesting because we haven't said the SortableList is dedicated to listing Day objects. What will certainly be the case is that YCPL's version of SortableList will have a method GetItem(i : integer) (or similar) which returns an *object*. Is d in our code an object? Yes.[112] In this case d inherits from the great grand daddy class of them all Object. So there is no problem using SortableList for storage.

When it comes to retrieving an object from a generic container in this way we might hit a snag. .GetItem() returns an object of class Object but really of course it is some fancy sub-class. How do we tell the program that the collection of bytes just handed to you is a certain class of object? (In our example a plain object doesn't know how to .SaveToFile() itself.) Some languages are clever enough or don't check but others need telling. This is called *casting*. When one type is shoe-horned into another (for example byte to 32-bit integer, or byte to character) or when an object has to be treated as some particular class you cast it. Rules apply! See your documentation.

---

110 Some languages can provide protection against saving a an object, updating the class definition then reading the 'obsolete' data.

111 The complete nesting of all objects within a containing object is called a Graph in tech-speak.

112 Technically it may be a sub-class of object - all objects will be.

Load from file

So easy! Just a mirror image of SaveToFile().

I've rolled (possibly wrongly - you decide) LoadFromFile() into the constructor with the file argument. (Remember that the no-argument constructor uses the parent's constructor.)

> This illustrates one of the central dilemmas of programming: To go for compact and efficient code or slightly more long-winded but explicit code.
>
> Extremes are bad.

```
constructor DayList(Fstream :  FileStream){
  SortableList();  // call parent constructor
  count = Fstream.ReadInteger();  // how many items follow
  for (i=0,i<count,i++){
    d = new Day(Fstream);  // read and construct
    AddItem(d);
  }
}
```

- AddItem() is an inherited method which 'in some way' adds an item to the list. the 'in some way' refers to the matter of sorting which the list *might* do as each item is added.
- Exactly how we read/write days is delegated to the Day class.

> Delegation is one of those things that people feel is a good idea but somehow can't let go immediate control. Obviously this applies to human relationships, but it also colours our judgement when splitting tasks between objects and their components. Perhaps there is some deep-down belief that smaller 'low status' components shouldn't be trusted with responsibility.

Display

What should we put into the Display() method of DayList and what should be delegated to Day? DayList really needs to control the arrangement of items as each Day can't really be expected to pry into it's neighbour's affairs. Also we might be setting some overall parameters for display both content (eg filtering, sorting) and presentation.

Let's establish a simple allocation of screen space which will work with StdOut as well as GUI.

- We could have a global header section for title and messages etc.
- A table headers section naming columns for tabulated output.
- A row by row item section for Days
- Assume we *don't* need anything at the base of the tabulation of days
- But we do want a global footer.

The old style of programming this would be to write all this code together in one function, but the more modern idiom is to write separate functions, these functions are of course methods of DaysList.

```
method Display(){
  DisplayHeader();
  DisplayColTitles();
  DisplayDays();
  DisplayFooter();
}
```

Why code like this? Surely all we are doing is putting off actually writing some useful code. Where is all this nest of method calling getting us?

- We are quite likely to want to fiddle around with the global header. So it makes sense to package it up in order that each time we tune it we can focus on only that task without having to worry about masses of other things in adjacent code.
- The same applies to the other methods. In particular DisplayDays() because we're probably going to start with a simple 'list everything' approach but refine it later.
- If we are going to print more than one page (perhaps on paper), or break the diary up into segments of weeks or months we can imagine a layout which goes:

```
Global header
Column titles
First block of dates
Column titles
Second block of dates
...etc...
Global footer
```

  For which we want to get at the column titles method as a separate entity.
- And we mustn't forget that the Display() method itself is very easy to understand.

Method visibility

The Display() method of DaysList that needs to be made available for other objects to use. In tech-speak we might say "The Display() method is *exposed* so it can be *invoked* by other objects." But do we really want for example DisplayColTitles() being invoked from outside? No, because (a) there is no need and (b) there might be interactions with the other methods which need to be carefully managed. For example in this example DisplayHeader() might set up a column layout which is necessary for DisplayColTitles() to work.

OO languages have ways to restrict the visibility of methods and data fields. There are tricky variations so you need to read the documentation for YCPL. `private` is a common key word used to indicate that the method (or field) can only be seen 'from within' the object. Java insists you use the `public` keyword to *allow* other objects to access methods and fields - defaulting to private.

You will probably find the *visibility specifiers* tricky to use to begin with. You can get particularly knotted when transferring from one language to another as their schemes have subtle variations on an already confusing base.

> I'm going to assume a StdOut interface and use the Fudge Print(foo) to send something to the display and PrintLn(foo) to print and go to a new line.

```
private method DisplayHeader(){
  PrintLn("Diary listing");
}
```

```
private method DisplayColTitles(){
  PrintLn("dd mmm yy  | Time | Event");
}

private method DisplayDays(){
  var d : Day;
  var i : integer;
  for(i=0;i<this.itemCount;i++){
    d = (Day)GetItem(i);
    d.Display();
  }
  // any finish code at the bottom of table goes here
}

private method DisplayFooter(){
  PrintLn("A=Add, S=Save, D=Delete, E=Edit, X=Exit");
}
```

- Notice that even without the comments that should be in the code you can see what each method is doing.

- `var d : Day;` is a bit like the variable allocation statements you'll probably be using in YCPL. It is generally a good idea, and many languages insist on it, to say up-front what variables you will be using in a function and what type they are. 99% of the time variables declared in this way are local to the function. This is a *Good Thing* because it avoids say the *same* `i` being used for multiple purposes. If `i` is declared in more than one function then each `i` is separate. See Scope¤.

A common frustration experienced by beginner programmers is not being able to access 'global' variables from within functions and methods. Don't be tempted to cheat if you have a cheat mechanism, it is really unlikely that you should be doing that. See Passing by reference in the glossary.

- The `(Day)` in `d = (Day)GetItem(i);` is an example of casting which we discussed above.

- In a GUI based application we'd be doing much the same thing, except having to do some more intricate manipulation of the drawing surface. If we were outputting HTML you can probably imagine we'd be doing much the same thing but wrapping the text up with tags.

- The user input hinted at in DisplayFooter() is something to think about. How are we going to get the user's commands? Which of our objects should be looking out for X being pressed? The Diary itself. The same with S. It looks like the Diary will have to field all user requests and if necessary pass them down to specific objects (say to edit a Day). At the moment we have made no allowances for any of these communications.... ...something we'll fix after polishing off Day.

Day class

```
class Day{
```

```
    field dd : integer;
    field mon : integer;
    field yyyy : integer;
    field hh : integer;
    field min : integer;
    field event : string;

    constructor Day(D:int,Mon:int,Y:int,H:int,Min:int,Event:string){
      this.dd = D;
      this.mon = Mon;
      this.yyyy = Y;
      this.hh = H;
      this.min = Min;
      this.event = Event;
    }

    constructor Day(D:int,Mon:string,Y:int,H:int,Min:int,Event:string){
      monthNum = Calendar.GetMonthNumber(Mon);
      Day(D,monthNum,Y,H,Min,Event);
    }

    method SaveToFile(Fs : FileStream){
      Fs.WriteInteger(this.dd);
      Fs.WriteInteger(this.mon);
      Fs.WriteInteger(this.yyyy);
      Fs.WriteInteger(this.hh);
      Fs.WriteInteger(this.min);
      Fs.WriteString(this.event);
    }

    method LoadFromFile(Fs : FileStream){
      Fs.ReadInteger(this.dd);
      Fs.ReadInteger(this.mon);
      Fs.ReadInteger(this.yyyy);
      Fs.ReadInteger(this.hh);
      Fs.ReadInteger(this.min);
      Fs.ReadString(this.event);
    }

    method Display(){
      // major fudge coming up
      Print(this.dd this.mon this.yyyy);
      Print(| this.hh : this.min);
      PrintLn(| this.event);
    }
  }
```

Lets pick the bones out of this:

- I've chosen to keep the parts of the date and time separate for ease of explanation. Normally I'd be using some ready-made date and/or time object.
- WCPGW with the first constructor? How about arguments such as the 77th of the 13th 93 at 7:77? Oops! No validation. Do we need validation? Yes - *always*. With special sharp-spiked, high-voltage validation for user inputs.

  Validating dates is a pain, fortunately most languages have date routines much like this one which build and validate a date from the parts. Would it be possible to hi-jack this in some way? Yes. See below.

Class methods

- Look carefully at what is going on here:
  ```
  monthNum = Calendar.GetMonthNumber(Mon);
  ```
  monthNum we'll assume is an integer being the result of the function? ...err... method? that follows. Hold on! What *exactly* is `Calendar`? From the naming conventions it appears to be a *class* not an *object*. How can that be? Surely the whole point of methods is that they work specifically on the fields of a given object. This isn't an object but a class which is an abstract collection of methods without any concrete instances.

  We can get away with this if the function doesn't need to access any object fields. In this case there will be a look up table which takes the name of a month and spits out a number eg 1 for January, 2 for February and so on. So `Calendar` isn't an object but a class. You need to look up ***class methods*** in the documentation for YCPL.

  Where does Calendar come from? Do we have to write yet another class? Hopefully YCPL will come with something similar already.
  - You may have to tell YCPL you'll be using Calendar.
  - A common class used without instances is Math (or similar) so you'll see things like `Math.pi` and `Math.logarithm()`.

- Now the answer to the previous question. Can we hi-jack some date validation? How about something like:
  ```
  ok = Calendar.IsValidDate(D,Mon,Y);
  if(ok==true){
    ok = Calendar.IsValidTime(H,Min);
  }
  ```

- What if there are no explicit date validation methods? We can still trap problems as follows:
  ```
  var dt : DateTime; // assumed existing built-in class
  ok = true;
  try
    dt = new DateTime(D,Mon,Y,H,Min);
  except
    ok = false;
  ```
  Here dt is always going to get thrown away, but we see if we can make it anyway.

- What do we do if the validation fails? We still have to construct a Day object even if the data in it is unreliable.
  - We could set the date to say 1/1/2000 and the time to 00:00. It is often a good idea to force illegal data values to something 'safe' or 'obviously wrong' or 'obviously wrong and safe'. *Whatever you do you MUST CLEARLY DOCUMENT IT in places other people are likely to notice.*
  - We could add some text to the event such as "ERROR". This is a human readable flag which doesn't destroy too much original data. Sometimes it isn't

very helpful to say just `Invalid input Redo from start`.[113] Instead displaying the offending data (preferably with something more informative than ERROR) allows the user to fathom out where they are going wrong. For example this program is set up for d-m-y as in the British convention. An American might first be baffled, then cross, that 3-16-2005 wasn't being accepted, so a hint explaining the style of input required is probably in order.

- We could add an `ok` field to the object. This seems like a good idea because for a very little overhead we can instantly tell if a certain Day is valid.
- We could raise an exception to be passed back up the calling chain until there was a suitable place to handle it. At the moment what we might do where is a bit hazy so we'll leave it.

- The exception handling in LoadFromFile() and SaveFromFile() appears to be missing. Do we need it? Probably not. Suppose in SaveToFile() there was a run-time error (eg disc full). This is what would happen:

  Red Alert! Stop normal processing! Try to deal with exception... No method specified to handle this exception in this method. Abort this method and report exception to caller. [DaysList object, SaveToFile method] Returned from Day method with exception. Red Alert! Stop normal processing! Try to deal with exception... No method specified to handle this exception in this method. Abort this method and report exception to caller. [Diary object, SaveToFile method] Returned from DaysList method with exception. Red Alert! Stop normal processing! Try to deal with exception... Aha! We are in a try-except block. Do the except bit: Print "Unable to write to file foo".

  The error is getting passed back up the calling chain and eventually handled. *In this case* we can't do much better than that by trying to handle the problem inside Day. For one thing we don't know the name of the file which is something that is going to be handy to have in the error message.

- The formatting of the Display() is, to say the least, suspect. I can get away with this as I'm coding in Fudge but you're using YCPL which will have stricter requirements. See printf¤ in the glossary.

Review

Although we've taken a long time to get here as we've been looking at issues along the way, our nest of objects didn't take much coding and is easy to understand at a first glance.

Tricky things such as how exceptions propagate up the calling stack to the most appropriate place to deal with them are provided almost for free.

Always the appropriate object is used to do the appropriate thing.

There's still work to do on the input and edit side of things but first wouldn't it be a good idea if we could test what we've got?

> One of the reasons programming is such a challenge is that there are so many ways to write a program. You're going to need dozens of 'sit down and work this out slowly and logically' exercises before everything slots into roughly the right place first time. Putting in this time is probably the hardest aspect of becoming a really good programmer - Stick with it because you'll end up being in the elite.

[113] ~~If you remember seeing this you're older than I thought! This used to be the classic~~ message given by BASIC programs when input couldn't be understood.

Testing(1)[114]

There are four main things we want to test at this stage. Adding new dates, display, loading and saving. Once we've got something that we know works (at least in a fashion) we can refine it. The obvious test procedure is to create a blank Diary, add some days, save, destroy the Diary object then create a new one from the file just saved and display it.

Are these tests going to be one-off props used just during this phase or might we need them at later stages? Let's keep them because having a test suite means we can thrash our application at each stage to make sure nothing gets broken as we add more things.

The tests are probably best initiated by the main program, but we may need to add test routines to the objects themselves. We'll see how we get on by adding a test function to the main program.

```
function Test(){
  try
    d = new Diary();  //
    // d.AddDay(1,2,2003,4,5,"Five past four on 1st Feb '03"); // any?
    // d.AddDay(1,2,2003,4,5,"1 Feb 03 04:05 - Duplicate"); // dups OK?
    // d.AddDay(31,12,2005,23,59,"Last minute of 2005"); // stretch args
    // d.AddDay(1,"Jan",2006,0,0,"First minute of 2006"); // month name?
    // d.AddDay(32,1,2006,0,0,"Should fail - bad day");
    // d.AddDay(0,1,2006,0,0,"Should fail - bad day");
    // d.AddDay(1,13,2006,0,0,"Should fail - bad month");
    // d.AddDay(1,0,2006,0,0,"Should fail - bad month");
    // d.AddDay(1,"Wednesday",2006,0,0,"Should fail - bad month");
    // d.AddDay(1,1,2006,-1,0,"Should fail - bad hour");
    // d.AddDay(1,1,2006,24,0,"Should fail - bad hour");
    // d.AddDay(1,1,2006,0,-1,"Should fail - bad minute");
    // d.AddDay(1,1,2006,0,60,"Should fail - bad minute");
    s = "Very long string for testing purposes. ";
    vls = s + s + s + s + s + s + s + s + s + s;
    // d.AddDay(1,1,2006,0,0,vls);  // what happens with 390 chars?
    // d.AddDay(1,1,2006,0,0,"");  // what happens with empty event?
  except
    PrintLn("An exception occurred in Test()");
```

Woah! Let's stop there for a moment:

- Notice all the AddDays are commented-out. This is because if the reverse was true and something unexpected happened we wouldn't know where. We can uncomment these a few at a time once we've established that we can actually create the Diary.
- And, ahem, we don't actually have an AddDay() method for Diary. The nearest we've got is a constructor for a Day object. We will join this up in a moment after tidying up the test routine.
- We're trying things we think should work and things we think should 'fail'. Recall that we decided not to propagate an exception if the validation failed but just to set an OK flag.

---

114 This is only an introduction to testing. The development process will be dealt with in more detail later.

- Would it be useful to have a method which tells us if there are any Days which are not OK? Quite possibly. For our testing purposes we want to know exactly how many, even if for everyday use all we want to know is if there are any at all. You won't be surprised that in the OO style of programming we will knock out two methods (DaysList seems the right place)

```
method AnyBadItems(){
  return (CountBadItems>0);  // returns a boolean
}
method CountBadItems(){
  badCount = 0;
  for(i=0;i<this.itemCount;i++){
    d = (Day)GetItem(i);
    if(d.ok==false){badCount++;}
  }
  return badCount;  // returns an integer
}
```

  And now a method in Diary to access the one in DaysList. (Notice the multiple dots.)

```
method CountBadItems(){
  return this.days.CountBadItems();
}
```

> This is quintessential OOP style. Lots of compact methods, each with it's purpose, with lots of interactions and lots of flexibility.

If we keep on going with our Test function what will it look like? A long list of method calls. Now that isn't OOP style at all. Style isn't everything but in this case we do well to stop and ask is this the way to do it? Shouldn't we split things up into segments where there aren't lots of competing issues floating about? There might be all sorts of strange things we haven't thought of coming out of the woodwork, so the simpler we keep each segment of testing the easier it will be to investigate and isolate.

Lets have a bit of a rewrite:

```
method TestAdding(d : Diary){
  PrintLn("Testing adding");
  try
     d.AddDay(1,2,2003,4,5,"Five past four on 1st Feb '03"); // any?
    ...etc...
  except
    PrintLn("An exception occurred in TestAdding()");
  PrintLn(d.days.itemCount + " Total items");  // fudge
  PrintLn(d.CountBadItems() + " Bad items");  // fudge
}
```

- The fudge alerts in the results are because at the very least, even if YCPL allows numbers and strings to be freely intermingled there may be a gotcha¤ if the + is applied in the numerical-plus context not the string-concatenation context. See printf¤ in the glossary.
- You should have lots of useful comments at the head of the code saying how to use the method, when to use it, what results to expect, why some items are commented out and only to be used for destructive testing.

Test() now looks like this:

```
method Test(){
  d = new Diary();
  TestAdding(d);
  TestSave(d);
```

(to be continued)

- Do we really need a separate TestSave() method? It's hardly worth the bother of a separate method just for the line `d.SaveToFile("DiaryTest.dat");` is it? Come on, you know the answer to this one by now...
  ...Yes we do because
  (a) Compartmentalising things is the safest and simplest way
  (b) To test saving we will need to fiddle with more than one file name. We need to test for broken filenames and URLs and gotchas¤. We need to do these so that we can validate our exception handling routines.

(Test() method continued)

```
  d2 = TestLoad();  // should return new Diary
  if(d==d2){
    PrintLn("PASS:Save-Load works");
  }else{
    PrintLn("FAIL:Loaded version differs from saved version");
  }
  TestDisplay(d2);
}
```

- TestLoad() must return a new Diary. (I wonder what happens if it doesn't - You can experiment using YCPL and see.)
- `d==d2` needs to be approached carefully.[115] d and d2 are not the same object but should have exactly the same data in them. That's what we're trying to test for to see if anything was lost by saving and loading.
- Notice that the test *always shows something*. Also the messages are clearly flagged with a pass/fail flag.

Review

If you are getting fed up with this long drawn-out process I can understand your feelings. The good news is:

- We are going to stop here for the time being.
- You are learning a huge amount very quickly. This would takes months on a computer science course.
- You are learning far more than the science of programming
  - The art of computing
  - The mental processes of dividing tasks into easy to grasp units
  - The slog of making a bullet-proof application

Almost anyone can cobble together code but few can be trusted to deliver anything more than a lash-up.

By the way, as promised, the AddDay() method for Diary.

```
method AddDay(D:int,Mon:int,Y:int,H:int,Min:int,Event:string){
  this.days.AddItem(new Day(D,Mon,Y,H,Min,Event));
}
method AddDay(D:int,Mon:string,Y:int,H:int,Min:int,Event:string){
  this.days.AddItem(new Day(D,Mon,Y,H,Min,Event));
}
```

- AddItem() is an assumed method inherited from SortedList.
- We could have created a method for DaysList called AddDay(), in fact that would

---

115 See page @@@ for discussion.

be the safest thing to do. The inherited AddItem() method lets us and anything we like which is a potential source of errors - late at night you'll be surprised at the sort of daft mistakes like this you can make.

Finish_the_diary_for_now
0. With your diary program
1. Finish coding in YCPL from the complete fudge listing in appendix @@@
2. Test to get working
- Disable tests by commenting them out
3. Test failure modes
4. Write a 'read-me' to go in your development directory
4.1 Overview of the project
4.1.1 Description of how the project came about
4.1.2 Status (Ready for use?)
4.1.3 Ownership
4.1.4 Where the design documents are or a top-down description
4.2 How to compile and run
4.3 Limitations and To-dos
4.4 Any other notes you might find useful if you returned to this in a few months time.
- See 6.
5. Backup the project
6. If possible give the backup to somebody else who knows about YCPL and see if they can get it to run on their system.
- This is a practical test of backup/restore *and*
- may illustrate some difficulties associated with transferring
- a code base between so-called 'identical' systems.

10. Databases

Programmers need to know about databases. Even if a lot of the time you find it convenient to store objects as ordinary files, and even if you work with esoteric data structures you still need to know when jumping on the database bus is the best answer.

In this chapter we'll do a lot of looking at how databases are constructed then right at the end in no time flat we'll do some real SQL practical.

Database are used to store data. No really?
- *Any* data.
- *Lots* of it.
- *Shared* between users, applications and systems.
- *Optimised* for speed of update and retrieval
- *Designed* for robustness
- *Supported* by many programming language interfaces
- *Used* by millions (Safety in numbers.)
- *Incorporating* sophisticated features
- *Providing* various security features

Basically databases are easy to use, efficient tools for practically any data storage task which requires lots of the same sort of record being managed in a central repository. (If your data structure is more naturally a tree then you may be better off using an alternative to off-the-shelf relational databases.)

There is a nice 'standard' programming language called SQL[116] and many tools to help you design, build and maintain them. We will be looking at SQL, how databases work, how you link your programs to them, how to design a structure for an application and briefly sketch the administrative aspects that you may want to think about. (@@@Check last sentence)

A short history of databases

In Ye Olden Days if you had large quantities of data it was on magnetic tapes. There were two drawbacks:
- You might have had 100 tapes but only 3 tape drives
- You would have to 'slowly' wind through a tape to get to a particular point.

Nowadays data is held on always-ready discs which can fetch any particular bit in a few thousandths of a second.

Suppose a firm was receiving payments for goods dispatched. The book work might go something like this:

1 Create a list of payments received. This list would have the invoice number on it

---

116 Pronounced Es-Que-El. Anyone who says different is a whelk-brained droid. (The significance of this remark will become apparent if you encounter it.)

which might have to have been looked up by hand.
2 Sort this list of payments into invoice number order
3 Run through the outstanding invoices tape looking for ones to pay. (Remember you can't keep jumping backwards and forwards through a tape it would take ages to wind between invoices at random.) If a good match then remove from the outstanding invoices tape[117] and add to the end of the paid invoices tape (which will get sorted - no mean feat - later) and add the customer number to the payment record. (If a bad match then create a reject log.)
4 Now sort the list of payments by customer number
5 Now run through the customer tape updating the account balances from the payments list.
6 Now sort the list of payments by account code
7 Now run the accounts tape updating by the amounts paid by the customers.

Nowadays we simply locate the invoice record, customer and account records at the same time. Mark the invoice paid, credit the customer and debit the account and update *as a single transaction*. We can do one single transaction when we feel like it without having to plan a complicated schedule of which tapes are on which drive. In the old days the payments might be done in a batch nightly or even twice a week, so allowing the data to get out of step with reality. Also what happens if say a tape drive crashes half way through step 5? Some transactions are 2/3rds complete and some 1/3rd complete - An utter nightmare. Nowadays if one part of the transaction fails it can be 'rolled-back' to the state before the problem automatically. Brilliant! And problems are reported straight away which makes them easier to deal with. And you can check the real credit remaining of a customer not a record which hasn't been updated with the goods that went out of the door this morning.

There was a lot of ingenuity used to optimise the processing of data like this. In fact "Data Processing" became a recognised, actually the mainstream, flavour of computing. ("Information Technology" hadn't been invented then.) Jobs were carefully batched and scheduled to make the maximum use of resources in primitive circumstances.[118] The art of Systems Analysis evolved with two aims : To apply the new technology in useful ways to business and to optimise use of computer time - with a join somewhere in the middle. With three tape drives and perhaps 96K memory words (that's for a mainframe) you weren't going to be doing much in the way of collating business information. Manager's and clerks would have shelves full of large listings which in the scheme of things it was easier to wade through than have a program written specially to extract certain items or calculate specific ratios.

Along came the magnetic disc which could be used for the storage of live business data. This data was way down the priorities for what should go on discs, but eventually it was discovered that a random access file opened up so many possibilities for

---

117 Which might mean (a) Marking it as to be deleted then (b) Some time later copying the tape this time leaving out the to be deleted ones.

118 Resources were pitifully small, very expensive, slow and often required 'engineering' down-time to achieve decent reliability. The PC on your desk is probably theoretically capable of doing all the work done by all the mainframes in London in 1970.

investigation and quickly cross referencing records that even though in 1980 disc storage cost more than $50 per megabyte it was worthwhile. One of the reasons the DP people used to sell the idea to management was that in future they would be able to access their data and extract summary information within hours of having a need. Tools called report writers evolved to make it easy to specify the records to be extracted from the random access files, sums to be done, sorting and presentation.

It was soon realised that the back-end, the collection of random access files needed to be well defined and structured with indexes. The definition of files and record layouts (and later how to cross-reference them) is called a ***database schema***.

In the early 1970s the ***relational database*** was created and made to work. This combined the back-end of fast, safe access to records with an easy to use front-end of commands for specifying the schema and working with data all based on a simple model of 'tables' of identically formatted records with links between them which could then be combined into a more useful compound 'record'. For example each record of the invoice table would contain a reference to the appropriate record in a customer table allowing all invoices for a customer to be extracted, or to display the customer's details when looking at an invoice on the screen. One of the key ideas is that information is not duplicated. In this example the invoice doesn't contain any customer details just the link that *relates* to them. This means that if for example a customer changes their name then there is no need to change anything with invoices, delivery notes, late payment demands, sales force records and so on.(@@@ Diagram?)

There are more possibilities for database building than SQL and you may have to work with legacy code¤ so we will start with random access files with fixed length records, work our way to SQL and get really stuck into all aspects of relational databases using it, then finally look at some alternatives.(@@@Check last clause)

Random access files

A plain file stream is really handy for piling all sorts of different things in so long as you don't need to get to a particular item without reading all the preceding ones. There are two methods we could use to get round this:

1 Index the start of items by how far into the file they are
2 Make all items the same size and calculate the offset into the file to find the n^{th} item by multiplying n by the size of each item.

Method 1 is complicated and full of snags. For example if you delete something you now have so many free bytes of space - What's the chance of the next thing you want to write being able to use up this space exactly? Ordinary file systems work like this by allocating and freeing space on disc in variable quantities and keeping a list of filenames with where they are physically located on the disc.

Method 2 is simple enough for everyday use. Providing you've decided how many bytes you're going to need in the worst case - the ***record length*** - you can work out where the record starts, fiddle with it, write a new one or mark it free to be recycled. It's about as easy as working with array elements, but significantly slower. The notable characteristic of such random access files is that all ***records*** are ***fixed length***. This means that the bits of data that are combined to make up a record are fixed length. If

you recall ne of the tests we did in the diary program was to see what happens when there was a very long string. This would have failed if we'd had to decide that, say, an event could only be a maximum of 30 characters. The alternative would have been to have allocated a huge (how huge?) space for the remote possibility of 300+ characters being in an event description.

The easy way to read a fixed length record is to define a type (sometimes called a record, sometimes a structure, possibly an object) with fixed-length *primitive types*[119] without any fields that repeat a variable number of times. The hard way is to work out the number of bytes into the record something will be and add that to the start of the record so you can read just that item. (It appears you're reading fewer bytes from the file this way but in practice disc reads are cached so asking a random access file to read say the integer at the very start will possibly result in the operating system physically reading dozens of records on the assumption that you are likely to want some more from the vicinity shortly.)

Record layouts, that is the order and type of the component the fields, need to be clearly documented. As they tend to be external to the program, possibly being shared by complementary programs it makes sense to have the documentation separate from the program code. (Hint: You need to be clear where this documentation lives in your filing system.)

When you're contemplating creating a record layout from scratch you often have to ask yourself about how information is to be represented. Suppose for sake of argument your application needed to know if a user's email address was valid. The obvious way is to have a boolean[120] flag using YCPLs boolean type.

- Are you sure you know how this will be physically written to the file? (See Gotchas below.)
- Might a single Y / N character be clearer and easier for other applications to interface with?

But who says there are only two possibilities? What about "Not yet known" and "Suspect" and "due for revalidation"? Even if you haven't planned on using this detail at the moment it looks like quite a good idea to allow yourself room for manoeuver later.

But why does it matter to get the record format right first time? Surely you can change the record layout as required? Only with difficulty - sometimes great difficulty. Even if this file format is used only by you for a single program you'll find yourself having to write a program that reads in the old format and copies in the new format to translate your data. Believe me it is a pain. Not only does everybody have to stop work, but then you have to do the translation and replace every program and test - complete with roll-back plan. (Not to mention making sure that you never *ever* use any old versions of the programs that might be

> Creating a record format is like cutting a bit of wire. Always allow a bit of slack.

---

119 Integers(specified precision), reals(specified precision), booleans and fixed length strings. Actually you can normally nest records for added convenience.

120 Boolean markers are often called 'flags' but a 'flag' doesn't necessarily have to have just two states although that is usually the case.

lurking around the place - Can you be sure you've upgraded every copy of every program.)[121] Then some of your customers don't upgrade and you're stuck with supporting two incompatible systems until they do.

Record gotchas

- Not all languages or operating systems store values using the same pattern of bits and bytes. We discussed this on page @@@. If other applications or operating systems are likely to use your random access files you need to be clear on a byte-for-byte basis how data is stored. Strings need special care in this respect.

- When counting up the length of all the fields take care to 'read the rules'. It is quite possible for a boolean to take two bytes![122] A fixed length string may require an additional byte or two to store the length as well as the characters themselves.[123]

- Due to a quirk of how processors operate there are advantages of 'aligning' fields within a record in multiples of two bytes. Most numeric fields are 2 or 4 bytes anyway but characters and booleans aren't. The effect of this can be for the internal representation of the record to be adjusted by padding odd length items if required with a blank byte. Suppose you've got a single character and boolean defined as fields in your record then if you used a hex editor¤ to look at a file written using this record specification you'd probably see each record taking four bytes.

  ```
  char data | blank     | bool data | blank
  ```

  or it might be laid out

  ```
  blank     | char data | blank     | bool data
  ```

  or

  ```
  blank     | char data |    16-bit boolean
  ```

  When what you (quite reasonably) expected to see was

  ```
  char data | bool data
  ```

  You can hit this problem when writing or reading somebody else's files.

  The answer is to use packed records. Look up "word-aligned" in your documentation.

Indexing

How many people do you know called "768"? In practice even house numbers are not numeric.[124] But in our random access file system we always need to get the nth record.

---

121 Old versions are practically impossible to eliminate.

122 Obviously a single boolean can't practically take less than one, but a bunch might be packed into just one byte.

123 That's assuming one character equals one byte. Unicode isn't!

124 As Sherlock Holmes could tell you.

The answer is usually an index (but could be a hash - see chapter @@@) For those that are interested we will deal a bit more with the inner workings of indexes in a later chapter.(@@@ Will we?) There is one fundamental concept, which you know already, and from which everything else flows: An index is two parts : The ***key*** and the (pointer to) the data. The structure and subtleties of the key are crucial to the usefulness of the index. Do you ever see a phone book where Ann Smith comes before Bertie Brown? To be useful we need the index *sorted* by something accessible and reliable.[125] There are all sorts of wrinkles with ordinary 'phone book' names.

- Variations: Mac -v- Mc
- Aliases: William -v- Bill
- Odd characters: D'eath, Dombey & Son Ltd.
- Changes : Miss Jones becomes Mrs Brown
- Titles
- Variable, sometimes excessive, length
- Unknown spelling: On the phone is it "Davis" or "Davies"

> My approach to people's names is ask them *both* what they want to be known as *and also* be certain what their surname and first name is. It's not unusual to get "Gill Smith" and "Dr S G Jones" as the same person.

If similar sounding names might give you grief, look up 'Soundex' which is a method for hashing. (Remember hashing from chapter @@@? In this case similar sounding words hash to the same thing.)

Another issue with names is case sensitivity. If you are not careful little `a` will come after big `Z`. This is a gotcha. Suppose your part numbers are normally all upper case and somehow `f123` gets into the list instead of `F123`. When people phone up and ask for part number "Ef-one-two-three" it doesn't appear on the screen because although it is in the index, it is right at the end, and computers work like humans do when searching indexes so to the computer `f123` doesn't appear in the F section and therefore 'doesn't exist'. Stop and think what the consequences of this might be. Now think of all the situations where indexes are used (ie everywhere - in airports, hospitals, banks - the list goes on.) Don't ever let this happen: *Always* use ***case insensitive*** indexing.

Review

I bet none of the documentation on how to write YCPL warns you about this gotcha. That means there are all those people out there writing programs who don't know they're a menace. Scary!

If nothing else, at least you will be competent. As I hinted at the start of the chapter,(@@@ did I) Data Processing, though vital and all around us, it is much less challenging than other aspects of computing. This may suit your temperament - or not. Worth remembering if you have career choices to make.

---

125 Don't go writing your own sort routines!

If you found the previous chapter a bit of an interesting eye-opener and whetted your appetite for things to come then Data Processing as a career will be a bit tame. On the other hand if it was 'hurting your head' don't worry those few pages took me years to learn. - Just step back a bit.

Indexing in practice

Unique keys

If you've ever had the same name as somebody in the same environment you won't need telling how confusing it can be if keys aren't unique. Quite simply, at some level of detail, record A has to be distinguishable from record B. In the file itself you could have 100 records all identical down to the last bit, but in the index you need to identify them distinctly.[126]

As we'll see soon a unique key is a prerequisite for items in a database.

So how do you ensure, without any doubt whatsoever, ever, that you will have a unique key for every item? For people we might consider

- surname+initials. Hmmm A clash looks pretty inevitable over time.
- What about surname+other names? Less likelihood of a clash but still a real possibility.
- What about surname+initials+date of birth. This is a common key used where date of birth is available but there is no guarantee of uniqueness.
- Wouldn't it be logical to hash *all the data* in the record to get a magic number and use that? Not very good because the data will be changing and so will the hash.
- What about picking a number at random (making sure we haven't used the number already) and sticking with it? This fulfills the guaranteed to be unique criterion but we'll never find the person again unless they can tell us their random number.
- What about using some existing unique identifier? How about national insurance number? This might just work for employees but what happens if we are taking on a temporary worker from another country where paperwork might take ages? (In Sweden every person is given a unique number at birth. Isn't that handy ... Unless you are a foreign tourist needing hospital treatment and the hospital needs your birth number.)[127] And you still need to know the number.

Umm...

The answer is to have multiple indexes. One which will always be unique is typically just an incrementing number which then sticks forever to that data record. This means that *if* you know the record ID number you can guarantee finding it. (And, *very importantly* - if you can't find it you know *for certain* it doesn't exist.) Then you have more general purpose ***secondary*** indexes based on how you want to get at the data in everyday situations. With people you'd probably use surname.

---

126 Although in which case you don't really need an index at all.

127 You'd be surprised how often existing unique identifiers aren't. But you only find this out a year or so afterwards. Ouch! Unfair! Swindle!

Having multiple indexes allows us to efficiently scan and process our data in different orders. A school might have a pupil index based alphabetically on surname and another on class+surname. The latter would make it easy for us to look at all the pupils in a particular class as a bunch - How useful would that be!

Indexes for speed

Specialised indexes[128] can be handy for quick access in particular ways. For example I wrote a database containing stocks and shares for a financial services firm. Each company might have a number of share types. We found that *most* of the time we could index using the first two characters of the company name and the first two of the share type combined into a short-cut code. At least we'd get only small handful to select from. This meant that four quick keystrokes took us 95% of the way. We needed the index to *speed up* access to the data to keep up with the keystrokes. Suppose we'd had to search all the records in the database from top to bottom - perhaps that would take ten seconds - what an age! But with an index we can get to the right place straight away in fractions of a second.

Indexing odds and ends

The important thing about an index is of course that it is an ordered list. We can process it from beginning to end and we can find an item (or be confident that it isn't in the list) quickly and with confidence. Does magic ability to jump to the record we want come for free?

Yes: In a way, all you have to do is command a database program to index for you and Hey-Presto it's done and you are unlikely to notice any overhead.

No: Every index needs to be maintained. Each time say a record is deleted the index needs to be adjusted - which if you think about it can never be done at *exactly* the same instant as the alteration to the data. Complicated juggling goes on inside indexes (we'll see this in a later chapter@@@???) which all requires time, memory and disc space. All of which means more things to go wrong. (One rare[129] problem is when an index becomes, for unknown reasons, (but the cause doesn't matter)[130] out of step with the data file.)

Sometimes you need an index to be accurate and ready for use all the time, but on others you are only requiring a particular order to be used once in a while. In the latter case it may be better not to have a live index and only create one from scratch for the times you need it. For example once a year you have a stock-check when a list of all the items in the stores printed out in the bin-shelf-aisle number is required. If you didn't

---

128 Terminology: I prefer to use 'indices' for pointers to array elements and 'indexes' for more than one index.

129 "Rare" is synonymous with "inevitable in the long run". (This concept is practically the essence of serious, professional programming. If by this stage of the book you're not happy with this then please chose something else.)

130 As a programmer you won't be worried about nested brackets. This is not a traditional use of brackets in English, but the logic is straightforward and sensible. Programmers are not fazed by this, in fact they appreciate it. However repeated nesting is probably a sign that you should be thinking of adjusting the structure of your document and perhaps re-ordering your thoughts. (By the way - Footnotes are a sort of block.)

need this order for anything else then the complication of a live index for the other 364 days of the year is not justified.

Hey, but I want to index my blog. How can I retrieve all the fault reports containing the word "smoke"? So? What's your problem? That's not a rhetorical question. Have a think what the issues are.

The indexes we've dealt with until now have been based on alpha-numeric sorting of key fields. Also there is exactly one index entry (per index) for each data record. There is no reason why we need to stick to this scheme. If we have a process that can look at text[131] to pick out words and collect those then we can make an index. Sometimes this is called a *concordance*. This is basically a list of keys with pointers to all the records that contain the word. This can get quite sophisticated, for example some full-text indexes can tell that "country" and "countries" are 'the same'.

Review

We are about to hit databases proper. Behind the scenes relational databases are random access files with indexes, so all of the above applies.

Whatever you do, do not try to write your own indexing routines - thousands of people have worked at doing the job well and dealt with the wrinkles, so don't try to re-invent the wheel.

We will now look at real databases in two stages. Firstly 'wrappers' for random access files that do the hard work and housekeeping for you. Then secondly the development of that to the fully fledged relational database model.

Tabular data files

In between the do it yourself random access file and the fully relational database are wrappers, which often hit the 'usefulness' spot, for random access files, allowing fields to be accessed by name, indexes to be maintained automatically, lots of interesting data types, cross references to be established (matching fields from different files), transactions (all files are updated or none), simple interfaces and integration with rapid development systems.

Terminology: You know what a random access file is :- A file with umpteen records of a fixed length. A *table* is a concept which is implemented using a random access file and index files controlled by a table *definition*. What in a random access file would be called fields are called *columns* and what in a random access file would be called a record is called a *row*. So for example employee's length of service is a field or column, and each employee is represented by one record or row.[132]

The long and short of it is that you are using a suite of functions rather than having to cobble together your own. These functions are more likely to be reliable, efficient, comprehensive and take into account all manner of wrinkles that you never thought of.

---

131 Or more sophisticated possibilities such as characteristics of images.

132 At this stage row/record and column/field are to all intents and purposes interchangeable.

Typically you define fields in some table definition utility, define one or more indexes similarly then you can use these inside your program. You might say

```
use the customer-by-sales-area index...
  go to first record
  while not 'end of customers file'{
    read current record into variable(s)
    process customer record
    fetch next record
  }
```

So much work, including *record locking*[133], goes on behind the scenes, making the useful bit of programming so much more productive, that tabular databases soon took over from raw random access files - where languages and operating systems could be made to work with them. The step-change is that *the database program now owns the files*, and you have to go via the database functions to get to your data. This is great in one respect because we can leave the chores to the database and take advantage of the work of specialist database programmers for speed, reliability and function. On the other hand we can't get at the data without using the program. This means that if I write a program using such a library of database functions, you are going to have to install it on your system. At the very least this might be a bind, but quite likely or expensive or impossible.[134] So tabular database files are not generally used for exchanging information, and there is the temptation to use light-weight alternatives for stand-alone applications.

Tabular database programs generally have ways to let you search for items by matching key and partial match. Partial match is especially useful for applications where a clerk needs to retrieve something by name - making them type the whole thing is a bit tedious and prone to spelling mistakes and the Davis/Davies trap. Also you're likely to be able, either directly or with a simple find-first/until-not-found/find-next procedure, to list all those matching a partial key. Typically you will be limited to the order of the results by the indexes you've defined. That is if you decided on a whim to sort on some aspect or aspects of your data which wasn't indexed you'd need to do the sort yourself.

> The basic functions create, retrieve, update and delete are sometimes known collectively as 'crud'.

A handy function used when creating a new record is to automatically provide a unique, integer field value. This is done by adding one each time a record is added. You can then use this as your unique record ID. The code goes something like:

```
AddNewRecord(tableFoo,arrayOfData);
id = LastAutoIncrement(tableFoo);
```

A typical scenario where you might use this is

```
if 'Create new record' button clicked
  create blank record with default values
  write blank record to table
  get record ID
```

---

133 Record locking will be dealt with very shortly.

134 I wrote a mapping program that was 400Kb itself plus 4Mb of database program. In the days of slow Internet connections this was a pain. WCPGW? My software 'upgrades' the users already installed database - See upgrade in glossary.

```
do EditRecord(ID) procedure
```

where you re-use the editing screen rather than write a brand new one.

Managing simultaneous access.

Think for a moment what would be the consequences of two people modifying the same record at the same time. Only the last one to write the record will get saved as it will overwrite any changes just made. User A reads record containing 10 items in stock. User B reads the same record and finds 10 items in stock. User A removes an item from stock and writes 9 back to the file. User B adds an item to stock and writes 11 back to the file. Result: Computer is out of synch with reality.

OK so we make a rule: "Only one user[135] can work on the stock file at any time". The file will be locked while somebody is working on it. This will certainly fix the issue just raised... but the storeman in stores won't be best pleased if he can only use the system on Mondays, Wednesdays and Fridays; the office clerks won't be happy if they can only do their work on Tuesdays and Thursdays; and everybody else with a need to work with the stores inventory won't need to bother coming in to work! So you think that's extreme - your alternative is ...?

The magic is *record locking* instead of *file locking*[136]. This is managed by the database program and is vital for all multi-user applications. There are various ways of dealing with multiple updates. The common one is to allow any number of readers but only one to edit. Since there tend to be a lot more reads than edits this works well in practice. Another scheme checks when writing back a record that no other data has changed since it was read. Transaction integrity gets more complicated when relevant data is held on a number of files which need to be kept synchronised.

What can you do in your program if when it requests a record for editing it has already been locked for editing by another user? That depends on the nature of the applications. Perhaps you try again after a short delay, perhaps you report an error and carry on, perhaps you report an error to a screen "another user has this item - please try later" or something similar. The important thing about record locking is to avoid using it 'just in case'. Try to keep the time from getting the lock to writing the update and freeing the lock as short as possible. NB. Make sure you free all locks if you encounter exceptions otherwise you can end up with a ghost lock. Most of the time locking is done behind the scenes in what may be a non-obvious way. Mostly you can use record locking without much worry but you will need to look at the documentation of your database to find out the details.

Combining tables

There are very few database applications that only have one table. An address book, possibly a diary, membership list and that's about it. A list of employees will be tied to a list of departments, a list of CDs will be tied to a list of artists, a list of parts will be

---

135 User doesn't need to a be a person - it could be the monthly accounting program - any program.

136 We haven't dealt with file locking. The operating system books-out each file to the programs requesting access. An application asks for permissions such as reading only or reading and writing. A typical rule is only one writer at a time.

tied to a list of suppliers, a list of invoices will be tied to a list of customers and a list of orders.

What do I mean by "tied"? It depends, but the basic issue is: "How can I combine the data I have about my CDs (date, title, genre etc) with the artists (Name, description, date of birth etc.) to give me useful information like: All CDs by a given artist and The artist's details for a given CD?

The brute force approach is to put the artist's name and details into all the CD records. You don't need me to point out that if the artist's details change we'd have to find all the CD records and change them too. Worse, what happens if the artist name changes, even by a fraction - Now all my CDs by `Ellington D.` are no relation to the artist `Duke Ellington`. The *whole point* of multiple tables is to avoid these issues. We try to avoid duplicating any data so that if it changes, we only have to change it once in one place, and then, however we reference it, we'll get the new version. This is done by cross-indexing records and the result is called a ***relational database***.

Let's look at the two example record definitions for CD and Artist.

```
CD                      ARTIST
  ID - Integer            ID - Integer
  Title - String          Name - String
  Artist - Integer    Description - String
```

The *relation* (as in relational database) is between CD.Artist and ARTIST.ID. (Dots used like this are a common convention to indicate *`table.field`*.) Let's see how the computer can achieve two sample tasks:

`All CDs by a given artist`

Get ID of selected ARTIST record. Look in CD table (preferably using an index if there is one) for all CD records with that ID in the ARTIST field.

`Artist's details for a given CD`

Get the Artist field from the CD record. Look this up in the guaranteed-to-be-unique primary index of the ARTIST table. Return the Description field.

Not impressed?[137] How about all artists for a given genre? We'll add in genre to the database as a table and as a field of CD.

```
GENRE               CD
  Genre - string      ID - Integer
                      Title - String
                      Artist - Integer // matches ARTIST.ID
                      Genre - String   // matches GENRE.Genre
```

GENRE.Genre will be a unique list. (Remember primary keys?) but many CDs will have duplicate CD.Genre fields. Is that good? Obviously CD.Genre can't be unique, but doesn't that ring a warning bell? What happens if we were to change `R&B` to `Rhythm and Blues?` Whoops!

```
GENRE               CD
  ID - Integer        ID - Integer
  Genre - string      Title - String
                      Artist - Integer // matches ARTIST.ID
                      Genre - Integer   // matches GENRE.ID
```

Now we can fiddle with GENRE.Genre as much as we like without affection the *data relationship*.

---

137 Actually I don't care whether you're impressed or not. It is one of the technological breakthroughs of the 20th century.

Now do something useful

Display CD information

Fetch the bits of the CD record. Fetch the ARTIST record as indexed by CD.Artist. Fetch the GENRE record as indexed by CD.Genre. Display the data from these three records.

List artists for a selected genre

Get GENRE.ID. Now find all the CD records with matching CD.Genre and put them into a working list. Sort these CD records by CD.Artist. (This won't be alphabetical!) Remove records with duplicate CD.Artist. Create a new working list by looking up each CD.Artist once to fetch ARTIST records. Sort the working list of ARTIST recods by name. (This will be alphabetical.)

I don't know about you, but rather than program all that lot I prefer to write "select Name from ARTIST where ARTIST.ID = @@@" and have the results delivered reliably and efficiently on a plate. That's what SQL will do for us in a moment.

Referential integrity

It is worth noting that when we were looking up CD.Genre we knew that all the CD.Genre data values would be one of the GENRE.ID values. But suppose we've got fed up with a genre and decide to delete it from the GENRE table. Oh dear! This condition is no longer valid because we could now have some orphan CD.Genre values which never appear in the GENRE table itself. Spotting these loose ends is one thing, doing something about them is another. Typically you write-in protections in your code to do things like insist on all CDs having their genre changed before allowing the GENRE record to be deleted. However you may also command your database to enforce referential integrity, or occasionally you'd run an exception report to discover 'can never happen'¤ cases.

4GL and all that

Fourth generation language. I expect there are computer science courses where this is defined and where students write essays on the subject. In the 1980s the term was bandied around by every database salesman as a mantra for the must-have in the march of progress.

Actually the hype and pseudo-definitions were wrapping an extremely useful step forward in technology. The three related advances were:

- Integrating databases with front-end application builders.
- Making databases easier to design and use
- Putting 'works out of the box', 'build-your-own', 'do something useful' databases on PCs.

Suddenly a new kid on the block could put together a working system in hours and days rather than months and years that would be required if a department had to join the queue for a

Looking back, the mid-80s was a time when the technology was becoming widely available to small businesses and departments of large firms who suddenly found there was a lot that was feasible even without networking. It opened up the possibilities to many more people who thought they'd have a go at either programming themselves or getting somebody to knock up an application for them. This resulted in the 'IT revolution' of the time which was more a feeling that one day soon we'd all have a screen on our desks at work so we'd better train all school children as computer scientists.

mainframe application. So PCs were purchased[138], the newly arrived LAN technology was tried out and bespoke systems proliferated. The 4GLs marked the change from that strange thing that happened in special centres 'data processing' to desk-top applications - 'here and now doing the things we want in our way'.

Application building

A list of records on the database displayed in a scrollable, searchable table on the screen is an obvious access tool. Everyone wants to lay out fields on a screen for editing and 'connect' them to tables. Tabular printouts and mail merge join the list of standard functions. Since CRUD is so universal it made sense to provide CRUDness with varying degrees of prettyfication customisation on the screen and flexibility behind the scenes.

There are many different approaches to marrying the front-end bodywork on the database chassis. Very roughly, the two ends of the spectrum are

- programming environment with screen and report building tools
- basic database layout plus 'easy-to-use' graphical design

Professional programmers prefer to get stuck in with the details of the data and so naturally prefer the first approach. The speed of building screens and reports is of course very welcome, but screens remain just one part of the application. The second approach is preferred by people who find themselves having to 'do something' and find they can get started painlessly by painting screens. Professional programmers wince at the resulting systems for reasons we'll shortly see. They prefer to put the wallpaper on *after* building the walls and building the walls *after* looking at the plans and drawing the plans *after* understanding what the building is for.

> Do not underestimate the importance of 'anybody can write an application'. Why should somebody pay you a lot of money to ask irrelevant questions, cast doubt on the way work is done at the moment - after all they should know (Err.. No but you can't tell them that directly) and take weeks to deliver a first draft when they could make it themselves in an evening at home on the PC. Discuss.

Review

The relational database is an incredibly useful model and modern technology makes it very straightforward to use. There are some important design skills to come shortly which relate to the way data is split across a number of tables.

If you were all fired-up on objects from the previous chapter you may be feeling a bit queasy about dispersing data which 'lives together' (such as all the information about a CD) across multiple tables. You have every right to be. The oil of objects and the water of relational databases don't seem to mix very well. Both are proven in practice and you'll probably be working with both a lot in the same line of code. It's not something to worry about now.

We've whisked through a very important revolution. Databases became something that everyone used. The pressure for 'results today' gave us application building tools of

---

138 Often by devious subterfuge to foil the DP department's monopolistic veto.

varying sorts which opened the doors to a new generation of application developers. Suddenly it was possible to knock up a demonstration system in a couple of days and finish it in a couple of weeks something unheard of in the mainframe world. Databases fuelled the need for LANs and computerised smaller businesses and departments.

We will return to rapid application building in a later chapter. For the time being we must stick to the skeleton before we can add the flesh. The next two sections discuss how to design a database schema from scratch and then look at the power of SQL.

Database design

The things that will set you apart from the rest of the world that thinks it can program are:

- Your ability to understand what needs to be done
- Your skill in being able to choose the right components and decide how they should be assembled.

We did some of this in the previous chapter where the components were objects and methods. Now we do the same thing with tables and relations.

The CD collection

Let's refresh our view of the CD database again.

> Purpose:
>
> > To list, add, sort, select, edit CD catalogue information

Notice that everything here is doing-something-with-data not data per se. As an overview it is fine but as an *application specification* it leaves something to be desired in detail. How can we find this detail?

- Method 1 (Obvious and useless)
  Get the user to write down what they want and sign it.
  WCPGW?
  - The user doesn't have the language to describe what they want in technical terms and they might use jargon related to their own trade which you don't properly understand.
  - The user takes lots of things for granted, perhaps assuming data is always complete, perhaps not realising the big difference between date in the style of `1999` and `12th Jan 1999`.
  - The user looks at a few CDs and finds the longest reference is 8 characters and gives this to you as a 'fact' when of course it is just the results of a survey.
  - The user wants to do things without collecting the necessary data. This isn't quite as daft as "I want a report of the items not in the database" but only just. Generating statistics is a minefield with unreliable and missing data making a mockery of any reliable results.
  - The user 'knows' about databases and designs it for you. This is a real swine because when they've made their minds up on the back of an envelope the scene is set for literally hours of 'my way's best'.[139]

---

139 I designed and built the CD collection database used in this chapter... ...The first CD I picked up to enter into it as test data turned out to be a double CD. Oops - Never thought of that. Actually it breaks the design because I can't develop the database further by adding a track table and relating it to a 'box' as tracks 'belong' to 'discs'. Sigh - If only I'd

And most importantly

- The user is looking at what they do now and try to replicate that with a computer system. No. No. No! That's like saying "We're in a hole - if only we had a bigger digger".

- Method 2 (*Systems analysis*)
  Talk to the user, read what they read, talk to their customers, talk to their colleagues, ask them about the calibre of staff, the cost of mistakes, what the competition is doing, what causes 90% of the hassle, what goes wrong, how standardisation would help, where could they usefully use more flexibility, where could they usefully use stricter standards. Look at and *count for yourself* the volume of data and transactions. Pester them about the details of the forms they process.[140] Find out who provides the data they work with. Find out who uses the information and communications they produce.

  Now think, drink[141], scribble, crystallise, re-visit and discuss.

  Now clarify your vision.

  Now present your vision.[142]

  WCPGW?
  Systems analysis takes time, is intrusive, doesn't produce instant results, offers alternatives not definites, opens up possibilities that require - deep breath - decision making.

> On a clinical assignment I was looking at the waffle that was being touted as so-called 'best practice' governance as background to the main task. The Real Programmer in me, disgusted at this tripe, invented a better system which transformed the motivation of the project from "how do we 'comply' with the rules" to "strewth what a shambles - we can see exactly what needs to be done now".

Bridging the gulf

So there's a tension between those with the need for instant gratification (them) and those with the vision of a new future (you); the tension between the carry-on-as-we-are types (them) and the what-a-can-of-worms (you); the tension between the 'don't even think of rocking the boat' brigade (them) and the champions of

---

done the systems analysis properly.

140 I arranged for monitoring of the quality of form filling in a clinical environment where practitioners were supposed to personally check four items on a form and sign it. They had a very clear formal clinical and clerical protocol to follow. How simple is that? Too complex for 60%! We got the error rate down in the end by insisting the forms were resubmitted until clerically correct. (We had no way of checking the *clinical* error rate. It could be done for a fraction of the cost of the mistakes that must be happening but that requires leadership - Sadly not available in the NHS.)

141 Particularly if you're used to following procedures and only getting things right first time a glass of beer or two can be a great liberator of the what-ifs, let's follow that thread a bit and sketching alternatives that are the ingredients of a thoroughly sound design.

142 Get the important person on board before ever letting your ideas go before a committee. Let 'important person' do the selling to their colleagues.

carefully considered change (you).

What to do?

- First get paid for your feasibility study/design/report. The more you've charged for it the more seriously the conclusions and recommendations will be taken. This isn't a joke but a serious observation. (Work back to how you split the systems analysis/consultancy from the implementation aspect. Impressive presentation is vital. A clear agreement of what you'll be doing and what it will cost up-front in writing coupled with an informal overview to whoever is pushing for change - focussing on what they see the benefit *to them* will be.)

- Get the *client* to expose problems ("The warehouse manager observed that few if any of the warehouse men could be relied on to read and write English"[143])

- Get the client to think they've made worthwhile contributions by saying so at the time and in your report even if you don't follow that path. "Barcoding as suggested by Henry initially looked like the ideal answer to solving the .... but further investigation unfortunately ..."

- Learn the technical lingo, 'share' their frustrations, be on their side. Sneak in some brainwashing of the advisability of getting it right first time. - "We don't want a repeat of..."

- If necessary, make up illustrative disaster scenarios and present them as they actually happened to you. - Your client then gets a warm feeling of superiority "We'd never be that stupid Ha Ha!". Be clear what point you're trying to make and how it relates to your almost superhuman talents and the clients own predicament.[144]

Overall you're looking for a reaction of 'Rome wasn't built in a day. When it comes it will be fantastic' rather than disappointment that you've raked over old ground, reopened old wounds and told them what they knew already.

> As with systems development, so with this book. The ordinary person thinks the job is to get to the top of the hill as quickly as possible. The real programmer and engineer understands there first needs to be a survey of the whole mountain so that you can then build the *best* route. (For values of 'best'.)

Review

How about that! Systems analysis in 1000 words. When you do the necessary reading up of the details don't forget the bit about bridging the gap which seems to get left out.

---

143 Believe it or not, this management problem was 'too difficult' to solve and I never went back to do the implementation phase.

144 All the anecdotes in this book are real.

Don't be afraid of facts and statistics, but always do a double reality check on the important ones: First for the numbers and second for where they lead in physical and monetary terms. Whenever you see "percentage" go over the logic of what it really means.[145]

To back up the doing-things-with-data you need to get a very good grip on what that data is. Now we can get started on the database, as opposed to the application, design.

CD database

We've already seen the tables listed with their fields and talked about how they are related. Now we need to make this a bit more technical.

- Overview : Three tables. CD linked to GENRE and linked to ARTIST. This means we can have one GENRE record maintained for many CDs. Also we can have a single ARTIST record maintained for multiple CDs.
- A CD will never link to more than one GENRE (In fact always exactly one.)[146]
- A CD could link to 0,1 or a number of ARTISTS.
- A CD has a given unique ID being the record company's catalogue number.[147]
- GENRE names are unique but not necessarily permanent. Therefore we need an indelible ID for the sole purpose of being the GENRE primary key.
- We probably want to list GENREs alphabetically so a secondary unique key using the name would be efficient.
- ARTIST names are almost certainly unique but not necessarily permanent. Therefore we need an indelible ID for the sole purpose of being the ARTIST primary key.
- We probably want to list ARTISTs alphabetically so a secondary unique key using the name would be efficient.
- Dates are probably only relevant to the nearest year and could be missing or vague. Integers will do the job - Date fields are not necessary (And may introduce unnecessary complications.)
- Field lengths for strings ... Systems analysis, ie sampling, will tell you.
- We may want to writes notes about an ARTIST and CD. Not sure how much but could easily be more than 1000 characters but probably not more than 5000

Schema

---

145 The quality manager of an electronics company asked me "how to analyse and present defects which were rising alarmingly". "We make 100 a month and faults are coming back to from customers in ever increasing numbers. This fault rate is up to 6 a month and increasing - that's 6%." 'Percentage' rings the reality check alarm bell! These were failures in service. What would be the 'percentage fault rate' next month if they only made 6 next month? Eventually light dawned.

146 Alert! "never" sounds like a sweeping statement in need of examination. Users are always giving you this story - be very very sceptical.

147 WCPGW? Your Aunt Augusta gives you a CD you've already got in your collection. Whether this matters or not depends on the exact purpose of your catalogue. Are you referencing actual bits of plastic as in a lending library, or the information on them for reference?

The relationships between the tables can be described as shown below:

```
GENRE  1 to many      CD many to many ARTIST  (spoken)
GENRE        1:n  CD       n:n          ARTIST  (typed)
GENRE        --<  CD       >-<          ARTIST  (sketched)
```

What about the fields and keys necessary to implement this?

GENRE.id - Unique key (*nonce*) (accessed from CD.genreId)
ARTIST.id - Unique key (nonce) (accessed from ...???)
CD.id - Unique key (uses real world data) (accessed from ...???)

A nonce is a bit of data without any real-world meaning. It could be random or sequential.[148] (The pattern of the key to your house is a nonce - its value is important but bears no relationship to anything else.)

Why are we having a problem joining CD to ARTIST when CD to GENRE isn't a problem? We can see if we take an example CD with multiple artists.

CD.name : `Country music grates - The worst ever C+W`.
CD.genreId : `15` (ie points to the Country and Western row of GENRE)
CD.artistId : `567` and `345` and `222` and `103` and ...

Whoa! In CD.genreId we could simply put in the appropriate integer but CD.artistId isn't a single a number - 'lots' would be a better description. The same problem arises when an artist appears on more than one CD - we need a list.

This problem arises when there's a *many-to-many* relationship. The way round it is to have a list in the middle, which of course this being a database, becomes a table in the middle. Here is how it works with a middle table called CDAR

```
CD              CDAR              ARTIST
.id     1:n     .cdId
                .arId      n:1    .id
```

We now have two 1:n relations instead of one n:n. We know how to deal with 1:n relations so we've solved the problem. Let's see how we can look up the artists on our worst country and western CD ever.

> We have the appropriate row from the CD table. Use the CD.id to look up the index to CDAR and fetch all rows from the CDAR table with a matching .cdId. With each of these (zero, one or many) rows, use the index to the ARTIST table to locate the ARTIST.id that matches .arId.

This gives us a list of artists. We can similarly work in the other direction. Given an ARTIST.id we find all the matching rows in CDAR and use the .cdId field data to reference the appropriate CD records.

Review

What we've just done is extremely important. It is a bore and a complication to add a table-in-the-middle to simplify many-to-many relations but absolutely necessary.

We've been developing a *database schema* which I use to mean a written (in text) description of a database. This is pretty much the same thing as a *database model*, a term I use for the concept and sketches of how tables relate.

---

148 Most, if not all, databases have a special 'autoincrementing' field type.

You now know how databases work and the theory of how to design them. It's a far cry from the glorified spreadsheet that most people think of. The power of the relational database is great for tackling large data models. If the design is done right in the first place (that's a very big 'if', you need a lot of practice[149]) then it will be able to grow and adapt as the real world changes. On the other hand a poor design will be awkward to implement in the first place and a nightmare to maintain.

Terminology refresher : Field and Column are the same thing. Record and row are the same thing. Row and column are the 'official' relational database terminology but most people use the terms completely interchangeably.

Now you've got the background you might want to do a bit more research on the technicalities of databases for yourself as I've left out several things that happen behind the scenes.

SQL

Nowadays most people will be using Structured Query Language, always referred to as SQL[150] to create their database, control access, perform enquiries and make updates. Although SQL is 'standard' each implementation has its quirks and features so you need to check up your documentation. Here is a flavour of SQL as used for enquiries:

> If you don't have a SQL database already installed then now is the time to download one. (Try MySQL - it is 'free', very good and used by millions.) Download and read the documentation too.
>
> There will be hand's on exercises in this section. The table definitions are given in appendix @@@

```
select id, name from CD where name like '%best of%' order by name

select count(id) from CD

select CD.name, CD.id, GENRE.genre
from CD join GENRE on GENRE.id=CD.genreID
where GENRE.genre = "Country and Western"

select * from ARTIST
where (  (ARTIST.birthYear >= 1960)
  and    (ARTIST.birthYear < 1970)
)
order by ARTIST.name

select birthYear,count(id) from ARTIST
group by birthYear
order by count(id) desc
```

---

149 In my opinion a database design is easy to sketch but difficult to perfect. This is one of those tasks that needs a days and nights to mature.

150 Pronounces Es-kew-Ell remember.

```
select name,year(now())-birthYear from ARTIST
where (deathYear IS NOT NULL)
order by name
```

> In my experience clear SQL guidance is difficult to find. Documentation has holes or is badly organised and tutorials expect you to be really enthusiastic.

- SQL is plain text, it doesn't usually worry about white space. It's often clearer to put each clause on a new line.
- Exact syntax may vary in detail. It may vary between versions of the same implementation.
- Most implementations of SQL are fairly casual about case. (You will often see documentation with the SQL keywords in capitals - That might be OK when highlighting the syntax in user documentation, but in use these keywords should be in the background because they're so obvious when you know about them that they don't need emphasis. On the other hand the main players are your tables and their fields. Personally I capitalise tables and use a lowercase start to field (ie column) names.)
- How quotes are handled can vary subtly between implementations and you may come across strange variants of single quotes. Make a study of your documentation to get their use absolutely clear.
- Extracting information always starts with `select` something `from` somewhere.
- Notice that in some of the examples fields are referred to in the TABLE.fieldName style and in others just as fieldName. The second form works providing there is no possibility of ambiguity....
- ...One way of avoiding ambiguity (which I use all the time) is to prefix all fieldnames with a couple of letters from the table name. For so for example all the fields in the CD table will be called `cd`*`Something`* - cdID, cdName, arName, geId. This means the you can't use the wrong field by mistake.
- `where` is a key word used at the start of the section that defines the criteria for selecting rows. A very common condition is when you want a single record and know it's unique id as in `... where cdID = 66 ...` It's a good idea to put brackets round the active bit of the where clause and sub-clauses for the reasons we discussed way back in chapter @@@
- In `like '%best of%'`, % is the wildcard character. (* is used for 'all fields').
- The `join` key word is the start of a definition of how we will connect another table to our spine query. The syntax is `join` *`newtable`* `on` *`somefield`* `=` *`newtable.somefield`*. *Joins* are the heart (and most puzzling bit) of relational database use. If you can see what's happening in these examples then that's all you need to know for the moment.
- There are various built-in statistical functions such as count(), avg(), stddev() which work on a column at a time.
- `order by` (which can be modified with `desc` for 'descending') sorts the results after extracting them from the tables. You can specify more than one column to sort on.
- `group by` aggregates rows into categories as given by the column specified. In the example we are counting the number of artists born in each year. (In this example the built-in function count() needs an expression so in this case we give it anything to keep it happy, but we might do more complex sums.
- The final example, which tells us how old artists were when they died, is a bit

tricky. select means here come the columns to retrieve. name is ARTIST.name, that's OK. Umm the next column is complicated, it appears this needs to be calculated as we go along. now() is the built-in function for current date and time. Take the year part of that using another built-in function. Now subtract ARTIST.birthYear and that's the result for this column/row.

- Null is an important 'value' in databases. It means "no data". In the final example we have a where clause which weeds out rows that don't have a year of death. Null isn't (usually) the same as 0 or an empty string so be very careful. You'll have to be especially careful when joining tables with nulls in the joining conditions.

At last we're doing something useful.

In just a moment we'll look at how to put data into the database and how to interface with a front-end or application program. But stop for a moment and think about the implications of having a 'standard' method for interfacing with a database.

It means that you can write your application and design your database independently of the database engine. (That's the theory anyway - There may be wrinkles, and of course you only find out about them at the time, but no need for major surgery.) You are not stuck with a particular vendor or having to use a particular manufacturer's overpriced and badly supported system. You can add your bit to a client's existing database - imagine your application that took postcode information to produce efficient delivery routes having to get data from the customer's existing database not to mention they'd have to manage another completely strange database system. It mean you can develop just the once for all the various SQL systems out there.

This is an example of a *layered* approach where your application layer sits 'on top of' a database layer. You can spend all your development time on your application layer and treat the database as a black box. Some people think ahead and build-in a bit of 'glue' to cope with adapting to different databases consisting of translation or special feature handling functions which can be tweaked according to the underlying database.

Hooking up to the engine

So far we've been talking about how databases work inside, and the type of queries we can do, but not looked at the practicalities.

You know that when the computer is switched off the data in the database is stored in files. You also know that you need to run a program to get at that data. There are two ways to run this program:

1 By incorporating a large library of database functions into your program. These will do the database work for you with a few calls from your code.

The disadvantages of this are

- Your program *is the database program*. This means it is difficult for it to interact with any other application and share data. If you are sharing data then all programs will need to change version numbers at the same time.
- Your application is swollen to many times it's non-database size by the overhead of all the database code. Each program has a copy of the baggage.
- Customers have to use your make of database or you have to get a copy of their database program to test - which may be expensive - and you have to keep

upgrading to the latest version of the database.[151]

2 By running a separate database engine or database server (DBMS : Database Management System) that can inter-operate with your applications.

The advantages of this are
- Each of your programs need much less database code
- The database is available for other existing and future applications to use.
- The customer is easily persuaded that your program is 'standard' and will be no bother to look after and work with.
- Your customer's database strategy is not dictated by your application.

The DBMS might run on the same computer as your application or on another one (or even be distributed amongst a number of servers.)

151 Lookup ODBC for steps to mitigate this issue.

Technical background - a diversion

You may recall a long time ago I roughly indicated that your text gets turned into something more suitable for the computer to operate with. We need to look at this a bit more to explain linking of code libraries. The short story is that there are ready-to use libraries that you can stitch into your program.

What this means is you can

- weld outside code tightly into your program to give a monolithic package
- or rely on ready-to-run utility libraries being available on the customer's computer
- or get your program to interact with a serving program explicitly set up on the customer's[152] machine.

Database administration

Unless you have very specific needs you will be using an interface to a DBMS (Database Management System) You will use manual utility programs or functions to create the empty database, allocate access rights, import and export bulk data, and generally poke around. We'll call this bit 'administration'. Who does the administration? An administrator of course. Err... Of course? Occasionally a customer will have somebody who is given the job of database administration. Out of these few, some smaller number will be clued-up about the value of the data and the importance of looking after it and have the responsible attitude and skills to do real administration. For the rest they 'do a backup' but that's their limit and they don't know how reliable it is.

The actual or anticipated competence of the customer's database administration is very important to the programmer. If staff can't be relied on then you need to build robustness and fail-safeness into your program.

You'll be using database administration tools quite a bit. While you're doing so consider what the implications for users would be, how you will document any actions that may have to take, perhaps once in a blue moon. How will you make it easy for them? How can you get across the consequences of carelessness or deviousness? Whatever you do, be sure to document privately the things that matter in case the plot is lost.[153]

---

152 I'm using 'customer' not 'client' because client has a specific computery meaning which could be confusing in this context. When dealing with customers you must be careful not to use words with technical and ordinary connotations unless everyone is absolutely clear about use. (Even then avoid if possible - somebody down the line will get the wrong end of the stick with all sorts of consequences.)

153 From time to time I ask my customers to give me a backup copy of their data to make sure (a) the backups work and (b) I can still get their systems working from scratch in case of disaster recovery. Also it is handy to have reasonably up to date data if you're closely involved with support.

Administration tasks

- Making sure the file system is suitable, secure and data is backed-up. A whole lot of memory is often a good thing on a machine used to host a database. Make a record of the essential configuration points so you can re-create quickly in an emergency.
- Take a long look at permissions and the practicalities of access control.

At some point you'll need to consider which functions will be packaged in an idiot-proof utility and which parts the user will have to fend for themselves.[154]

If an organisation loses it's data or suffers a security breach and you had a hand in setting things up then some nasty person might look to you for financial compensation! There's more to programming than coding.

Building the database

This is the easy part. You have a schema sketched or typed. The DBMS will have an administration tool which lets you create tables, allocate key fields to indexes and report the database structure.

There are graphical tools which some people prefer and can be handy if you like pretty documentation. These might be fully integrated with the DBMS or for documentation only or capable of working cooperatively. At the early design stage I prefer pencil and paper - it's a lot quicker and the emphasis is on the concept not the presentation - you can tidy that up later.[155]

A lot of pretty documentation is overrated. Clarity and accuracy trump presentation any day. Remember I said you needed a really cast-iron filing system.

One useful purpose of writing documentation, especially the first draft of the user guide, is clarifying areas which you hadn't really looked at hard enough in the first place and revisiting parts of the system you hadn't been working on for a while and looking at them in a new light.

Database operation

Sorry. That's all there is to creating a database. The hard work was the design. Because you don't yet know how to write a program to make use of the database we'll have a play using the DBMS administration tool to simulate the sorts of activities. (To do this you'll need to create the CD database as described in appendix @@@ then return here.)

The code shown here is to be typed into the SQL window or prompt.

Genres

Lets add some genres. Here is the SQL. I expect you can see what's happening: The

---

154 Remember Real Programmers carry notebooks. Here is one self-protection reason. At some early stage you *will* record a meeting where it is agreed who will carry the can for security and their name will go an all security related documents, procedures etc.

155 I also have a two screen installation (sharing one keyboard and mouse) where I can have documentation displays on one for instant reference while working on the main screen. Highly recommended.

first line sets geId to 0 and geGenre to Classical Symphony. And so on. Except something fishy is going on with geID.

```
INSERT INTO genre VALUES (0, 'Classical Symphony');
INSERT INTO genre VALUES (0, 'Blues');
INSERT INTO genre VALUES (0, 'Jazz');
```
[156]

Now have a look at the data to see what's happened to the geIDs. You should see them as 1,2 and 3. That's the autoincrement function at work. If you try

```
INSERT INTO genre VALUES (2, 'Folk');
```

you should get an error as you've deliberately specified a value for a unique key which already exists.

CDs

```
INSERT INTO cd ( cdId , cdTitle , cdSubTitle , cdGeId , cdNotes )
VALUES ( 'CD 53002', 'Django Reinhardt',
'Giants of jazz - The gipsy genius', '3', '24 tracks' );
```

This is similar but the field names have been explicitly listed before the values.

```
INSERT INTO cd ( cdId , cdTitle , cdSubTitle , cdGeId , cdNotes )
VALUES ( 'CACD 1013', 'Beethoven',
'Viol concerto (D op61) Romances 1(G op40),2(F op50)',
'1', 'Using period instruments' );
```

Query?

The following query should report two rows and two columns.

```
SELECT cdTitle, geGenre
FROM cd, genre
WHERE geID = cdGeID
```

Hurrah! A relational database in action. (Just to remind you that the code you had to write to provide this functionality amounts to a couple of dozen lines.)

```
SELECT cdTitle, geGenre
FROM cd
LEFT JOIN genre ON geID = cdGeID
```

This should produce exactly the same results.

The first query was a bit of a shortcut. The essential relation geID = cdGeID is the same it is just that WHERE can guess that we would write the join out longhand if we could remember the correct JOIN syntax.

- Notice that = sign. It's a 'same as' not an assignment.
- Can you see the usefulness of prefixing fields to identify which table they come from
- JOINs come in various flavours that unfortunately combine subtle differences with complexity - and no two DBMSs seem to implement exactly the same set.
  - Have a look at your DBMSs documentation and put a bookmark in the place as you'll need to go back to it quite a few times. There are wrinkles and gotchas.
  - Track down a book on SQL and see if you can make sense of the examples.
  - @@@ gourmets

Artists

@@@ add artists

---

156 My DBMS (MySQL) or perhaps the administration utility (phpMyAdmin) decided for itself to convert my choice of upper case table names to all lower case. Grrr.

@@@ connect the join
@@@ query

Review

Modern databases are fast and easy to use. I suggest you rush down the library and have a look at the database books there to browse lots of examples. You'll come across some syntax differences from your particular DBMS, don't worry at this stage you're in need of quantity in order to get a feel for design and style of database operation.

In a couple of chapter's time we'll look at how to build applications using a database as a back-end. (@@@ Will we???) We've had a taste of translating the user's needs into a data model - there's more - but the good news is by working at this high level you can create comprehensive systems very quickly.

The next chapter is going to be a complete contrast. It's time to sharpen up YCPL coding skills.

11. User interfaces

Gotcha - Users are assumed to be people but might be programs.

In one way or another a program exists to serve a need. That need might be a person looking up diary dates or a program asking if any email is waiting to be delivered. Programs that run in the background and wait for requests are called ***servers***. Programs that are called to do something they specialise in as required are called ***utilities***. In both cases there are typically two types of interface: One for administration and configuration and the other for doing whatever it is the program does. *The distinction can become blurred. We will come back to servers and utilities later.*(@@@will we?)

Where people interact directly with programs we can use the old style 'terminal' type of a typed conversation (called a command line interface - CLI) or a WIMP (Windows Icon, Mouse Pointer) interface, called a Graphical User Interface or ***GUI*** for short. You need to be able to work with both styles and select the most suitable on a case by case basis.

Comparative example

On a *nix system there is a command called `ls`. Windows uses the `dir` command to do the same thing. Both provide a list of files in a directory.

- The result is plain text
- The operation can be tweaked using cryptic flags on the command line
- Because they use StdIn and StdOut[157] they can be easily called by any other program that uses StdOut and the results passed to any other program that uses StdIn. This means the results are not just for human consumption - for example they might go to a backing-up program for further processing.

Or you could use a graphical file manager.

- Lots of scope for prettification
- Settings controlled by menus and mouse drags
- Large lists can be navigated easily and sorted in-situ
- Mouse clicks can make interesting things happen instantly

If I am on the phone to you and I want to give you precise instructions about copying or moving files it is lot safer to do it using the CLI. I can say things like "type d e l star dot t m p" with some confidence that you've followed my instruction.[158]

> Myth: GUIs are 'obvious'. No they are not. What they are though is relatively standard so that once someone has learnt how one application/OS operates then they are able to guess how others work.

---

157 See page xxx in chapter xxx @@@

158 I speak from long experience. Moving and copying files in a GUI over the phone is a nightmare.

Command line basics

Programs that use the command line may just use it for initial configuration in which case it is almost like a function where you give it a set of arguments following the name. YCPL will probably have a method of being able to read these parameters. The convention is that arguments are separated by spaces.[159]

Generally there are two types of argument: Flags which tend to start with a hyphen in *nix-land or a slash in Windows-land and variable parameters such as file names. It is worth making a quick study of the style of parameter syntax being used by your target audience. You could use key words as flags but often these can be confused with variable parameters.

Commands object

What you are normally given to work with inside your program is an array of strings and a count of the number of items in the array. Over to you...
...You're in charge, you've seen the style of parameters used...
...decode, validate, guess what the commands refer to in the real world and check for consistency.

- Best done in one place, inside an object perhaps.
- Do you care about case? - Hopefully not - It really is annoying when `-x` is acceptable but `-X` is rejected. (And difficult to document as x and X look pretty much the same.)
- What do you do if commands are not understood or consistent?
- Are you sure whitespace¤ and control characters are being removed before they get to you?
- If there are variable parameters with whitespace in them
  - Do you insist they get put after the flags? (Perhaps no need - just a thought.)
  - Do you need whitespace as part of the parameter. eg File names with spaces or text with tabs? Special measures will be required.
  - How will you deal with parameters in the form *key=value*? The = might have whitespace around it or none, on either or both sides.
- How flexible are you in guessing what the user means. For example if you are given just a filename where will you look for it? Just the current directory or in the search path or in some special locations?[160]
- How flexible are you with flags? Do you let the user use all of `/v`, `/V`, `-v` and `/verbose` for example?

The first of these points is not only useful but important. It is a layer that conditions and buffers input. It enforces rules and decodes text into logic and acceptable parameters. We will see this again in GUIs where a good grip of events is even more important.

---

159 Hey! Isn't that a very good reason never to use spaces in file or folder names. On my computer, courtesy of Microsoft there is a very important directory called "Program files". I can't use the fundamental command cd to change directory without specially putting it in quotes. Ta very much - not.

160 Don't forget to document this clearly. Sometimes this flexibility is a bad idea - for example if there might be confusion between exactly which of a number of identically named files in different directories or fall-back options are used.

Command line decode example

Do you recall the diary we were writing back in chapter @@@? The parameters we might want are the name of a data file, a date and the maximum number of lines to show on the screen. All should be optional.

> The UI *includes the user guide* and reference documentation. It is not just the bit inside your program that offers options and results and processes input. You might throw in on-line help and hints as well.

Defining the interface

Let's define some command line syntax then write a handler. How about starting with maximum number of items to show on a screen? This might appear in the user guide as:

```
Max no lines to show :
    l=nn
```

Is that as clear as we can make it? If for the sake of 30 seconds thought we can avoid a single support call we have come out ahead.

Is that as simple as we can make it. We could 'cheat', and so avoid the whitespace-equals issues by saying that if we come across a number in the range 2 to 99 we will assume it is the maximum number of lines. We can get away with this if won't clash with any other arguments. In this case the date might be a problem. There are two ways out of this: Relying on position or strict control of whitespace.

```
Date to start from:
    dd-mm-yy[161]
```

We need to declare our stance on converting two digit years to four digits. Perhaps we should silently allow four digit years. If so what limits do we set? All this is part of the UI so has to be available for reference.

```
Filename
    filename
```

If the user specifies "myDiary.dat" do we need to preserve case? (Yes on *nix, except we might 'be helpful' and always crash case¤ so it doesn't matter.). If the user specifies "mydiary" do we assume they mean "mydiary.dat"? Once again the details of the behaviour and the defaults need to be explained.

The final bit of command interface definition needs to be messages that get returned to the user. We might chose to be silent except for outright errors. We could display a handy help message giving usage instructions.

Design and build the CLI object

Overview

The object will be created from an array of parameters, we interpret and validate, display any error messages then leave it in existence in order to be interrogated by the remainder of the program.

Notes

We would normally put extensive documentation in the code as the definitive

---

161 Note this is in UK date standard. You'd mention this in the user guide and other places of course. You might also offer an additional flag say -u for US-style date order and re-interpret the date parameter as mm-dd-yy.

version of information for the user.[162]

Fudge[163]

```
object CLI
  field maxLines : int;
  field startDate : date;
  field fileName : string;
  field ok : boolean;

constructor CLI(){
  // sets up default values
  this.maxLines = 20;
  this.startDate = getTodaysDate();  // assume system function
  this.fileName = "diary.dat";
  this.ok = true;
}

constructor CLI(Parameters:array of strings,ParameterCount:int){
  // the useful constructor that takes command line parameters
  CLI();   // call default constructor
  // now interpret the parameters
  // NB often parameter 0 is the name of the program itself
  // so the first actual parameter is 1.  See YCPL docs.
  // Assume here possible [0],[1] and [2] for up to 3 parameters.
  ppointer = ParameterCount-1;    // last one
  while(ppointer>=0){
    pstring = Parameters[ppointer];
    if(pstring.Contains('-')){
      ok = ok and DecodeDate(pstring);
    }else{
      if(pstring.LooksLikeInteger()){
        ok = ok and DecodeMaxLines(pstring);
      }else{
        ok = ok and DecodeFileName(pstring);
      }
    }
    ppointer++;
  }
  if(not ok){
    DisplayUsageInstructions();
  }
}

function DecodeDate(DateString:string){
  // interpret dd-mm-yy
  // return a boolean false if doesn't look like a date
  d = DayStringToDate(DateString); // Fudge.
  if(d.IsRealDate()){
    this.startDate = d;
    result = true;
```

---

[162] You might just be able to get this converted into user-friendly format automatically by standard documentation tools.

[163] Fudge uses all sorts of imaginary 'system' functions such as Print(), GetTodaysDate() and so on. YCPL will have something along these lines but may be used differently. Note also that types are rather vague which may be contrary to YCPLs philosophy.

```
      }else{
        result = false;
      }
    }

    function DecodeMaxLines(LinesString:string){
      // interpret max number of lines
      // note (to show alternative UI technique) we display error
      // as well as returning false
      i = StringToInteger(LinesString);  // Fudge
      if((i>1) and (i<100)){
        this.maxLines = i;
        result = true;
      }else{
        Print("ERROR: " . LinesString . " is out of range.
              Default max no lines used");
        result = true;  // what!!!  This is naughty but we've just
        // ignored the user's request and carry on with the default
      }

      function DecodeFileName(FileString:string){
        // similar to DecodeDate() or DecodeMaxLines
        // code left as exercise for reader
      }

      function DisplayUsageInstructions(){
        Print("Usage ... etc etc ...");  // StdOut
      }


    // ---- get at results of decoding ----
      function DoWeHaveUsableCommands(){ result = this.ok;}
      function GetMaxLines(){result = this.maxLines;}
      function GetStartDate(){result = this.startDate;}
      function GetFileName(){result = this.fileName;}
    }
```

This code is now in a black box and we shouldn't have to worry about it or have to wade through it while dealing with the rest of the code.

```
cli = new CLI(params,paramcount);  // args from system
if(not cli.DoWeHaveUsableCommands()){Die();}  // stop!
...
loadfile(cli.GetFileName())  // pass one part of cli
...
display(cli,...);  // pass whole cli object
```

This method beats occasionally enquiring what a parameter is as you require it in the code hands down.

As I've already mentioned, having a buffer layer between the inputs (and possibly outputs) and the program logic applies to GUIs as well. Because many GUI paradigms make it easy to skip this you may not realise the possibility of using it where appropriate.

Review

* uguide @@@

* careful design
* parsing - skills
* package cli logic

Where am I?

There are some programs that accept any commands at any stage (or only use them at the launch stage anyway). What about a program that has say a Calibration mode and a run mode? The user needs this information clearly but unobtrusively displayed to them. A typical method is to adapt the prompt by putting a label or status information on it. For example : `configuration>`

Good UI design depends on being able to group together functions so that the user is focussing on one activity with all they need for that activity to hand but no superfluous distractions and no immediate access to unrelated actions. This applies to CLI and GUI.[164]

@@@mode

More feedback

What should you do if your program takes a long time (for values of long time) before appearing to do anything? It's a really good idea to provide progress indication of some sort. Sometimes you want the opposite, particularly if your utility is being managed by another program and it doesn't want a load of progress messages mixed up with real results. - Provide a flag to switch off information messages. Sometimes you want to know what's going on behind the scenes so you may provide a flag to switch on extra messages or logging to a file for later analysis.

'Is something happening?' applies equally to CLI and GUI. With programs running locally (or where the action happens locally) it is normally very easy, and expected, to show 'something's going on' with a progress bar or flickering lights. (See Time left in glossary for handy tip.) With CLI you can end up madly scrolling the screen if you don't moderate your progress feedback. A major problem exists with web page fetches, remote processing controlled using web pages and uploads using web pages to submit large files. This is because the HTTP protocol[165] is whole page or nothing. From a UI point of view you find users think nothing is happening after clicking on 'process my order' or 'upload my file' and click again with all sorts of consequences.[166]

Completion status

---

164 And books. One of the distractions I've been trying to keep you away from is unnecessary detail and trying to jump to the end. Bit by bit we'll be finishing off the whole picture, so if your favourite bit of essential programming technology hasn't been covered yet that's probably deliberate.

165 More on protocols in a later chapter.@@@???

166 You have to deal with repeat web page submissions as a matter of course. See Idempotent¤.

If your utility fails in some way when being used by another program how will it flag that error? (Imagine your program checks for changed files and provides a list to a program that will back them up. If it keeps failing then the backing up program is quite happy and reports 'nothing to do - all backed up'. YCPL and operating system may cooperate on this matter or you might have to find another way to flag ok/fail.

Graphical UI development

We are going to develop a search and replace program to investigate some of the issues involved when creating graphical interfaces. Everyone join in! - This is in Javascript.[167]

- Create a file (in your well organised filing system) called `SrchRepl1.htm` with the following code.
- As you type in each line ask "What is this line doing? Why is it formatted like it is? What does it represent?" [168] (RegExp will be discussed in a little while.)

```
<html>
<head>
  <title>Search and replace 1</title>

  <script LANGUAGE="JavaScript">
    function DoSearch(){
      // what happens when we command s+r with
      // the go button
      document.form1.area.value = DoReplace(
        document.form1.srchStr.value,
        document.form1.replStr.value,
        document.form1.area.value);
    }

    function DoReplace(SearchFor,ReplaceBy,Body){
      // utility function to wrap regexp
      // search and replace
      rexp = new RegExp(SearchFor,"gi"); //g=all i=ignore case
      return Body.replace(rexp,ReplaceBy)
    }

  </script>

</head>
<body>
  <form name=form1>
  <input name=goBut   type=button value="go" OnClick='DoSearch()'>
  <input name=srchStr type=text   size=10>
  <input name=replStr type=text   size=10>
  <textarea name=area rows=3 cols=40>
There is a young carver from Cressing
With curious habits of dressing
He always wears shorts
Of various sorts
```

---

167 If you're new to it there is a Get started with Javascript appendix.

168 When creating your own code "WCPGW" is a handy mantra.

```
But never a thong - What a blessing</textarea>
  </form>
</body>
</html>
```

When you point your browser at SrchRepl1.htm you should see a button, two empty text boxes and part of a limerick. These graphical elements are called *controls*. Just for fun put text to search for in the first box, something to replace it with in the second and click the Go button. (If nothing happens add a line `alert("here");` to the Javascript code and move it down until it stops displaying a pop-up. NB You also need Javascript enabled in your browser.)

You've probably thought of some improvements to the UI. We'll have a look at those in a moment, but first let's look at why a mouse click on a button does anything.

Events

When your phone rings you pick it up. Or perhaps the answering machine kicks in. Or perhaps it rings until the caller gives up. In the first two cases an event causes an action. In the last case the event happens but nothing comes of it. Clicking a mouse button is an event that the operating system recognises. What you have to do is tell the operating system what you want to happen as a result. Unless you do this the event will be thrown away.

The raw operating system events are mostly generated by the keyboard and mouse. (There are other OS events such as connecting to the network, ticks of the system clock and 'closing down!') The components in YCPL probably have more sophisticated events. For example telling when a control gets or loses *input focus* or when it is being created. (With a bit of simplification) input focus is the place where keystrokes go to.

In `SrchRepl1.htm` we've created a button and told the browser (which is acting as a surrogate operating system) that if there are any clicks while over the area of screen painted to look like a pushbutton to call a function called DoSearch(). That's what the `OnClick=DoSearch()` is. Click is an *event* (as defined by Javascript and implemented by the browser with cooperation from the operating system) OnClick() is a method of the button object which by default does nothing. We can assign a function known as an *event handler* to the OnClick() method to act on the Click event.

> *`variable = function`* might *not* mean Set the value of variable to the RESULT of function. I admit it always has done until now, so this bit may make you feel a bit queasy. Suppose variable has a type of 'function'. Then you can say variable foo is a reference to function bar() by coding `foo = bar()`. Now should you wish to run the function bar() you could instead run foo. OnClick is normally set to a do-nothing function, but in our program we've given it something to do by referring it to DoSearch().

> Reminder: OnClick is a variable which has the type 'eventhandler' so when you code OnClick=foo() you are setting things up for future use not actually executing the function foo().

There are more events than Click and different controls respond to different events. For

example a timer control[169] doesn't care about clicks (It won't even show up on the screen) but it will have an OnTimeUp event handler. For a start you need to look at YCPLs documentation for events, event handlers and event model.

Event to action

I was very keen when dealing with CLI that you should try to insulate commands from active routines. The same argument applies ti GUI. Not only is it clearer, but it is easier to re-use code. A typical GUI style is to have more than one way of triggering an activity. For example the 'save file' activity might be run (a) by picking from a menu, (b) by clicking on a button, or (c) when the program finishes, or (d) when a new file is about to be loaded 'on top' of the current one. That's four events and one action. It makes sense to code the action part in its own function rather than duplicate the code.

The functions in SrchRepl1.htm are organised a bit along these lines.

Alternative : Event listeners

In an OO world an event is an object that can be passed around. Typically an object will say to a component that it will handle certain types of events then implement those that matter to it. That's two things:

1 Tell a component the object wants events to be passed on to it.
2 Do something with them.

This seems quite a good idea from the point of view of 'getting things done in doing-things code' as opposed to whatever gets the initial message. Your 'doing-things' code asks to be passed a copy of events but otherwise doesn't care about the UI. However setting it up requires a lot more concentration. Java programmers note that you'll need to get stuck into a number of examples before you become fluent.

Review

Graphical components normally receive events such as mouse moves and clicks and also keyboard actions. These will be defined in YCPL's documentation.

The way actions are connected to events depends on YCPL.

- Assigning event handlers to otherwise null event methods
- Registering as an event listener with a component that might supply events of interest.

It is often a good idea to separate the interface related code from the 'doing-things' code. (Not always - keeping code together can make it easier to understand.)

Events are not limited to UI inputs. Components typically come with three main parts:

- *Properties* - What we've so far been calling object fields
- *Methods* - Functions
- *Events* - Hooks to attach event handlers

Components are ready-to-use objects designed for programmers to use in building applications. They will be documented to tell you how to use them. Many times you'll be able to use them off the shelf, but you might also want to sub-class them for specific requirements.

---

169 There isn't a timer object in Javascript but YCPL may have on.

Logical interface layout

Although you know that SrchRepl1.htm works, it doesn't get any marks for ease of use. There are no clues to which bit is which and Go on a button could mean anything. Let's smarten up the layout.

- Some proper indication of what the object of the exercise is
- Labelling the input controls
- More logical layout of controls. Button 'after' inputs.
- Grouping associated controls
- More informative button label

Note: When changing code and trying again in the browser you might want to force a complete reload with Shift+Ctrl+R.

```
<html>
<head>
  <title>Search and replace 2</title>

  <!-- no change to javascript from version 1 -->
  <script LANGUAGE="JavaScript">
    function DoSearch(){
      // what happens when we command s+r with
      // the go button
      document.form1.area.value = DoReplace(
        document.form1.srchStr.value,
        document.form1.replStr.value,
        document.form1.area.value);
    }

    function DoReplace(SearchFor,ReplaceBy,Body){
      // utility function to wrap regexp
      // search and replace
      rexp = new RegExp(SearchFor,"gi"); //g=all i=ignore case
      return Body.replace(rexp,ReplaceBy)
    }

  </script>

</head>
<body>
  <form name=form1>
    <table border=0>
      <tr>
        <td colspan=2 align=center>
          <h2>Search and replace</h2>
        </td>
      </tr>
      <tr valign=top>
        <td>
          <fieldset>
            <legend>Controls</legend>
            Text to search for<br>
            <input name=srchStr type=text   size=10><p>
            Text to substitute<br>
            <input name=replStr type=text   size=10><p>
            <input name=goBut   type=button value="Replace now"
                   OnClick='DoSearch()';>
```

```
          </fieldset>
        </td>
        <td>
          <fieldset>
            <legend>Text being worked on</legend>
            <textarea name=area rows=3 cols=40>
There is a young carver from Cressing
With curious habits of dressing
He always wears shorts
Of various sorts
But never a thong - What a blessing</textarea>
          </fieldset>
        </td>
      </tr>
    </table>
  </form>
</body>
</html>
```

I'm sure you'll agree that's a great improvement.

- There's a 'natural' left-to-right top-to-bottom flow about it. (It appears that when people look at a web page (and we might assume also any screen) they do so in an 'F' pattern. Sweep left to right across the top, investigate the left side then creep across a bit lower down.) *Ignore ingrained habits at your peril! You'll be amazed at how controls you think are 'obvious' can become 'invisible'.*[170]

- Notice the label for the second input text box is *not* "text to replace". An easy mistake to make. Even "substitute" is ambiguous. This example is particularly knotty because we could end up writing an essay to clarify what's being replaced and what's replacing it - at which point the user loses the will to live. Luckily we can cheat with the convention that you specify what's to be replaced with what to replace it with.

  This is an important issue: As the programmer you know everything, but how is the user supposed to have this knowledge? A bit of experience in interpreting context, guessing and most of all clear instruction. ...Where does it say the program will replace *all* occurrences?

- There was quite a lot of additional HTML coding. 31 lines instead of 11 in the body. This is quite typical when coding GUIs by hand. It's quite important to get it right first time otherwise you spend ages fiddling and debugging layouts.[171]

Validation and live logic?

WCPGW? Try replacing 'blank' with "x". The result is a mess. We can do something

---

170 See how many menus you can find on a single web page. Three seems to be the minimum - grrr. (The reason is to keep your attention as you hunt for a needle in the haystack.) For efficient applications you need your users to know where to find the bits they need without a second glance.

171 How *do* you debug GUIs? - Find some users. It's not very scientific but often quite revealing. The scientific, and cheaper, bit is being methodical about the possible inputs and combinations.

about this in two ways:

1 Do a validation check at the time the button is pressed and the procedure is executed.

2 Do validation and consistency checking before the button is pressed.

(Or a combination.)

Add the following code to the top of DoSearch() then see what happens when there is no search text.

```
if(document.form1.srchStr.value==''){
  document.form1.srchStr.style.backgroundColor = "yellow";
}
```

Striking, but only really a hint.

We could use the built-in Javascript function alert() to display a box with a suitable message.

> *DOM* - Document Object Model is the way elements are represented and manipulated inside a browser document. The details vary a bit between browsers. You will need to find then read the documentation. However without this you should be able to twig what the Javascript example is doing.

What about trapping blank text before we get to pressing the button? We can do this with the following alterations to the code:

- Leave the trap in DoSearch().
- Add a new function as follows

```
function HighlightBlank(InputBox){
  if(InputBox.value==''){
    InputBox.style.backgroundColor = "yellow";
  }
}
```

  InputBox will be passed to the function by the caller. This means we could reuse HighlightBlank() for trapping blanks in the replacer entry as well.
- Add the following inside the `input` tag for srchStr

```
OnChange='HighlightBlank(this)'
```

  This hooks-up the OnChange event to the validation function. `this` in Javascript is a way to say 'the object calling this method.'

You are now ready to try this out.

- If you leave the input blank it doesn't work.
- If you type in X then backspace-delete it you'd expect it to highlight... but it doesn't until you leave the input box.

Ummm. We were hoping for something a bit better than that. There are two problems: Not acting on the OnChange event until leaving the input box. (This is the official behaviour for Javascript - other GUI building languages may be different.) and not trapping leaving the input box blank without making any alterations. But at least HighlightBlank() works when it is called.

- Change `OnChange` to `OnKeyPress` in the input tag and try again.

This works as advertised unless we use the mouse to jump to the second box.

- Add a second event handler in the input tag as follows:

```
OnBlur=HighlightBlank(this)
```

Now clicking directly to the second box shows the alert.

Review

The process we've just followed is typical of GUI input development although one hopes

to get it more nearly right first time.

- Search the documentation of your components for suitable events. (In many languages you can sub-class components in order to hook to other events that may be particularly significant. For example a list of files, derived from a list of strings, might need to 'do something' if the files change behind the scenes.)
- Multiple events hook to the same event handler.
- Testing and debugging is necessary. Never let a GUI out before some guineapigs have had a go. Some people are very mousey, others love keyboard shortcuts. Watching people struggle with your 'obvious' and very clever GUI can be a but humbling.

> Getting the UI tested early-on is a good way to (a) show eventual users that their new system is coming along, (b) is an opportunity for you to check details about the way they work (c) give the users some idea of new ways of working that the new system might entail.

Warnings about control by control validity checking

- It is possible to force the input focus back to an input control which has failed to validate. This is not a good idea. If the user wants to waste their time on the rest of the form then that's their problem. At the very least you have to let them get to the cancel button.
- It is considered bad practice to have controls that magically appear when everything is ready and checked. Buttons are normally disabled rather than hidden until all other necessary inputs are complete.
- Experienced users tend to look at the keyboard or source document rather than the screen.
  - Make sure all the screens in your application 'work the same way'.
  - Provide keyboard shortcuts so navigation doesn't need the mouse
  - Validate inputs as they happen. You might consider an aural alarm if your users are going to be working 'head down'.
- Inexperienced users need lots of simple cues and clues. How is that date entered? Is that input marked "ID" an ID number, access number, service tag or what?
- Not everybody speaks English computerese[172]. In the exercise we've seen the problems of everyday terms without introducing spin control, text area, tabsheet, canvas and so on.
- Generally you're checking for basic sense not complete validity. Naturally you want to do an accurate job but there may be things that you need to consult the database about that will take too long to check and thus interrupt the inputting process. For example if a part number is always a letter and 5 digits you might check that format in the UI and leave the checking that the part actually appears on the database until the form is submitted for processing.

Further warnings

- The validation as-you-go is for the benefit of the user and should be thought of as a bonus for the programmer. Sometimes the programmer will rely on logic in the UI to prevent executive commands being called in the first place but in general at

---

172 And your grasp of specialist terms used by the users might not be so hot either!

submit time the programmer has access to much more contextual information and will be validating the whole submission.
- Web page programmers need to be aware that many users deliberately disable their Javascript capability. So you can't rely on field-by-field validation anyway.

More about GUI design

Firstly have a look at how other people organise their UIs. Just because it comes in an expensive or cool package doesn't mean it is a particularly good UI. It might be great to use - but only once you've struggled to learn it. Or it might be really easy to learn but frustratingly pedantic when trying to get quantities of real work done.[173]

The second thing is to see what specialist books and articles you can find on the subject in order to extract handy hints. Be aware that one persons 'best practice' is another's 'Dogs' breakfast'.

Although we've only discussed input as the UI there are important considerations for the display and presentation of information. A lot of research has been done here : far too much for you to absorb in one go. Some of it may be 20 years old and out of date.
- Not all people have good eyesight and may be using special software to help interpret screens. This is particularly important for web pages where it may be illegal to write web pages that disabled people can't use.[174]
- Clear presentation reduces confusion, support calls and disillusion.
- Find out how common colour blindness is.[175]
- In Ye Olden Days a terminal screen would have 24 lines of 80 monochrome characters using a fixed character set. At least everyone knew every screen would be the same. Nowadays life is more exciting and you can really make a mess. There is a fair chance you are developing on a screen with a lot more real pixels than users. It is easy to be carried away with the amount you can see on your screen and assume users will see it too. It is extremely important that not only do you try out layouts at the first opportunity on reduced hardware but you find a real low-spec system. Of course it is possible that the nature of your application is such that it's reasonable to specify a minimum specification - obviously there's a trade-off.
- Use of colours and fonts can be really useful, but don't overdo it. Red 'comes forward' while blue does the opposite.[176] Use these features in a logical manner...
- ... All graphical elements should be used in a logical manner...
- ... Including the whole screen. Typically you always put status information in the

---

173 When you get to know the alternative options for UIs you'll easily be frustrated at some applications you're forced to use every day. If you can't do anything about it at least try to learn from it and wonder if there were any good reasons for the choices made.

174 Not as such, but possibly in combination with other business policies. Free, automated web page accessibility checkers are available on-line so there's no excuse.

175 Far to common to ignore. I found out the hard way and had to re-do many of my lovingly crafted graphics for an on-line questionnaire with friendly "Which cyclist is in the right position : Red, green or blue" options.

176 So I wonder why the default Windows colour for active window frame is blue?

same place, menus on top or on the left and so on.

- Distinguish between:
  - Icons Handy eye-anchors that help to visually distinguish alternatives
  - Cartoons Graphics that adding interest for those with limited intellectual abilities.
  - Diagrams Instructional pictures illustrating an activity.
- Don't confuse instruction about a feature with help about efficiently using the UI itself. The former is about explaining what the program does and leading the user through the relevant processes. The latter takes the premise that the user knows *what* they want to do but is having a bit of problem with the *how*.

> If you want to mix *what* and *how* you can put sidebars, boxes or call-outs in with your main text. This works both ways:
> "- Advanced Primp-clogging features"
> or
> "- How to set up a new account"

Application UI structure

We've swiftly covered GUI for a single screen but in practice applications have a variety of tasks and sub-tasks. Somebody has to decide how to divide up these tasks into manageable units and how a user will navigate between them. There's a fair chance that the nature of the task will get you[177] most of the way to building a map of functions (user activities not methods in code) and sketches of the interface elements and their grouping.

- You may need to consider that some users might have options not available to others. Do you segregate these in separate screens or blank them out for the others? This depends on the nature of the tasks. For example a system administrator is likely to want a console with all the things they do handily grouped together.
- Consider what is a task in itself and what is a sub-task. These definitions are not the easiest of things to decide, but with a bit of inspiration you can clarify tasks into stages that must follow each other, or cover a different aspect of the whole. Tip: Tasks have goals. Write these down in simple terms the user could understand. (Use Beginner if you like as a framework for getting started.)

Wizards, tabsheets and dialogs

For a lot of the time when getting input from the user you're trying to keep them focussed on a single item in a single sub-task. There are three GUI structures that can help you do this.

- Dialog boxes
  Pop-up, do the sub-task then close to return to the calling point. File open, file save are ubiquitous examples. These come in handy in three situations:
  - where the sub-task is likely to be repeated throughout the application

---

177 If there is an interface designer or a ready-made design you still need to study the details in order to understand how you'll be communicating with the user, what contextual information they'll need, what parts are common and to be able to communicate clearly with the designer when practicalities suggest alterations are required.

- where complex input is required for a relatively simple result
- where there might need to be a check before proceeding; either by the user "are you sure" or by the system "is this a valid ID"

Always (for values of always) fully validate a dialog before closing it.

- Wizards
  A series of dialog boxes chained together to step through a process. Installation programs tend to use this method, collecting information and settings in a number of steps. Programming wizards looks easy but you need to be very careful about the order of dialogs, maintaining, validating and destroying state as the user moves forwards - and particularly back.[178] A tricky UI feature to get right for intensively used wizards is avoiding the need to use the mouse.[179]
- Tabsheets
  A random access stack of labelled dialogs. Typically used for settings where there are lots of defaults and therefore it is not necessary to visit and validate them all. A user will pick just the sheets they want, complete the dialog and return to the main application.

One task at a time and/or multiple documents

You are probably familiar with a word processor which allows you to have more than one document open on the screen but still forces you through a single print dialog to focus your attention on a sub-task. That seems like a great idea, it works well and we should have more.[180] Ummm but what about accounts for more than one customer or medical histories of more than one patient? Perhaps it's better if here the rule is 'one-at-a-time' to avoid mistakes.

> Clerks tend to work with single documents at a time. Professionals tend to collate and use a variety of ready-use tools together. (Sometimes you need to use brute force to make professionals use proper clerical protocols, in which case switch to a wizard for that task.)

When we're looking at those patient histories would it be handy to have the whole picture in summary form, details of individual episodes, a place where we can write our own notes, a form to fill in if we're forwarding the patient for further attention and so on all on display, all ready to be worked on at the same time? Yes of course - Having all the parts of a single case ready is one of the really marvellous things about using a well designed GUI. The typical implementation is to select a case then provide subsidiary windows[181] with supplementary information or task sheets. For example a doctor might have a patient history listing, details of recent blood tests, specialist's

---

178 It is traditional, but not at all necessary, to have cartoon graphics with wizards.

179 Necessary because having to use a mouse slows down keyboard input. Difficult because you need reliable 'next' without compromising 'have you checked' and getting tangled with 'Bzzzt-wrong' validation alerts.

180 Once you can process one 'document' having more than one is very simple and requires hardly any code - provided you didn't build the 'document' into the application but kept it as a distinct object.

181 By the way 'windows' were invented long before Microsoft nicked the term.

reports and 'what happens next' windows all floating on the screen together.

Untangling these displays and tasks in your mind

The first thing to remember is that behind the UI are tasks that need to be done and objects that will be worked on. A customer record should be as independent from a 'customer details screen' as possible.

> Are fixed procedures really fixed and should they be?
> - Closer inspection might show that a "we always do this" turns out to be for values of "always".
> - It may be a good idea to impose a fixed procedure. More about this in a later chapter.@@@???

Secondly look at where fixed procedures can be implemented. Quite typically there will be a context which will need setting up beforehand and external information that the user will need. A 'Dear Sir, thank you for submitting your manuscript for our consideration. I regret it is [insert reason from list]' standard letter needs name, address and reason. The first two come from one source the third from another. We can't start the rejection letter until we've got those bits of data and once started we don't need to divert until we've done filing, printing and posting.

Thirdly what are the paths between tasks. Many database/clerical applications seem to fall into the (login,) pick task from menu, select item to work on, work on record then return. This is a straightforward tree structure and might extend through more than one level as for example the customer details record offers an order history list to get to an order detail record which offers a list of order items and so to an order item detail. You'll probably find that this needs some modification where a task is accessible through multiple branches. A specific order might branch from a list of orders or a particular customer. Usually navigation between tasks is the easy part, it's the context you need to make sure you've collected on the way there.

Fourthly which sub-tasks should be dialogs, tabsheets or wizards. Do you need to partition tasks into logical sections. What context, if any, do they need?

Now you can look at what's left and combine an imaginative approach to the whole task with attention to the very smallest details. If that sounds like top-down and bottom up you'd be right. Your systems analysis should have told you what really matters so now you can craft - it's an art - a UI that's a pleasure to use.

@@@ diary as a GUI
TD
BU

Printing

Forms design and clear correspondence is a serious art. Some information on the printed page is more important than others. The same information sent to different people may need different layout or editing.

- All remote communications, particularly those sent to organisations which channel and prioritise correspondence, the subject and importance must be instantly clear.
- The effectiveness of communication is only ever measured in terms of receipt[182]
- Try to standardise layouts of similar forms. For example you might always put the address in the bottom left corner for ease of folding into a window envelope.
- Always put a reference number (and version number) on forms and couple them with a clerical protocol. (More on this in a later chapter.@@@???)
- Consider machine-readable elements such as a barcode on a 'renew your subscription' form. Make sure there is a human-readable version as well in case mechanical means fail.[183]
- Try to put numbers on pages and clearly indicate when there is something printed on the other side of the paper. You'd be amazed how many people ring up to complain saying "you haven't sent me..." only to be told "it's printed on the back".

> True story : I tried to find out how to get to five local eye hospitals as if I was a patient. None enclosed a map, only two even had maps - one said "it's signposted". This is where your programmer's skills kick-in. Before you know it you'll be ruffling the feathers of administrators who are under the illusion they're half-competent. That's why you need to be a good programmer with good communications skills and able to define the objectives that nobody else has ever really bothered to look at.

Review

Programming is about giving users reliable and easy-to-use tools. You can code the whizziest, coolest, fastest, code but unless you make it useable you might as well have stayed in bed. In a later chapter we'll return to the design and implementation of procedures.@@@???

It is now up to you, using YCPL to experiment with GUI controls. It's easy to get started but will take time to become fluent. Many languages have limitations, quirks, version differences and dialects. You'll just have to study the detail... ...not forgetting to decouple the UI from the action. I suggest you implement the diary design we've just discussed.

---

182 Please don't allow wooly language to obscure the message. Professionals need the facts up front, clerks need the data clearly and conveniently laid out. If writing to 'the public' state your business or the state of play in the first sentence. "Your clinic appointment is at Foo hospital, on 2pm Wednesday 3rd July 2006". Further explanations and instructions are required even if the sender knows all the information.

183 If label printing is part of your system then it can easily be a cause of annoyance. Somebody forgets to reorder the rolls to print on, the label printer breaks and the new model obtained at short notice isn't quite the same. Basically have a minimum of two *identical* label printers and spare label stock in a sealed bag with reordering instructions.

12. Good code

At this stage you could probably have a go at writing a web page using a database. Many people get a book like "programming web pages for idiots" and off they go happy with some result. As you now realise there's a bit more to it than that. If you have a cherished idea that you'd like to start work on, then by all means start sketching and trying out a few things...
... but read the rest of the book while you're doing it. The exercises are worth following as they are self-contained practice sessions where you can trip over your shoelaces, knock you head against intractable bugs, and generally make a mess without being caught out embarrassingly.

This chapter will introduce you to some technical subjects in enough detail to appreciate their value and work out where they fit in the grand scheme of things. Some of these subjects are huge so we can't explore all avenues and wrinkles here. You'll have to follow up with your own research according to what aspects are important to you.

Anatomy of melancholy

For sake of clarity, and to give you a sense of proportion when "the damn thing won't work" is driving you to utter frustration, let's define a couple of terms.

Bug Silly problem with code - it doesn't work you intended

Fault Major problem with program - it doesn't work as the user hoped

For example your program might work perfectly in all respects[184] except that there is no separate invoice and delivery address for orders - That's a fault.

A fault might be a bug and a bug is normally a fault but apart from this connection they're different animals. Bugs are caused by sloppy coding, typing mistakes, late nights and obscure things you can be forgiven for overlooking. (Apart from those caused by bugs) faults are caused by poor design and lack of knowledge about the right way to go about a job. The programmer is supposed to find all the bugs. You can leave the users to find the faults[185].

Delay and expense

Delay and expense will forever be your companions on the programming road. It's your choice: Do you let them hitch a ride with you (joined by their family of rage, despair, confusion, confrontation and disappointment) or do you wave to them as you go past.

---

184 But see Bug Free Program in glossary.

185 But often they stubbornly refuse to tell you so you don't get a chance to fix really crippling faults. I am a user of some very useful but slightly buggy software, every couple of releases I try upgrading but have to go back to version 1 because there are outrageous extra bugs added in later releases which make the software completely unuseable. Somebody who was new wouldn't have a hope. So have I raised a bug report? No. Life's too short.

By now I expect you are eyeing up possible projects to make a start on. What better time than now to make a start? Arghhh! Stop! Don't set out into the unknown with no experience, no guide and no map. WCPGW? You get half way then give up. This warning isn't just directed at novices, the last statistics I saw showed about half of all real-time¤ projects were abandoned before completion. Obviously a lot of WCPGW going on there.

The sorry fact is that this grief is completely avoidable, and yet it still goes on. A combination of underestimates, poor research, flaky and unreliable skills, unsuitable design, specifications without sufficient detail or appropriateness, enthusiasm before experience, changing requirements and 'circumstances beyond our control' continue trip up the smallest and largest project. It's a horrifying waste of money and effort.

As programmers often work alone or in small teams you need to have a realistic approach to *project management* and look after your own resource and delivery commitments. (There 100's of books on the subject, delving into some will give you some buzz-words, ideas and introduce some tools, but the best education I can recommend for anyone is to find a good manager and study their technique - It's a rewarding and enlightening experience to see good management in action after the usual bumbling.) You will get caught out time after time when things take twice as long as expected - OK so you have to adjust your initial estimates and stick to them.

Design before code

Perhaps the easiest way to make a mess of a project is start on the walls before the foundations. In programming of course we mean "Get the design right before starting programming" (And before the design comes the research.) We have done a fair bit on design already: Remember how long that took? We won't go over top-down, bottom-up again although we will refer to it so you may want to do two things:

1 Review the design section in chapter @@@
2 Decide now that you'll never start coding without a design.[186] (Sketched on paper or typed - it doesn't matter.) This document will play a very important part later so make sure your filing system is up to scratch.[187]

Threads

Question: How does your computer play music, connect to the Internet and operate as a word processor all at the same time? Answer: By giving each application a little bit of attention in turn. By switching between tasks quickly enough it gives the impression of doing them all at once.

Each independent task is called a *thread*. Like a chess grand master playing a number of games of chess

> Threads -v- Processes
> A process is a program recognised by the operating system and will be given its own priority, 'territory' and ability to own resources. A thread as we'll use the term lives within a program just as other objects and functions do.

[186] Sometimes for quick and dirty jobs of 20 minutes or so you can sometimes get away with top-downing in code. Other times you can crib from a template or cut and paste, filling in a few details from your head.

[187] You'll have to keep reading to find out what this mysterious magic is - It's too powerful for novices.@@@???

simultaneously, the processor deals with each task in isolation. As we'll see, threads can communicate and share resources and also 'sleep' if nothing much is happening in its neck of the woods just at this moment or 'die' if it has finished its assigned task. Some threads can be given higher priority than others.

Threads are hugely important. Even if YCPL doesn't support multi-threading (and many don't or with difficulty) you need to follow the story so you can make informed choices in future.

> Santa is in his grotto with his faithful elf, collecting toys to pack into the back of his sleigh. Santa's thread is about reading down a list, telling Elfred what to get, waiting, taking it off Elfred when he returns from the stores, checking it matches the requested item and putting it on the sleigh. Elfred's thread consists of accepting an order for a toy, trudging through the stores[188] picking a toy from the shelves, trudging back to Santa and handing it over then waiting until Santa has finished checking and packing and gives him the next item in the list.
>
> Santa is in rather a hurry and wonders how he could improve on this method.
> - By eliminating all the waiting
> - By having more elves
>
> For example to avoid the elf having to wait while he checks and packs the toy, Santa could give the elf the next item off the list immediately then the elf goes to the stores while Santa checks and packs. In a perfect world the time taken to check and pack would be the same time as it takes to fetch a toy from the stores and both would be fully occupied.

This gives a flavour[189] of the nature of the interaction between processes. Each has it's own routine but there are points at which they need to synchronise and objects that need to be shared safely. If there were two elves who simultaneously read 'the next item' off the list of toys, some good little boy or girl would get two identical presents.

A thread is a bit of code which is recognised by the processor (usually via the operating system) as an object to be treated to its own slice of processing time and having some privately owned memory. A whole program might be a single thread where the operating system automatically allocates a new thread to a program when it is started. More interestingly from the programming point of view a program may comprise many cooperating threads some as small as a dozen lines of code.

---

188 It's very cold at the North Pole and even colder in the grotto.

189 An utterly non-technical and not very accurate illustration - If you've never considered threads before you need a gentle introduction, in truth they're a bit wriggly in the details.

At this point coding gets tricky because implementation details vary a great deal between languages. In some languages threads are *possible* but very hard work. In the first instance just try to follow the concepts described below without trying to code them.

As already mentioned you need to know about threads even if your sort of programming appears to get by quite happily without them.

Santa the web server[190]

A web server has two types of processes.

- Listen for page requests (Santa - only one)
- Try to fetch and send the matching web page (elfs - as many as required)

Pseudo code for Santa(1)

```
Create a list to keep track of elfs
Open up for business
Loop{
    If(anybody trying to connect){
        Create a new elf using connection details to initialise
        Add new elf to the list of elfs
        Set the elf to run on it's own way
    }
    weed any finished elfs from the list by destroying them
}
```

Pseudo code for elf(1)

```
constructor{
    Reserve any private resources
    Store the client's connection details
}
run method{
    interpret the client's page request
    ask the operating system for file details
    if (file is available etc){   // if not then report error code
        open file
        while (more bytes to read from file){
            write a buffer-load of bytes to the connection
        }
        close the file
    }
    close the connection
}
```

Supposing we didn't have the elf threads and it takes five seconds to send a web page (due to delays in the file system and communications link) and there are more requests coming at the rate of one per second, then those requests have to join a queue or get

190 Here is an example of where it is really handy to give everyday names to things. We could keep referring to the 'main listener thread', but especially where we're trying to imagine individual entities going about their business a bit of anthropomorphism (or elfmorphism) works a treat.

lost. With threads there are two advantages:

- Santa is always ready to acknowledge requests because his loop can be executed very quickly.
- The time the elves can't do anything because they're waiting for the operating system or communications can be usefully used by Santa and the other elves.

If Santa is running round his loop when do the elves get a chance to have their share of the processing time? Any normal program that we've seen so far will never get to run an elf because Santa keeps looping. The answer is that there has to be some form of scheduler which either decides how to allocate time or a crude arrangement where Santa 'goes to sleep' for a while to give others a chance.

- Give-way or let-others-have-a-chance.
  This is often provided in modern languages. For example your code might be calculating pi to umpteen digits in a lengthy loop, but very politely 'writes' to the GUI to report progress... ...but no progress messages appear on the screen This is because behind the scenes the method calls to the GUI are being denied any time to do their display stuff because your computation is hogging all the processing time. The way round this is to do some calculation then give other threads (the GUI methods will be another thread even if you can't see it) a chance. When the progress message has been sent to the screen your code gets the baton back for another lap. And so on. Details will depend on YCPL.
- Sleep
  A thread may be waiting for something to happen. If it hasn't and can safely pause for trying again it may 'sleep' for a certain period, in which case it isn't a contender for processing time until however many milliseconds or seconds have elapsed. For example a thread that checks a connection is still alive might only wake up every five seconds.
- Proper thread scheduling
  YCPL may have a handy scheduling scheme built-in or you may need to use the features of the operating system. There are various ways in which thread scheduling is done[191] and you'll hear complaints about them all. At this stage any threading is better than none at all so don't worry.

  A typical method is to ignore lower priority threads and share time amongst the highest priority threads until they finish or ***block***. A blocked thread is one that has temporarily given up because it is waiting for some resource to become available or there are higher priority threads which are stealing all the CPU cycles.

How does Santa's loop finish? And what happens if there are elfs still active? The obvious way is to give Santa a Stop() method which when called will initiate a closedown. Take no more connections, clear up any remaining elfs then terminate. It seems rather rude to interrupt any elfs finishing off tasks so perhaps we wait until all elf threads have finished and been weeded from the list. (Now you see why we keep a list. In theory there is no need oi the elfs are completely free agents.)

---

191 Which can vary between different operating systems running identical code.

Santa pseudocode(2)

```
Constructor() - extends Thread{
    Create a list to keep track of elfs
    Set terminateFlag = false
}

method : Run(){
    Open up for business
    Loop(until terminateFlag is true){
        If(anybody trying to connect){
            Create a new elf using connection details to initialise
            Add new elf to the list of elfs
            Set the elf to run on it's own way
        }
        sleep  // *discuss*
        weed any finished elfs from the list by destroying them
    }
    Stop accepting new requests
    Loop(until elf list is empty){
        sleep  // *discuss*
        weed any finished elfs from the list by destroying them
    }
}

method : Stop(){
    Set terminateFlag = True
}
```

This looks more like something we can turn into real code. The business happens in the Run() method as before but with the addition of a closedown section which gracefully waits for any active elfs to finish their work.

- Notice that after we've called Stop() the thread is still active for some time. We can't rely on having called Stop(), or looking at the terminateFlag to indicate the thread has finished. In fact the thread officially finished when the Run method terminates.
- Whether there should be an explicit give-way call (sleep()) depends on the scheduling method in use. In principle we *have* to give way otherwise no elf would ever get to work, but the programming environment may deal with this necessity for us - at least if we're careful. Getting the compromises and rules right so that threads cooperate well in all circumstances is one of the fascinating challenges of the subject.
- WCPGW with the closedown? It depends on all the elfs finishing. Suppose for some reason an elf was stuck then we'd never get to terminate as the list of active elfs would never be empty. In this case the answer is simple: Santa doesn't need to wait for the elfs to finish there is no dependency. So Santa can die and leave free-range elf threads running wild.... ...But some process has to own these threads and be able to terminate them if necessary.

These issues are dealt with in various ways by various languages and operating systems. Don't expect to master threads overnight.

More reasons why threading is in the advanced syllabus

Sharing resources and methods is part of the game. One thread might be adding data

to a buffer and another reading it. If the data is static there is no issue but if there's a stream of bytes being added and reading involves taking what's in the buffer out there's a whole lot of trouble in store. Here's what can go wrong.

Thread 1 : Add "ABC" to buffer
Thread 2 : Read three characters "ABC" from buffer
Thread 2 : Delete three characters from buffer
Thread 1 : Add "DEF" to buffer
Thread 2 : Read three characters "DEF" from buffer
Thread 1 : Add "GHI" to buffer
Thread 2 : Delete three characters from buffer
Thread 2 : (No more to read)
Thread 1 : Add "JKL" to buffer
Thread 2 : Read three characters "JKL" from buffer

Although thread 1 sends `ABCDEFGHIJKL` thread 2 only receives `ABCDEFJKL`. (There are other failure modes as well.)

The solution is to allow methods to lock-out all other threads for a period. In this example Thread 2's ReadBuffer method demands exclusive access to the buffer throughout both it's operations (ie. read and delete)

```
method ReadBuffer(){
  Wait for exclusive access to buffer
  Read any characters
  Delete any characters
  Release lock-out on buffer
}
```

Because data is often accessed through methods you might want to get an exclusive right to use an *object's method* in the same way.

What about the following? As copying things might take a long time we think it's a good idea to make it threaded so that other foreground processes can get a look in. Here is the pseudocode. WCPGW?

```
constructor : Copier(FirstThing,SecondThing) - extends Thread{
  first=FirstThing      // store local ready
  second=SecondThing    // to be used in Run()
}
method : Run {  // the threaded bit
  Wait for exclusive lock for first
  Sleep(100 milliseconds)  // (contrived)
  Wait for exclusive lock for second
  Do some copying from first to second
  Release lock on second
  Release lock on first
}
```

Let's see what happens if we try the following:

```
c1 = new Copier(foo,bar)
c2 = new Copier(bar,foo)
c1.Run();
c2.Run();
```

c1 : Get lock for foo. Sleep.
c2 : Get lock for bar. Sleep.
c1 : Wake up. Try to get lock on bar...not yet available...wait
c2 : Wake up. Try to get lock on foo...not yet available...wait
c1 : Try to get lock on bar...not yet available...wait
c2 : Try to get lock on foo...not yet available...wait
c1 : Try to get lock on bar...not yet available...wait
c2 : Try to get lock on foo...not yet available...wait

And so on for ever. This is called a *deadlock*.

Review

Programming with threads is like riding a steeplechase. Exhilarating but not for the faint-hearted. If you're writing programs where more than one activity has to wait for something then you should look at how you can avoid the delay to one task holding up all the rest. These sorts of delays are endemic to communications. For example suppose you are writing a web crawler, (a bot that follows web page links by downloading them then following those links and so on.) Do you put all your requests in a single queue and wait for each one to be satisfied before asking for the next one? No of course not, you send out lots of requests 'simultaneously' and process the results however they arrive, sending more requests in the meantime.

Algorithms

An algorithm is a method in the sense of a method of getting to a goal given certain starting conditions. The interesting thing is that they can be studied formally to prove under what conditions they will, 100% without fail, get a correct answer, to give an estimate of the average and worst 'cost' (typically of time or storage), and to give guidance for programmers on how to check that their code matches the specified method exactly.

The reason algorithms are important is that there can be an enormous 'cost' difference, easily 1000 times, between an efficient method and the one you first thought of. Just suppose the phone book wasn't sorted into alphabetical order. That's exactly the situation you are faced with when trying to access a database. Do you physically sort the records, do you extract a bit and physically sort them, do you make an index with each item pointing to the next one in the sort order, do you make an index by putting all the A's together then within that the AAs and within that the AAAs and so on? How would you sort your records or index anyway?

Luckily you can find solutions to pretty much every problem off the shelf. There are typically a number of methods to chose from. Some are simple to code but take a long time to execute. Some are tailored for specific data representations. Some work well 'most of the time' but might occasionally fail or take a long time.

> Good news : The wheel has already been invented.
>
> Bad news : A lot of published code contains bugs or non-obvious limitations.

Your first job is to research the possible options and select one. Some of the presentation might be rather daunting, don't worry you're after the method clearly explained, or better already coded. Nowadays the Internet makes this easy and you're very unlikely not to find some ready to use code. Do not be afraid to look at alternatives.

Whether you find code or have to interpret the method yourself there is something vital you must do: Have some idea of how the method works. In researching the 'how it works' you should come across the limitations and alternatives. If you find multiple implementations of the same algorithm compare the details - it's often quite surprising

what a difference there can be between two bits of code that are billed as doing the same thing.

Let's look at an example:

> Given a list of numbers, what is their range. i.e. the difference between largest and smallest. (Assume numbers are in an array.)

This might not strike you as the most difficult coding job so before continuing with this book have a go in YCPL.

Here is one implementation in pseudo code:

```
Find smallest number
Find largest number
Result is largest - smallest
```

Here is another

```
Run through list
    Looking for smallest
    Looking for largest
Result is largest - smallest
```

And another

```
Sort list
Result is last number - first number
```

The first is a simple top-down approach with the advantage that The Find routines are likely to be 'the same'. This might make it easy to code. The second recognises that we can probably find the largest and smallest in a single pass of the array. This should make it faster. The third is easy to understand, possibly extremely reliable if we have a ready-use sort routine, but the sorting involved might mean a great deal more processing. ('Might' because the list could possibly be sorted already for some other purpose.)

Heuristic algorithms

The classical algorithm can be shown to deliver a specified result within certain resource parameters such as run-time. A heuristic algorithm is usually a cheap-n-cheerful or guessing method which generally works but can't be proved.

There is another method we could use for finding the range:

> *Take a sample of values*
> *Calculate mean and standard deviation*
> *Use the sampled values to compute whole population statistics*

This is a perfectly respectable way to measure data parameters and might be a lot 'cheaper' than looking at ten million data items.

Research

Now and again you should make some time to explore other people's solutions in as wide a field as possible to see what sort of problems have been 'solved'. There's no requirement to learn the details or make an extensive catalogue, just to get a feel for the many aspects of theoretical and applied computing where lots of work has been done. This should give you confidence and a starting point when presented with a real problem. Solve a maze, rotate a graphic, optimise cutting shapes from a sheet of material, locating pallets in a warehouse and so on.

A Real Programmer will (a) assume others have attempted the same problem previously

and (b) have a few ideas where to look.[192] NB Remember times change and techniques evolve - What was infeasible 20 years ago might be crackable in an hour on a PC.

Putting an algorithm into code

> What follows is what a lot of people think of as programming. Although it's only a part : Pay attention. Take note. This is sorting-men-from-the-boys stuff. *It could take years before you get your long trousers.*

Let's try coding the second method above:

```
largest = 0;
smallest = 0;
for(i=1; i<arraySize-1; i++){
  if(array[i]>largest){largest=array[i]}
  if(array[i]<smallest){largest=array[i]}
}
result = Largest - smallest
```

WCPGW? There's roughly one error on each line.[193]

- Which is the largest of -4,-5,-6 and ... err... 0? We need a better initial value for largest.
- Which is the smallest of 10,11,12 and ... err 0? We need a better initial value for smallest.

Cue initialising smallest and largest to the largest positive and negative numbers that the type will accept. Many languages provide you with handy constants or functions for this. Obviously if you know you're dealing with people's ages or shoe size you can stick in 0 and 9999.

- The array index i starts at 1. How do we know this is the right place to start? Possibly this should be 0?
- Is the loop finish condition correct?
- If there's no *off-by-one* error in the loop finish condition is the array full of data?

> Don't be afraid to work from the end of a list or array to the beginning and count down your index. This makes it easier to check the terminating condition both by looking at the code and at debug time.

If you assume there's at least one error in every loop you won't go far wrong. (Don't forget the ==/= gotcha for starters.) Debugging the first time round a loop is fairly easy (but unnecessary if you get it right before testing), checking the loop exits at the right time is a lot more difficult. Moral : Double check start and end conditions. Here's another gotcha: You process the elements in a dynamic list deleting them after working on each one... and increment your index!

- There's a copy-paste error in the next two lines

---

192 BISE (Before Internet Search Engines) you'd have to rely on specialist books or learned journals. Just because knowledge is 'available on demand' doesn't mean you should wait until you need it to make an exploration - How will you know it's there?

193 You may guffaw now but I'll have the last laugh when you're staring at code that won't work at one in the morning.

- If the language is case sensitive then capital-L `Largest` is an error.

Don't forget stupid typing and transcription mistakes. You will make dozens each day of which most will be spotted immediately... And the rest will lurk. One of the reasons for developing naming conventions is that you automatically put capitals in their 'right' places.[194]

Next try:

```
// find difference between max and min
// array elements in range hi to lo inclusive
largest = MaxPossValueForType
smallest = MinPossValueForType
for(i=lo;i<= hi;i++){
  if(array[i]>largest){largest=array[i]}
  if(array[i]<smallest){smallest=array[i]}
}
result = largest - smallest
```

WCPGW?

- The initialisation of largest and smallest is back to front
- What are the valid values for hi and lo. Should we check?
- If there are no items in the array we need to have a return 0 rather than a return difference between MaxPossValueForType and MinPossValueForType.
- Can you spot the error in the last line if lo>hi?[195]

Next try:

```
function FindRange(AnArray:array of int, Lo:int,Hi:int) returns int {
  // tell what variables we will be using
  var smallest : int
  var largest : int
  var i : int
  // validate indexes
  if(Lo<GetLowestPossibleIndex(AnArray)){return 0}
  if(Hi>GetHighestPossibleIndex(AnArray)){return 0}
  if(Lo>Hi){return 0}
  // initialise min and max
  largest = AnArray[Lo]
  smallest = AnArray[Lo]
  for(i=Hi,i>Lo,i--){
    if(AnArray[i]>largest){largest=AnArray[i]}
    if(AnArray[i]<smallest){smallest=AnArray[i]}
  }
  return largest - smallest
}
```

---

194 There is no one-right-way of naming. Some people always put an action word such as Get, Set, Do, Select, Wait etc at the front of functions. This can be useful when you need to be clear whether you're referring to GetLatestInfo() or LatestInfo. (GetLatestInfo() might do the necessary refreshing of the LatestInfo variable so mistakenly referring to LatestInfo is equivalent to GetOutOfDateInfo()!)

195 Subtracting the smallest possible number from the largest possible number gives a number which is larger than the largest possible number. You've tried herding cats. You've tried nailing jelly to a wall. You've tried keeping that unfortunate episode with the Boy Scouts quiet. Now try bug-free coding.

This looks a bit more professional.

- We have defined the inputs and output a bit more tightly
- We have told the compiler the names of the variables we'll be using. This should let it trap us trying to use typos and also if we were using incompatible types.
- Validation should work *provided* `return` takes immediate effect. In some languages you'd be *setting a return value* so the subsequent lines would continue to be executed. (Some people, including me, like to limit the number of exits from a function to one normally, but here it's very clear what's happening.)
- Should the third validation be `if(Lo>=Hi)`? Yes.
- Most languages have some functions for implementing GetHighestPossibleIndex() but remember that for zero-based arrays the highest index is the size less one.
- Why mess about with artificial values to initialise when we can use real values?
- Does the loop look easier to verify visually? I think so. (There's even a safety margin if we put `>=` in the condition instead of `>`.)

Additional points to note about this routine.

- You will want to add some comments to be used for usage documentation.
- Consider allowing out of range Lo and Hi arguments and constraining them to the limits of the array rather than rejecting them. This can make the routine a lot easier to use.
- Although not part of this routine, the purpose for calling the routine might be to allocate 'bins' for collecting statistics. If the lowest value is 5 and the highest 15 the difference (as supplied by our function) is 10. But we need 11 bins. This is called a *fencepost error*¤.

Review

How could there be so many problems with such a simple routine? Answers on a postcard please...

This is completely normal. After WCPGW comes HCIPBSS - How could I possibly so stupid!

You can probably get an inkling of why it's such a good idea to break programs up into small units and make sure each unit is bugproof. Of course *then* you have the hassle of sticking the bits together the right way, but you can document that in a few lines per function.

Optimising

What if, in our example, we were looking at say the number of images on lots of web pages? Instead of an array to give a value we might have a function that looks at a web page and counts the number of `<img>` tags. Here is what the loop code would look like.

```
if(GetImageCount(i)>largest){largest=GetImageCount(i)}
if(GetImageCount(i)<smallest){smallest=GetImageCount(i)}
```

That looks like a possible four calls[196] to the same 'expensive' function. We can reduce that to one very simply:

```
var imageCount = GetImageCount(i)
```

---

196 Actually two and a little bit because the conditions are only occasionally true.

```
if(imageCount>largest){largest=imageCount}
if(imageCount<smallest){smallest=imageCount}
```

That's more than halved the number of calls to GetImageCount(), but we can go further. The test might be 'expensive' in which case can we save unnecessary tests? Yes we can. Because largest and smallest are initialised to the same value we can be absolutely certain that if a value triggers the first test it can't possibly be the smallest as well.

```
var imageCount = GetImageCount(i)
if(imageCount>largest){
  largest=imageCount
}else{
  if(imageCount<smallest){smallest=imageCount}
}
```

In *this case* we won't save much[197] and there is more to go wrong and it's more difficult for somebody else who comes along later to alter the code to understand what's going on. So, on balance, even if we did as we ought to have done and put some comments in the code it is probably not worth the worry.

Optimisation is not a substitute for having an efficient algorithm in the first place.

More about the cost of algorithms

Lets suppose it takes you 1 second to put a record card into one of 10 pigeonholes marked 0 to 9. Assume there are no duplicates. To sort 10 will take 10 seconds. To sort 100 will take 100 seconds to sort on the tens digit to end up with 10 cards in each pigeonhole. Now remove those and place in 10 stacks ready for sorting each stack of 10 which will take 10 seconds a time. Total time will be 100 + reorganising time + (10 times 10) = about 200 seconds. To sort 1000 will take 1000 seconds to sort on the 100's digit then 10 lots of how long it took to sort 100. Also we'll need space for 10 more stacks of 10 cards (as used in the sort-100 routine) and 10 for stacks of 100. If we tabulate this (excluding time taken to empty pigeon holes) we get the following.

Number to sort	Time	Stack space	Time per card	Stack space per card
10	10	0	1	n/a
100	100+(10*10)=200	10	2	1/10
1000	1000+(10*200)=3000	20	3	1/50
10,000	10K+(10*3K)=40K	30	4	1/333
100,000	100K+(10*40K)=500K	40	5	1/2500
1 million	1000K+(10*500K)=6M	50	6	1/20000[198]

---

197 Because in practice there will only be a handful of cases where first condition applies even with hundreds of data items.

198 I know this is a thought experiment but doesn't 1/20,000th stack space ring any alarm bells? Err.. That's stacks of cards 30 foot high! Don't forget your reality checks - they should be part of your Real Programmer's psyche.

The time taken to sort each card using this method keeps increasing. You will see it's the same number of seconds as number of digits in the number of cards - bingo! - A logarithmic relationship. That is the time *per card* is *proportional* to log(N) (N being of course the number of items being processed.) The consequences of this are that sorting large numbers of record cards will be disgustingly slow. Not only are there more to sort but each one takes longer so there's a multiplying effect. So the total time to sort N items is N times log(N).

> We're about to dip into the maths you drowsed through on Friday afternoons. In particular logarithms and polynomials.

Order of

Wherever you see that "N times" you're in polynomial country. For example the number of ways to combine a set of N items[199] is "Half N times (N-1)" or $(N^2 - N)/2$.
When comparing algorithms we typically simplify

- We're interested in the *relative* cost so we forget the "/2"
- The N^2 bit will vastly overshadow the N so we only look at the largest power

We then say this is *order of* N^2 (Normally written as "$O(N^2)$")

'Order of' is used to refer to the 'cost' of the complete operation. So our pigeonhole sort is O(N logN) for time and O(log N) ish for stack space.

The reason for bothering with the cost-per-card columns is in case we were looking at real costs. If the space for a stack costs £10 and an employee is being paid £3.60 per hour we can soon see that for 1000 or so cards the stack cost dwarfs the wages bill but for one million wages cost £6000 but the stacks cost only £500.

> Step costs. Compare the stacking space required for 990 cards and 1010 cards. Suddenly we need space for an extra ten stacks. There are all manner of these limits in real life. Some of them are the difference between feasibility and unacceptable costs. It's good policy to be frugal all the time and very clever occasionally.

Know your N

So you can see that unless we know how many cards we'll be sorting we can't hope to optimise our operation. For example if we had more pigeonholes and stacks we might get away with less time. Or vice versa.

The sorting we've looked at is typical of a quite good sort being O(N logN) but there are some that are $O(N^2)$ which for moderate N

> You can easily research sorting algorithms as they've been studied extensively and many variants published. Don't re-invent the wheel.

---

199 For example all the distances between N places.

becomes enormous.[200]

The converse is true: Small Ns can use very crude sorts with small overhead. But - (Big But) - never ever[201] write a sort in-line. That is you've got a dozen records that need sorting so in the middle of your code you start typing something like:

```
for(i=0,i<count,i++){
  // shuffle records
}
```

No No No! By all means

```
SortMyRecords(arrayOfRecords)
```

with a separate function. The reason for this ban is that there's a 10 to 1 chance you'll introduce some bug or other which will be difficult to detect.

Adjust your overall approach

Suppose once in the dim and distant days the cards had been sorted. Now when we have a few to add we can sort the few new ones then work our way through the piles inserting the new ones as we go.[202] This won't take very long.[203] A typical computer version of this is incoming emails and news items that need shuffling into place.

Beyond sorting

Sorting is a very good place to start looking at algorithms because the science and trade-offs are all there. However there is a huge field of problem solving methods that have been researched and the results published or packaged. Here are three typical approaches that you may want to think about when looking at the solvability of your private problems. (Obviously after researching how people might have done it already many times over.)

- Breadth-first or depth-first
  Suppose you are exploring a network. It could be a communications network, a network of possible moves in a game of chess, a set of linked web pages or family relationships. Let's use the last example. From your starting point, let's say that's you and you're listing your ancestors, you can find your father, his father, his father

---

200 In the days of mini-computers I once found a crude O(N-squared) sort inside a friends program that was being used to plan government defence spending. Before I replaced it (say 30 minutes work) this what-if program was taking 2-3 hours for each run which tended to spoil everyone's day. Afterwards about 5 minutes. As I wasn't getting paid I didn't bother with a proper audit - I wonder if the results were worth the paper they were printed on?

201 For values of never. If you need to this you're probably better off coding a separate function to start with and only making the code in-line as a last resort when you can test against the normal version.

202 This is a merge-sort.

203 So long as we don't drop the sorted cards on the floor - something that could happen quite easily in days of punched cards. By the way: If you have a set of cards to be kept in order take a jumbo felt pen and rule a slanting line across the side of the stack. Each card will have a black mark in a slightly different place along its top edge. This means you can instantly see any that are filed out of order. There is a similar trick if there is a flaky collating process. Print a black rectangle right on the edge of each page and work your way down by a rectangle's height for each new page

and so on until you run out of data. Now, starting from the furthest father look for mothers, brothers and sisters and wives. If any found then look for their ancestors and then their contemporaries. When all possibilities have been exhausted come back one generation and repeat... eventually that's all of your father's side of the family documented. Do the same with your mother's side, then finally your brothers and sisters (who may have different parents to you.) This is known as the *depth-first* approach.

Alternatively you could exhaustively list 'your generation' then for each member of that generation investigate each parent and that parent's generation (ie uncles and aunts on one side) but before looking at their parents look at the other parent's siblings. Then repeat for all the parents of the first generation. This is a *breadth-first* approach.

So which method to use? It depends. (When you follow the iterature)

- Divide and rule
  Is it possible to break a N-sized problem into a two half-N-sized problems. If so you can probably repeat the division until there's a simple way of handling a small number of items. This is what we did with the pigeonhole sort.

- Repeat until done
  Quite often this is the heart of algorithms. There are one or more states where something is done and rules are applied to decide what the next state will be. For example when looking at the family tree example above we had rules to 'add new person to list', 'If father found then new person is father else if mother found then new person is mother...if no parents or siblings then go back to the person who led us here and mark this branch visited' [204] (The bit the computer scientists never fail to emphasise is 'until done'. Well they have a point!)

Loop invariant

Before each deal of a hand of cards you can always say that 'nobody knows where any of the cards are in the pack (And all 52 are in the pack)'. As the hand progresses this condition is no longer true, but after the play you collect and shuffle to restore the initial condition. A condition that is true at the end/start of each repetition of the loop is called the *loop invariant*.[205] Each aeroplane flight has a loop invariant : "All wheels on the ground." Anything can happen *during* the loop but the loop invariant must be true before the start and after each cycle.

Computer scientists like to use this to prove algorithms work and you may find it helpful to put in some comments to help whoever has to maintain and debug the code about

---

204 *Traversing trees* is quite an important subject. There are plenty of algorithms. In your branch of programming you may not come across this subject, but if you get an opportunity to give it an afternoon's attention do so.

205 If you look this up on the web or computer science texts be prepared for a long slog. It's a little gem the scientists make a big thing of because you can use it to prove with cast iron logic that an algorithm will do what it says on the tin.

what must be true so you can convince yourself that despite all the shenanigans going on in the body of the loop, certain key aspects will hold true ready for the next iteration. What aspects? Typically, and most usefully, one that describes the very condition you're hoping to achieve as your final goal. Flying to Sydney? Regardless of the number of times your plane lands to refuel, when you get to Sydney that 'all wheels safely on the ground' is a really good loop invariant.

Assertions

Practical programmers are more pragmatic and use the idea that some things must be true at a point in the program in a broader way. Suppose you have a number of web pages linked together to perform some operations. When you're at say "Order menu" what can you be certain of? Do you know the user ID and have you validated their login and permission to play with orders? Is the database connected? These are bits of miscellaneous state which may be modified somewhere else in the system so are tricky to be absolutely certain that you've lined everything up just right before you start work on this page.

An *assertion* is a statement that should always be true. Some programming languages provide an assert() function which would normally not be compiled into the code but can be activated in a debugging mode. Many programmers swear by assertions, others seem to get along fine without them.

Assumptions

My personal approach (don't forget the sort of programming I do may not be the same as yours) is:

- To put in comments about the assumptions being made
- To actively check important items if there is more than zero probability of unsuitable state.
- To check function arguments.

Ie not to use an assertion function, but to clarify what's expected and check. The more your code is split into functions the more frequently you can check the values as they get passed to functions. Clear documentation is essential.

> *Assumptions* generally relate to external state which you're going to be relying on. This is often a fact of life, but you can do a little bit about it by putting as much code as possible in functions that don't refer to external variables.

You may see this sort of code

```
if(IsUserLoggedIn()==False){
  JumpToLoginPage()              // can never happen
}
```

Which is a succinct way of dealing with a possible issue and highlighting for the benefit of anyone reading the code that we are assuming the user is logged in.

Exceptions

You've probably seen enough rude exceptions thrown in your face already. Here's how to make better use of error conditions and even create exceptions of your own.

When we need to deal with some situation we have two alternatives:

1 solve the problem locally

2 admit that at this point we're not qualified to fix the problem and refer the matter to some higher authority.

An example of the first method was given in the previous section where we checked to see if the user was logged-in. Another common method is to assign some default values to results, so that if some part of a procedure fails we've got something better than nothing to fall back on.

Sometimes what we want to alert is not a 'problem' but some situation which needs handling and might arise somewhere in a block of code.

```
Be prepared to be interrupted{
   do everyday activities
}
by the phone { answer phone }
by the microwave going 'ping' { stop for lunch }
by something else {
  Pass interruption to boss to deal with - tell them what's cropped up
}
```

In this example we avoid 'answering the microwave' by having distinguishable types of interruptions. The first two cases we handle ourselves, but the 'something else' needs referring to the boss.

Now imagine you're the boss and a voice on the phone says: "Boss! Something's interrupted me". How useful would that be? Not a lot. You might want to know which of your employees was calling, more details of what 'something' was and what the employee was working on at the time. You might tell people that you don't want to be interrupted for minor matters - Which begs the question 'how is minor defined'?

Finally, when there's an interruption there might be some tidying up that must be done before quitting. You wouldn't expect a tele-sales person to leave their station in mid-call, so

```
finally {finish current call}
```

The reason for `finally` is that exceptions can disrupt the normal flow of logic so that there may be no guarantee that appropriate finishing actions such as freeing resources and closing files will be taken.

Exception handling is not available in older languages. A traditional approach with these is for functions to return an *out-of-band* value. For example when converting months from names to numbers a function might return -1 if it can't make a proper interpretation. The caller then looks for this and treats it specially.

> Beware of exceptions inside constructors. You need to discover exactly what happens with YCPL.

Lets see these concepts in some (quick and dirty) Fudge code.

```
// Use a string in the form ddmmyy to create a day object
constructor Day(DateString : string){
  int d,m,y
  d=1; m=1; y=0
```

```
    try  // ... finally
      if(StringLength(DateString)<>6){
        raise InvalidDateStringExecption('Must be ddmmyy');
      }
      try{  /// ...catch
        d = StringToInt(SubString(DateString,0,2));  //[dd]mmyy
        m = StringToInt(SubString(DateString,2,2));  //dd[mm]yy
        y = StringToInt(SubString(DateString,4,2));  //ddmm[yy]
      }
      catch StringIsNotAnInt exception {
        raise InvalidDateStringExecption('dd,mm and yy must be ints');
      }
      if((d<1)or(d>31)or(m<1)or(m>12)or(y<0)or(y>99)){
        raise InvalidDateStringExecption('Invalid dd,mm or yy'');
      }
      // better date validation to go here
    }
    finally{
      this.day = d
      this.month = m
      this.year = y
    }
  }
```

- `Try...finally` operates to ensure that some default values are set regardless of any exceptions raised on the way.
- We do a straightforward test for string length and chose to *raise* (often called *throw*) an exception. We have defined this exception object somewhere else so you can think of `raise` as an instruction to construct the exception object and activate it.
- It is quite normal to add diagnostic information to exceptions.
- When we're converting the string into integer parts, the function we use to do it might throw an exception. Documentation will tell you exactly what exception types are thrown. Some languages insist that a function declares what exceptions it will throw which enables the compiler to check that something is going to catch them and that there won't be strange exceptions bubbling up through the nest of calls which burst out unexpectedly because you've forgotten to handle them at an appropriate place....
- ...So we catch any exceptions (If we didn't these would be propagated to the caller and its caller etc until the user sees a crashed program with a not so useful message like "`Invalid number at 4545432:65454323  Program halt.`") instead turning them into our customised exception.
- We do some more logic testing in order to trap illegal values. (I'll leave you to contemplate the poor quality of these lines of code.)

Here is how it might work in practice:

Pass "25 Dec" to Date constructor.

Construct parent object. Allocate space for three integer variables and assign values to them. Start a block of code which will deal with exceptions.

Pass argument to string length function...result is 6. Test fails so don't execute block. Start a block of code which will deal with exceptions.

Pass argument to string chopper function. Result is "25". Pass "25" to string to integer convertor function. Result is 25. Pass argument to string chopper function. Result is " D". Pass " D" to string to integer converter function. Exception of type StringIsNotAnInt 'returned'. Halt normal program flow and jump to exception handling at end of block.

We have got a specific trap for StringIsNotAnInt so do the associated block of code. Create a new exception object (specifically of type InvalidDateStringException) and give it a text message to carry. Halt normal program flow and jump to exception handling at end of block.

Set the day field to d, the month field to m and the year field to y. Return from constructor.
Oh dear constructor has returned with an error.... *(handle it....presumably by some message to the user)*

Note that there could have been the possibility that the SubString function returning an error. In theory we've guaranteed this can never happen because we've made sure with the preceding test that the string always has exactly six characters. But what if we made a programming error and have used a base index of 0 for the first character when we should have used 1?[206] There will be an exception like InvalidIndexException raised by the SubString function. What happens then? In this case normal processing halts. There is no matching trap for InvalidIndexException and so the exception propagates up the call stack 'as-is'. The finally block will still be executed. *In this case this is probably the desired behaviour*, but there are two alternative ways we could have coded the trapping of these exceptions:

1 Have a general catch-all which will trap any sort of exception.
2 Explicitly trap each type of possible exception and handle each one accordingly

As a programmer you've got the choice so use it wisely. The second method can be overkill - Why would you want to check for stupid programming errors you'll find on the first debugging run and won't mean anything to the user anyway? But the first method could be very confusing. Suppose we'd used the catch-all trap and came to testing our program. That's very strange! When we type in "121212" for the date we get a `dd,mm and yy must be ints` message. Half an hour later, after trying all sorts of ddmmyy combinations and swearing at the StringToInteger function we eventually realise our catch-all was masking the InvalidIndexException which we could have fixed in 20 seconds.[207]

Review

In Ye Olden Days we seemed to manage without exception handling but I wouldn't try it now.

- Exceptions come in various pre-defined and (usually) programmer devised variants. It is simplest to think of an exception as an object that will burst out of it's enclosing box and will continue to burst out of enclosing boxes unless trapped.

- Exception handling is generally based on blocks of code. `try` is often keyword used to indicate the start of such blocks.

- `finally` is useful to ensure that code which must be executed come what may, such as releasing resources and closing files is processed. You'll frequently see this:

```
open file
try
```

---

206 This is a very common error. Languages don't seem to have any consistency. Some that have zero-based arrays have 1-based strings. Those with zero-based strings need special care when searching for the first occurrence of one string inside another. How does it differentiate between 'not found' and 'found at first position'?

207 I use an otherwise extremely useful program which occasionally crashes with an "unexpected error" message. That's it! I wonder what the programmer was thinking about when they wrote that?

```
    process file
  finally
    close file
```

Which if you think about it is a really good idea because all sorts of things might possibly go wrong during the processing. Otherwise the operating system thinks the file is still in use even after our program has collapsed in a heap - so stopping future access until a reboot.

- Catching exceptions needs to be done with care. There's often a balance between catch-all and catch every possible exception. For the most part catch the ones you expect and leave the others to propagate.

- Throwing or raising exceptions of your own making is a handy way of identifying specific problems and being able to take appropriate action. Avoid masking detailed exceptions with generic ones.

- Remember to give useful error messages to users in language they understand.

13. Testing and quality

The practice of testing often goes as follows: "It hardly ever crashes - That'll do".

The theory of testing is about saving down-the-line effort by finding faults while the beast is still in the factory. Not just 'getting the code to work' but 'assurance that it does the job every time'.

In this chapter we'll look at the theory of testing first as everyone claims to do it, then consider practical issues then discuss what we mean by 'quality' and how the right approach can shine a light on your programming strategy.[208]

Introduction

As well as logic you may need to test utility, appearance, usability, reality of assumptions about inputs, speed and security. (There is a grey area where testing can shade off into tuning.)

Testing takes time, is labour intensive and boring, is an art, and is often an obstacle to signing-off the code as finished. These factors are not conducive to giving testing the necessary attention.

- How do you know what to test? - Inspecting the machinery
- What are you looking for? - Does it do the job

These are two different aspects which can easily become confused. There's no reason why they shouldn't be mixed together so long as you can distinguish between 'unit of code' and 'purpose of application'

Syntax checking

Many development environments have degrees of syntax checking built-in. They might find unmatched braces or just highlight key words. Normally though you only find out about syntax errors when you try to run the code. The consequences of this are that you need to get your code to at least run or be compiled in some form as soon as possible. This means either small stand-alone units or incremental coding for frequent code-compile-fix syntax-try to compile again cycles. The alternative is you spend all week coding then have to wade through all sorts of sections of code that you need to refamiliarise yourself with for another week.

> I've seen some people who put typing speed before accuracy. They don't make good programmers.

Visual inspection

---

208 This chapter is back to front - but it's easier to understand that way. At the end we'll be looking at the concepts implemented more practically at the start. The denouement will be "how do we know what to check".@@@check denouement

Put your WCPGW hat on and work slowly through the code looking for stupidities and usual hot spots. One technique to focus the mind is to go through the code commenting each loop with how it works. Documenting code just after writing it is a bit of a change in pace (as good as a rest) which might make you think about what you've just coded.

Bottom-up testing

Everybody does it. Test the components, assemble the components, test the assembly and so on.

Extension testing...

Start with the most basic functionality then add features testing as you go. This approach, particularly suited to OO programming, has two non-obvious advantages:

- You stay immersed in your code as it develops, looking at different aspects of the same thing while having all the details fresh in your mind.
- You appreciate the strengths, weaknesses and possibilities of the unit. This means you may re-think some aspects of your basic approach or may add a few handy features while you're at it to make the unit more generally useful or robust.

Evolving a solution is not a substitute for spending time on initial design, but on the other hand you don't need to stick to a plan if it looks like there is a better option. One common 'evolution' is where you realise that there is some aspect which should be spun-off into it's own unit which will be a valuable component to reuse elsewhere.[209]

> After a while you will develop your own style of code development. Rigid compartmentalisation is unlikely to be a successful strategy... ...neither is code before design... or calling it finished when it works at last. (More answers later.)

Defining tests

Way back in chapter @@@ when we were developing a diary we did a little bit of testing. This amounted to throwing a variety of inputs at the program to see if any would cause problems. The choice of test data was made by merging the sorts of inputs expected with the sorts of exceptional inputs that typically cause trip-ups. Remember that some of the tests are 'let's see what happens' and others were 'this should go wrong'.

We had to interpret the specification and make up a context. This is typical. One other aspect is that the writer of the code will know what logic tests and loops and data structures are buried within the code and so can specify tests which look at the boundary conditions and validate the validation routines. Pop back a couple of pages to look at the string-to-date example.

- *Abuse testing*: "-1" is a valid integer and "12-1-2" could possibly be inputted as a date by a user. Will our validation routine trap this? This is an example of a possible 'illegal' input which could slip through our validation. Abuse comes in three flavours :

---

209 Don't be afraid to postpone work that will distract you from your main goal. Leave suitable markers in the code and code with future development in mind so you can pick up the threads later.

    - Well meaning but misguided or careless
    - Systematic mismatch of actual input to specification
    - Hacking. (We'll look at this in a later chapter.)
- *Logic testing*: Suppose we tested the length of the string to be 6 digits as follows (with the appropriate re-jigging of the following logic) `if(StringLength(DateString)=6){` ... Then we must definitely check for 5,6 and 7 character strings. Why? Because of the equals gotcha this might always evaluate as true.[210] (With the <> test we can be a bit more confident of this basic test working without explicit testing.)
- *Functional testing*: When the routine doesn't fail... ...does it actually work? Our string to date code has an odd way of handling years. 0 to 99 inclusive. Does 88 mean '0088', '1988' or '2088'? Can we represent 1066 or even 1966? Part of this is presumably the responsibility of the designer but we still don't know for certain that "010100" gives a valid date (whatever valid means). To do this we need to exercise the routine in a wider context where we can validate the 'correct' responses are always correct. You may be testing for other matters such as speed, accuracy, reliability and resilience. 'What matters' will vary and ought to be specified.[211]

Functional testing is sometimes called *Black-box testing* where the tester doesn't get to see the internal mechanism. This denial is supposed to be good for the soul - or something. *White-box*[212] testing is where the tester has access to the innards, possibly with no context.

Test harness and environment

To carry out functional testing (and the other types) you typically need some way to exercise your code, feed it with interesting data and look at what goes on.

A *test harness* is a special program that simulates the environment of the unit being tested. For this to work you need to provide ways for the harness to inject data and 'press the start button'. Suppose you're writing a web browser, then your environment will consist of a web server with specific pages, and a program that simulates user actions. Once you have the tools for the job you then need batteries of tests (one for each area of functionality you want to test) and a way to tell if what happens is what should happen. This can get very involved which is why you really want to make sure the components work first and then you can concentrate more on the interaction of

---

210 Some people advocate writing tests like this with constants on the left and variables on the right in the hope of triggering a 'cannot assign variable to constant' compiler error.

211 Or you may be experimenting in the hope that by understanding the behaviour of your routine better you can improve it or avoid unpleasant effects.

212 And there is a Grey box as well - Not much of a box then is it. A classic case of the simplistic metaphor substituting for real thought. The 'opposite' of Black box is either Open box or Clear box. This is the sort of anomaly that a Real Programmer ought to be spotting. NB You don't have to stop the slide show presenter in mid-spiel - you've been alerted to their smoke and mirrors approach so can enjoy the exercise of looking for the other rickety logic, glossed-over statements, opinions-as-facts and so on. Mind you, in the discussion afterwards...

those components.

A *test environment* is all the tools, data, analytical tools, specifications and result logs used for testing. For example if you're maintaining some application that is currently in use it is considered very infra-dig to test using live data. So you have a separate database which you can experiment with. If you need real data then you can copy it to another database (that makes three - 'live', 'copy of live' and 'specially for testing') to make sure[213] you're not going to trash vital data.

Tools for testing (with IDE goodness)

You already have an editor and compiler and some way of running your programs. What more could you want to allow you to develop quality code quickly?

- A well organised filing system for a start
  - With language reference documentation
  - With application documentation (specification, design, user guide, test data, development history and to-do list.)
  - With general purpose sources and code library
  - With application code (past and present)

  You don't need a complex database - the ordinary file system should be sufficient. If you are part of a team then special rules might apply.

- A more 'intelligent' code editor. Funnily enough there are a lot of people who prefer very simple assistance with their code editing, the odd bit of automatic indenting and colour coding; while others drool over editors that sense what you're going to type and prompt for the right number and type of arguments as soon as you get to the opening bracket of a function name. Other clever editor features are showing an overview of your code elements in a tree so you can drill down to objects and methods or functions in units.

  > Editors are very personal things. One man's vi is another man's Emacs. (Two contemporary editors of the 1980s) The first you have to see to understand how primitive it is, the second a Swiss army chainsaw. Both had large bands of loyal followers!

- A compiler which has various levels of warnings and possibly different modes of compiling to include debugging code during development and cleaner code for release.

- A compile/link/make build engine which knows how to construct an application from components. Typically it knows how to do only the minimum amount of work needed to ensure that modifications to code are applied wherever necessary. Sometimes this will be built-in to an IDE (Integrated Development Environment),

---

213 Well at any rate ensure nothing immediately obvious goes wrong. With any luck the 'oh dear we never though of that side effect' will only become apparent after many months when the trail back to you has gone cold.

other times you will use a utility called make¤ or one of its derivatives.[214]

- *Debugger*. If you run a Javascript program with a syntax error in it, all that happens is the code stops executing at the error. That's all. No helpful hint to say "Hey look at line 77". (Javascript is interpreted not compiled - so the first chance it gets to hit a syntax error is at run time.)

  Wouldn't it be really useful to step through your code line by line and see what's going on inside? That's what a *debugger* does. You can set places and conditions to pause, examine the call stack, view variables, continue a bit further and so on.

  > Debuggers tend to cost money or come in the paid-for IDE The alternatives though are not pretty.

- *Tracing* is logging to a file as required to trace the logic path and variable state. For example suppose you are trying to track down why a routine doesn't give the correct result. You might get key variables sent to a file at entry then at every iteration of the loop in the hope that you could detect where the logic left the rails. Quite often you will write your own trace functions to suit your environment. (We'll do this as an exercise later.)

- *Dumping* is 'printing' an object to a file for analysis. Suppose you are having trouble sorting an array of objects. Your dump will iterate through the array dumping each object in turn. The object will contain fields some of which are objects or arrays themselves and so on. You might discover that unexpectedly some field is null and thus causing something strange to happen.[215]

- Exerciser programs as discussed above - all home grown.

An IDE, Integrated Development Environment, is considered by many to be a must-have. As with editors, people have personal preferences and hates so explore the possibilities. The advantage of a basic editor is that your main programming tool is the same regardless of the language you're working on, whereas typically it is a case of a different IDE for each language.[216] With any luck your IDE will provide an environment in which to run your code in debugging mode.

> If the compiler/debugging tools are the engines of your development system then your library is the wings.

---

214 Since we're concentrating on testing here, if you use *make* or can get a log of what's changed in some other way you might be able to generate a list of items that could usefully be re-tested.

215 How do you print null?

216 But research Eclipse which is a FOSS IDE intended for multiple languages.

Small is suspect

Tiny routines are frequently assumed to work. This can lead to embarrassment. If you're not going to make an explicit test then do a very good visual check to weed stupidities. Cut-and-paste errors are common culprits here:

```
method GetMonth(){ return this.month}
method GetMonthName(){ return this.month}  // cut-n-paste error!
```

Small routines are often quickly tested for functionality, often in the main program. The theory is that if the main program works then the small routine must be also. How can you improve your confidence that you're not just being lucky?

```
function FetchMiddleElement(AnArray){
  me = SizeOfArray(AnArray)/2
  return AnArray[me]
}
```

Oh dear! An even number of elements in the array means there isn't a true middle element. Quite likely this doesn't matter but so long as the situation is under control everyone will be happy. Oh dear! An odd number of elements gives an `me` of something and a half. If that half gets rounded down the code will work (for zero based arrays) but who can tell if it will always be rounded down; possibly 333/2 as an integer is 167[217]

Reduce sources of uncertainty.

Either by testing, inspection or good programming.

- Address any ambiguities in what's supposed to happen when
- Identify 'iffy' situations that could arise during processing...
  - Operations with uncertain results (eg divide by two to integer)
  - Boundary conditions (first, last etc)
  - Anomalous conditions (eg divide by zero)
- ...and clarify what actually happens. By experiment or altering the code.

This is one of the reasons that programming with threads is so tricky: A lot of things are happening at once in no particular order. It may takes days of *soak testing* to discover a flaw when just the wrong combination of timings conspire to upset your plans.[218]

Instrumentation

For the purpose of getting the damn thing to work, or improving performance - that is in a development environment - you may want to insert extra code which records activity within the code, typically to a log file. The important thing to remember is that this must be easy to switch off for release.[219] A typical bit of instrumentation might be something like this:

```
startTime = TimeNow()  //@@@
// do some activity
endTime = TimeNow()  //@@@
```

---

217 Possibly on one computer 166 and on another running identical code 167.

218 And then all you might find out is 'it hangs'.

219 Oh and don't forget to *actually* switch it off will you - You'd be surprised how often it doesn't.

```
LogToFile(endTime-startTime . "milliseconds") //@@@
```

- The @@@'s are handy flags to search for before release
- WCPGW? Spanning midnight.
- WCPGW? Clock not very accurate for short periods. (Typical clock resolution is 50ms.)

Test data

We've already touched on this subject but there is an important aspect which needs emphasis. Having structured data, well annotated with what is being tested and what the result should be in each case, is essential. One way of doing this is to attach the test data to the code as comments. Another is to have an index of test data referencing files of data (and results) in your library.

> Structured testing is honoured rather more in the breach than the observance. There are good practical reasons for this. Whatever method you use you'll find it really handy to be able to swiftly run through your application or the modules just modified to check everything still works.

Or you might have a test harness with the data and outcomes integrated. In the following hypothetical example file (where # indicates a comment) the third item is the required result.

```
# byte addition
# AddByte(col1,col2)=>col3
2,  2, 4
0,  0, 0
255,1, ex1 # overflow exception
2, -1, 1
0, -1, ex2 # negative overflow exception
```

Now when you use the harness to fire this test data at your byte arithmetic function you should see the result "All tests completed correctly".

Or you might have a dummy application that uses all the components. You'll be amazed how quickly things stop working for no apparent reason. Make sure your dummy application uses all components and you have a written test sequence.

Repeatability

If you have read-only test data it should be easy to repeat tests.

With databases your tests tend to modify the data, so you need a quick way to restore pristine test data. This can be more difficult than it sounds because the database may be evolving and you may be trying to test for a condition that only seems to occur on the live system.

If you're testing real-time¤ aspects then you may need to simulate real-time inputs. An example of this is *stress-testing* where you might load a server with more and more clients to see how much it can take before something goes wrong.[220]

---

220 One of the signs of a good design is graceful degradation of service - as opposed catastrophic.

Record keeping and versioning

Not a lot of this goes on either! It's quite difficult to do. Large projects find themselves spending a fortune on it and small projects hardly justify complex databases. You will have gathered that if the test data is to be matched to the testing and the results verified that's a lot of different tasks. Add to that the matter of *regression testing* where you go back through all the previous tests after making alterations to make sure nothing has been inadvertently knocked out of alignment and you can see why there's a lot of short-cutting going on or else nothing new would ever get released.

> There is no one answer, and anyway there is no one problem. Things might be happening that are out of your control, or your project is small and self contained, or you need to coordinate the work of many people across many units and many organisations.

There is a generally accepted tool for tracking the development (and with it testing) of software and that is versioning.[221] You've seen lots of these version numbers which in full applications tend to be in four parts something along the lines of

1. New release with lots of added features
2. Bits we really had to tidy up but didn't justify a major change
3. Small modifications
4. Internal build serial

Some people use the file time stamp to identify earlier versions of sources. This saves a lot of fiddling about when developing code and working intensively on a lot of sources, but as soon as the code ceases to be work-in-progress it really ought to be called 'version 1' or similar to flag to any maintainer that 'version 1' is a finished entity and modifications need to be documented.

Where is the documentation? Quite a good place for the details is in the source code. There's no excuse for not maintaining it as you work on the code and it's ready to view. If that's not sufficient then you need to develop your filing system.

Milestones

Milestones are points in a project where certain goals are achieved. These are specifications for what's ready for the next stage. Obviously they can be synchronised with versions. The first milestone might be getting a barely functioning rough-and-ready system as a scaffold on which to develop the 'real' application.[222] Since your milestones matter to management, someone is bound to ask what is the state of testing.

Review

---

221 A *version* is a development of an the identical application or unit - Not 'Free', 'Starter' and 'Enterprise' 'versions' which are *variants* of a product line. It is worth trying to keep the time-wise (version) and function-wise(variant) separate in your mind, even if they are, possibly quite rightly, merged in a development environment versioning system

222 I'm not recommending this approach. It might be appropriate - or lead you astray.

Bottom-up testing is the norm. Having reliable components is a really great efficiency boost. When altering a component you need to know what assemblies it is used on. Sometimes, with general purpose utilities this is practically impossible to trace, but if possible use some build engine to keep track of these dependencies in order to bubble-up the 'needs testing' flag.

Developing test data is a bit of art and bit of science. You need to think about two separate things: Checking how the internal mechanism works and what sort of external world the code will be expected to deal with.

A suite of test tools will never be far from your side. You may well find you have to develop your own. This won't happen overnight. Don't forget your filing system will save you hours if it is well organised and maintained.

Testing for functionality

So far we've tended to think in terms of 'where are the bugs in this code' rather than 'does the program do what we would like it to do'. Often throwing test data at a program isn't enough, it has to be tried out 'for real' to see where mis-understandings have arisen, unthought of snags appear and false assumptions cause some chewing of beards.

> Remember, most people are not very good at working with abstract visions. As a programmer you're trained and experienced but many people need to see a mock-up before they can begin to relate to it and have a meaningful discussion.

> This is where Real Programmers pat themselves on the back. You have read between the lines of the requirement and planned for the unwritten needs and bonuses inherent in the job. Then you spent time putting in the necessary hooks and allowing room for enrichment. Now the customer is amazed at your foresight and comes to respect your brilliance and competence.

Prototyping

Getting a demonstration system in front of a customer early on is often a very good idea. The main reason is that you can develop direct communications with the customer and hopefully users. This will help you understand what matters to them, where the difficulties are likely to lie, how simple you need to make the user interface and so on. The customer feels good when they see something and begin to realise how this new tool might be used in practice. You learn the customer's jargon and get their cooperation to provide you with some realistic test data.[223]

Cosmetics count

Would you buy and cherish a collection of bits patched together or an ugly machine with sharp edges and an annoying whine? No of course not if you had the choice.[224]

---

223 Easily promised but don't expect too much.

224 So it's strange people still use Internet Explorer.

But while you are beavering away at programming, paring your code to the bone for performance and enhancing the interface with all manner of sophisticated options, you fail to notice that your application is now as pretty as a junk yard and friendly as the control panel for a nuclear reactor.[225]

Check the following. Get more than one opinion.
- Appearance
- Readability
- Navigability
- Speed of use
- Ease of learning

Remember that different people have different views of what's good. They also have different hardware. For example, I happen to use a very old and obscure proprietary email and news reader. The modern alternatives all seem the same and not half as good for what I want. They will all read and send news and email but not in a way that I find convenient.[226]

Beta testing

Alpha testing is before customers get their hands on the application. Beta testing is when you've given up trying to find bugs and invite customers to see if they can do better.

Let's just say that there will always be a period where a complicated new device needs to settle down and have the wrinkles ironed out. (This applies to all engineering.) How you manage that process is a moot point. There are two fundamentals:
- The 'boat must float'. Critical failures will be extremely embarrassing.
- There must be a defined relationship between supplier and customer.
  - Common goals
  - Communications channels
  - Realistic expectations

A typical arrangement is to agree to a period of running a pilot, training staff and loading data during which time the software will be tweaked and bugs fixed. This will be a stressful period so make it as short as reasonably possible and whatever you do make sure you've done the best possible internal testing job first.

Separate critical, desirable and cosmetic issues so that you can prioritise your work. You'll need a way to provide reliable upgrades. Think through and document the complete update procedures. Typically some upgrades will be no more than copying a file or two and verifying that the users are actually using the new version of the program. Or you might have to synchronise updates with changes to a database.

---

225 I was discussing a design detail of a technology demonstrator boat with the chap that had designed, built then sailed her round the world. "Wouldn't it save a lot of work if..." "Yes" he replied "But I had to have some features to make it look conventional - not what I call an 'Inventor's boat' "

226 Also I can hack the old program which gives me satisfaction.

Listening

Having a good relationship with your customers is a good way to discover what sort of features they are prepared to pay for in a second version. You might be amazed that when things go wrong users don't tell you. Real Programmers are good listeners, they will work at it in order to be ahead of the game. You will appear as someone in whom customers and users can talk to in confidence in order to get to the 'real story'. This makes a huge difference to the utility of your programs.[227]

Review

The customer wants what they want - not what they originally said they wanted or what you (or your team) thought they said. Part of that is cosmetic, part is knowing that specifications are seldom as complete as they appear (or have false precision), part is coaxing them through a difficult transition period with clear instructions and a friendly bug-free[228] interface.

Giving the customer what they want is a process that starts long before any code is written and *continues throughout the development*.

Quality assurance

Quality assurance is normally a paper-trail[229] which gives more credibility to the statement "this code works". Let's get something straight: You don't need ISO wotnot or quality engineers crawling over your 20 line program that plays happy birthday on a given date or your personal blog. You still need to do some checking, and if there's an embarrassment then the egg is on your face alone.[230]

For big projects programmers are just small cogs in the machine and shouldn't worry about QA administration.

What about the smaller project where one or two programmers lead a team or work alone? Here the operative word is "lead". Ultimately the quality of the finished product is down to you.[231] If you have collaborators then they need a system to share information and allocate tasks to support your efforts.

---

227 You also need some self-discipline in order to avoid promising anything or rushing off to knock-up some code 'because it seemed like a good idea at the time'.

228 There's no such thing as Bug Free¤. Just keep clear of crippling faults and unsuitable design.

229 Another method is "1,000 people use it OK so It can't be that bad".

230 Small is suspect! Run the HTML checker and spell checker on the blog as well as checking the content for WCPGW?

231 You will be the most highly trained and intellectually engrossed person in the team. You will have a far greater knowledge of details, design objectives, tools, policies, the way the team members interact and the current situation than any of the others. Real Programmers will be professional leaders in all technical matters.

OK - Enough! In all but the tiniest or largest projects put your filing system to work. There are project management tools such as to-do lists, bug tracking lists, test schedules and version control systems which you can explore. You probably don't *need* these development assistance tools as well as a good filing system, but if you find some that turn out in practice to be useful then make the effort to find out how to get the best out of them.

Real quality

We need to understand Real Quality systems for two reasons:

1 To apply it to our own development process
2 To build quality into customer's procedures

Competence : Bad-Good-Best[232]

There is Good, Bad and Best practice. You'll often hear "best practice" talked about but what the speaker really means is 'good'. Good is OK. Bad is less than OK. Best is exceptional. Bad is unacceptable. Good is achievable by most people most of the time. Best is well above average.

A Real Programmer will definitely be in the Best category... ...unless they're showing signs of human failings such as drinking too much, in which case they become Bad.

> Many readers will consider themselves happy to be in with the majority mediocrity. If you've got this far then you're well on the way to being well ahead of most of the field. All it takes is another few years of practice...

Bad behaviour is moderated by a rule book. Good by standard manuals and default training. Best requires personal commitment to further training, research, projects and experience.[233] Typically the Best people take a wider interest in what's going on around them and will participate in some way in communal activities. See BGB appendix. @@@

Risk

B-G-B relates to human competence. Inanimate systems are classified in terms of risk and consequences.

- *Risk* is the chance of something going wrong.
- *Damage* is the consequences of a failure to prevent the risk becoming a reality.

Diligence is the level of attention needed to protect against a risk.

Of course computers don't 'pay attention' so *diligence* isn't applicable... ...but *reliability* is the equivalent for a program.

> Making a *program* very *reliable* requires a high level of *diligence* by the *programmer*.

---

232 This paragraph might seem a bit out of place but it contains one of the most important ideas in the whole book with ramifications in all professions.

233 It isn't necessary to have formal qualifications in computing. If you're the sort of person who likes formal courses then look upon their value as being the structured approach and putting emphasis on details rather than certificate at the end.

If you're designing and building systems for people who are performing high-risk high-diligence tasks then one of the things you can do to help them is to make extra efforts in your programs for spotting mistakes or redesigning the user interface to make the whole process more reliable.[234]

How can you, as a programmer, tell where the high-risk, high-reliability code is so you can put extra effort into being absolutely sure your code is fireproof?[235] Somebody tells you or you find out for yourself. This is a job for the system designer.

- They will try to design-out high-diligence-high-risk tasks. If that isn't possible will be putting safety nets and double checks into the specification.
- They *should* tell you about high-damage cases of WCPGW. Should, but you'll have to be following along closely behind with your own WCPGW antennae twitching. For example sorting and searching for names like McTavish/MacTavish is often dealt with by having a policy of assuming users can make their own bed and lie in it, or alternatively silently force one form to be used. WCPGW? What if the ambulance control centre is trying to find "`Mace Road`" and your system is 'helpfully' converting that into "`Mce Road`"?[236]
- Deliberate abuse, which we'll look at later, is also a joint responsibility.

So now you've got a reason for testing. You can apply your finite resources to where it matters most. Your testing of components was looking at 'does this code operate as specified' in order to be reliable enough to be used in applications. Your functional testing adds the criterion "Is this application safe".

Built-in system-level quality

The appendix @@@??? explains how built-in quality works. Your programs are part of the whole system and can be used to

- enforce rules
- split complex tasks into separately checkable parts
- ensure that communications work as they're supposed to
- spot anomalies
- collect statistics and samples for quality management.

Part of this is the system designer's task but Real Programmers will be reading between the lines in order to design the program details. For example suppose your program writes a letter referring a patient to a hospital. Naturally it keeps a record in its database along the lines of who, what and when. Is that the end of the story? WCPGW? Letters get 'lost in the system' sent to the wrong person or simply filed. So

---

234 Something to think about (before we get there in a later chapter(@@@ check)) - What are the reliability and diligence issues with user logins?

235 Umm - well. 'As sure as you can be'.

236 Let's all be worried about increasing centralisation of emergency control centres. Not only do they have an awful track record of being over budget and getting scrapped when the simply don't work, but local knowledge is being lost. One morning I phoned the police Me: "Late yesterday evening..." Pol:"Why didn't you phone us then?" Me: "Because you'd get lost" Half an hour later: Pol:"Where did you say?" Pol:"Oh yes it's *there* on the map - and we're *here* - Sorry to trouble you again."

your programmer's reflexes fire up and you add in a method of spotting if nothing seems to be happening to the case.[237] In a related case I wanted to ensure separation of Observation, Decision and Action. The paper form with all parts jumbled on the same page lead to very poor quality communications, but on the screen I could implement a three-step wizard to enforce the distinction.[238]

Review

People who employ programmers seem to think only in terms of specifications, lines of code and bugs. The WCPGW programmer's mentality doesn't register with them. In one way this is strange because without it programmers are a liability. Even the average programmer has some WCPGW flowing through their veins. In another way it isn't so strange: They're employing programmers to continue making the same mistakes and taking the same risks as are enshrined in the existing system.[239] Because you will have twigged how to build-in quality to their systems and have a very clear model, preferably in a well presented report, they get very nervous that an outsider can so confidently grasp what this quality business is all about without swamping everyone in paperwork.[240]

Until you have seen it, you have no idea of the resistance there is to the idea that there's a logical, professional, less risky way to do things. To start with this is just clinging to a comfortable traditional shambles. Then the brighter ones realise that if you insist they split the 'what's happening' from 'what's wrong' and 'what should be done' these things will be checked from time to time and form part of a decipherable audit trail... ...Which might lead back to ... them.

When you appreciate how quality, which is really only a word that means, 'doing what we want to do' depends on knowing what you want to do from the very start and then applying a bit of risk spotting, ie WCPGW, then you'll be building-in quality to your code as you go. You won't write code *then* think about testing it, as you'll have made a note of what the critical inputs and situations are either as you go or before you start. You'll still have plenty of bugs in your code that need exorcising but you'll be lying in wait for them - not the other way round.

---

237 You probably have no idea how ramshackle communications are in the NHS. Be frightened! The people who work in the system just treat it as a fact of life - there's nothing they can do so why worry? Real Programmers could do something about it if the organisation wasn't so badly managed from the top down. This isn't just a NHS thing - Poor communications are endemic - and often made worse by electronic systems. On a personal note: Your diary should have quite a few 'chase foo' entries.

238 What a stink that caused! "We're professionals so we should be allowed to write any old rubbish". "It's worked up to now" - Yes right - for values of 'worked'.

239 With some extra exciting new ones just to make a change. Some managers just love to show off their abilities in a crisis. Others wouldn't spot a problem if it landed on them from a great height.

240 If somebody is threatening to send you on a 'quality' course - insist on a full course guide before starting. Then apply whatever methods (if any) are promoted to the course itself. Some are an abysmal rip-off and don't even do administration properly.

I hope you forgive me for writing this chapter back to front. The easy, mechanical bits came at the front with the difficult bit at the end, You'll have to do a lot more detailed research into the tools that are appropriate to your particular development environment, and it takes a long time to develop your testing skills with a variety of complex aids. Not withstanding this, the quality of your code in its usefulness and reliability (remember the two aspects - the function and the mechanism) depends entirely on your determination to do the job well. ie diligence.[241]

241 Don't be suckered into doing all the testing for a group. How boring would that be? How efficient would that be? If you get a reputation for reliable code and having a well organised approach to testing then pass on your skills. Don't take on the responsibility for clearing up other peoples mess.

14. Code interlude

How about some hands-on? This chapter is a look at the practice of programming during which a lot of the issues discussed in the last few chapters will be put into practice. You could call it "a day in the life of a programmer".

It's traditional for apprentices to build some of their own tools. This is what we're doing here.

Coding

Try to develop code with pencil and paper and YCPL step by step. You'll need to interpret what I'm writing here in the context of your own development environment.

I'll be using the PHP language[242] which should be easy enough to follow. Practically the only things you need to know about PHP are:

- Variable names start with a $ sign and are not case sensitive
- Variables are not typed so that the code `if($a==TRUE){$a=5;}else{$a="five";}` is perfectly valid.
- Arrays are dynamic and associative. So if we executed the following two statements `$a[64]="x";` and `$a["y"]=10;` then listed the array contents we'd have an array of two key/value pairs.
- Strings can be single or double quoted. If double then variables are automatically merged in so $msg="Hello $userName" is valid.
- Object fields and methods are referenced with ' `->` ' for example `myDate->GetDay();`
- Other language specific points are commented in-line.

Objective

In the last chapter I mentioned one debugging method of writing a trace of internal events to a log file. This works by adding calls to some logging function inside the code. What we could do with is a convenient method of capturing, recording and displaying this information.

Dive right in

Thought...

- The key executive action will be something like:

```
function Trace(SomeText){
  WriteLineOfTextToFile(LogFile,SomeText);
}
```

- We need a way to open the log file before writing to it
  - name?
  - append or overwrite?
- We must close the file whatever happens
- If we're writing text we can use any editor to view the results.

---

242 Version 5

...Code

How many lines of code will that be? Estimate : Trace function 3, open file 5, close file 3. Total less than 20. We might as well just type in the code off the top of our head as it can't be much simpler.

ProgBook/Ch14/v1/tracing.php

```
<?php                              // all PHP code starts like this
function Trace($TraceFile,$TextToTrace){ // fwrite is file-write
  fwrite($TraceFile,$TextToTrace."\n");  // dot is string concatenate
}                                        // \n is newline character

function StartTraceLog($LogFileName){    // fopen is file-open
  $traceFile=fopen($LogFileName,'w');    // w for write new
  return $traceFile;                     // a file handle
}

function EndTraceLog($TraceFile){
  fclose($TraceFile);              // fclose is file-close
}
?>                                 // all PHP code ends like this
```

ProgBook/Ch14/v1/testtracing.php

```
<?php
include('tracing.php');        // Insert the tracing functions
                               // automatically globally available
print('Start<br>');            // something to show on screen

$tfile=StartTraceLog('logfile.txt');
$x='foo';
for($i=1;$i<5;$i++){
  Trace($tfile,"$i $x");       // variables will be substituted
  $x = $x . $x;                // dot is concatenate operator
}
EndTraceLog($tfile);

print('Finish');               // something to show on screen

?>    // all PHP code ends like this
```

When this testtracing.php is run[243] it displays Start and Finish on the web page and creates a file called logfile.txt with four lines.

It works - Hooray! ... Job done. All down the pub... Err... Is something wrong?

> Dear reader, the reason I'm writing this book is so that one day this bletcherous style of programming will be a thing of the past.
>
> If you're an established programmer you might want to read the next bit in private - there might just be the odd bad habit of yours that gets savaged. (My ears go a bit pink too when I think back over some better-forgotten episodes.)

[243] For which you need a web server configured to interpret PHP scripts. If it's easy to do then have a go, otherwise it isn't worth the effort.

Let the bletchfest[244] commence

Yes, lots of things are wrong. Do this now.

0. With the above two code files
1. Write a list of good points
2. Write a list of bad points

It's important you have a go at this exercise before looking at my analysis because it is a way of gauging your own progress in the art of good programming.

OK, let's see what the good points are:

- Quick to write
- Simple to understand how code works
- Standard[245] naming convention used
- Clear layout
- Files are obviously located in an organised filing system

These points make the code useable but that's about it. Definitely on the cowboy side of workmanlike.

There's a great deal wrong on a number of levels. There are missed opportunities, poor user interface, numerous error conditions, absent documentation and haphazard testing. Here they are in more detail:

Coding standards

- There is no identification of what the files are written into the text.
  - No name
  - No version, date, author
  - No title or short description
- There are no comments to explain each function or the way the test operates or what we should expect to see as the test result. (The comments in the code are only to explain PHP language elements to non-PHPers.)
- It is normally best to put functions and methods in alphabetical or in logical groups.

> Clear layout and clear explanations are vital during development, during use and during maintenance.

Argument checking

- Trace() requires a file handle as it's first argument. What happens if we call Trace() before StartTraceLog()? Something unpleasant. What else CPGW with this argument to Trace()?
  - StartTraceLog() fails so $tfile is undefined

---

244 Bletcherous is computer jargon for something (not somebody) that's disgustingly cretinous - But I expect you guessed that already.

245 My standard. Yours may be different.

  - Trace() is called after EndTraceLog()
  - The $TraceFile argument is supplied as a filename by mistake
  - The order of arguments to Trace() is reversed by mistake
- The same applies to EndTraceLog().

> If there's the slightest risk of anomalous arguments then check them as soon as possible.

Error trapping

- fopen() is prone to failing for all sorts of file-system reasons. Perhaps the file is already open, or the filename supplied is bad or we don't have the necessary permissions. But there's no attempt at trapping these conditions, let alone reporting the issue so something can be done about it.
- The same applies to fwrite() and fclose().

> - In general, trap exceptions close to their source.
> - If the situation isn't completely dealt with then be clear how the event or consequences of the event propagate.

Result checking

- When the test program calls StartTraceLog() the result is assumed to be a file handle. (A comment in the code would be nice.) StartTraceLog() might fail with a crash or perhaps silently. In the latter case we've got an undefined variable $tfile waiting to clobber us. Since we know StartTraceLog() can easily fail we should be taking a bit more trouble to check to see if we've got a live file handle.

> Don't rely on unreliable functions

Haphazard testing

- All we've done is see if we can get the functions to work as we expect them to under ideal circumstances. Cars get put through crash tests to make sure they perform under abnormal conditions so should your code. In this case we don't have much to go on because we've not thought through what's abnormal and how we might test it.
- We've no guidance about what to expect in the log file.

> *When you've got it working*
> *then you start testing.*

Documentation

- There is none. How is a user (possibly yourself late at night, under pressure) going to tell in 10 seconds if this is the right tool and if so in 30 seconds how to deploy it?
- How does somebody maintaining tracing.php know there's a test program and vice versa?

With good coding standards a lot can be done by commenting the source and diligent

use of filing system.

> If you can't remember it (and believe me, you can't) then you need the documentation.

Poor user interface

On the face of it you might think the API¤ is nicely minimalist and practical - those signs of sparse efficiency beloved of every designer. So is a brick!

- Why is the user being asked to look after $tfile? So they can hand it back to Trace(). • Can't we find a way of 'black-boxing' it, so the user doesn't have to worry about providing it?
  - $tfile will have to be accessible throughout the code wherever it's being used for tracing, probably deep inside methods. That's going to make it a bit more difficult to keep track of.
  - It would be easy in Trace() and EndTracelog() to mistakenly use the *name* of the log file rather than the *handle*.
- We didn't address the issue of overwriting or appending
- There's no timestamp written to the logfile. It would be very easy to repeat a test but look at an old log file[246]
- We've not provided any facilities for
  - Logically formatting the logged text. It's quite likely that we want to report where we are in the program followed by variables which may not be text.
  - Switching tracing on and off. Typically we might want to catch only certain parts of our instrumented code. We certainly want to be able to switch it all off without physically removing if from the code.
- It would be easy to forget StartTraceLog() or EndTraceLog() or think the code will operate with them but in practice is somehow bypassed by logic, exceptions etc.

> Ease of use trumps ease of coding every time.

Missed opportunities

- Logging is likely to be a useful facility beyond debugging.
- It is going to be handy to look at complex objects as well as simple variables. We could use the trace as a way to dump variables for inspection.
- Instant screen display might be a useful alternative or addition
- We may need to limit excessive output or highlight particular items of interest.
- If we had a test(works) program we could use that as an example for users. We'll exercise all the facilities so that will be ideal as a learning resource. We'd keep the test(crash) program for ourselves.

---

246 In at least two easy ways: Changing the directory the file is being written to so not overwriting the old one. Failing to refresh the editor view.

> Synergy is the key to efficient reuse - more for less

Review

That was rather a demolition job wasn't it! You probably ought to write the morals out in cross- stitch or poker-work and hang them over your bed. The big moral is

> Be broad with your ideas,
> tight with your technique,
> code for completeness.
> Build your reputation
> on the quality of your programs.

What's just been shredded is only very slightly 'below average'. Add few comments and that's about the normal standard. A little bit of practice and you'll be streets ahead.[247]

In a moment we'll have another go. You might like to spend a couple of minutes sketching out what the main good points from the user's and developer's points of view might be.

Second attempt

Do you recall when we were discussing quality that we had a split between the *mechanism* and the *function*? We can use exactly the same split between the *How* and the *What* when outlining our program design. (That's the same as the split between user's and developer's issues in the previous paragraph.)

If you were a clockmaker the *How* would be the details of the mechanism. The *What* would be the style of timepiece, would it have an alarm, be compact or illuminated. The *whole* would be matching the *How* and the *What* economically and reliably.

Let's use that as a guide for our first thoughts.[248]

---

247 When you're always on-time there's nobody *else* to appreciate it but at least you have an inner glow of righteousness. But when you code right first time even you don't realise the trouble you've saved. There will always be the temptation to take 'short-cuts' ... ...see what happens when patches start fraying at the edges and learn. One of the signs of an elite is that they can produce excellent solutions quickly without resorting to sloppy technique.

248 Don't be afraid to re-think your first thoughts. Books like this one are deceptive as they get to the finished result without going round loops and up dead-ends. It's a fiction - My creative processes are probably more eccentric than yours. (The real skill is spotting the good and culling the bad which will come with practice.)

First thoughts - What

- We're trying to implement a special type of log file
- Trace needs to be able to report any variables.
- Trace needs to be switchable. Log is more of a fixed thing.
- Log files may need special naming such as yymmddhhmm...
- Logs will often need timestamps
- Display: Immediate? HTML? Text? Formatting in columns?
- Log to database?
- Will logs be scanned visually or processed automatically?
- Can multiple programs/users use the same log?
- Trace could do with options to be selective
- Trace needs to be easy to use for instrumentation. Logs can be a bit more involved.
- Users will be looking at log files - may need pretty formatting
- Utilities for weeding old/bloated logs?
- Checks to avoid too many logs being opened by error?
- Log files need closing at end of program (if not before)

Notice the danger of *Creeping Featurism*. Hey! What about a server for all logs on a system and a log-control-language and search tools and alert tools and audit trail reports. At some stage you're quite likely to log important actions to an audit trail in a database - but perhaps that's something for another day.

First thoughts - How

- Trace sub-classing Log
- Filename can be internal to the log, but specified in constructor
- Class methods[249]
  - Build filenames
  - Timestamp formatting
  - Possibly organise log paths
  - Possibly other file management utilities
- Maybe we should have a class hierarchy of
  1. Log - Basic text logging with timestamp and filename functions
  2. HTMLLog - Add framework to produce HTML eg tables logs
  3. Trace - Switchable, variable dumping with lots of defaults
- Logging to a database is probably getting too sophisticated and esoteric
- When opening files for writing we ought to have some way of dealing with inability to open because another copy of the program is already logging. (Or the log file wasn't properly closed on the last run.)
- We could do with a method for deleting a log file and starting afresh. Also one for renaming and starting afresh.
- Two possible approaches:
  1. Write log to memory then write to file in one go at the end. (Any memory constraints? Do we need a permanent record as we go - say in the lead-up to a crash.)
  2. Have permanently open file and write line by line. (Already demonstrated in version 1.)

---

249 **Functions which are not methods inside a class. We haven't discussed these yet but the upcoming code will be an opportunity.**

Can we usefully combine these methods or offer them as alternatives?

The *How* follows the *What*. We're already looking at the value of sophistication and making judgements.

Review

All this takes a lot longer to type than to think about and sketch with pencil on paper.

Notice how the *How* resembles the bottom-up design approach and the *What* resembles the top-down design approach. There are similarities but in a minute when you get to design you'll be working exclusively on converting the How thoughts into a design.

> **Thought for the day**
>
> Engineered objects have 'blueprints' suitable to be passed to a constructor.
>
> Art objects often 'get made' with very little documentation. The prototype is the finished article.
>
> And computer programs?

Design

It might be worth quickly revisiting the design work we did in @@@, then having a go at writing your own blueprint. As you'll be the one building the programs and writing the final documentation you won't have to be particularly detailed, but you must at least be able to list the components and describe their contents.

Top down

- Three classes in a hierarchy: LOG ... HTMLLOG ...TRACE
  Leave HTMLLOG as a shell for now, we can enrich later when we've got a better idea of *What* we want to do in that area.
- We can add features to LOG that might be useful in a general context later.
- The extra features TRACE adds to LOG will be
  - Simplification of setup
  - Ability to switch on and off
  - Ability to dump variables (not just text)

Bottom up

1. How will files be named and located?
2. How will we ensure any open files (or other resources) are closed at end?
3. How will we deal with 'cannot open log file'
4. How can we have a globally accessible log file. (How will code operating inside a black box be able to 'see' the log file?)

1 How will files be named and located?

Do we leave naming of log files entirely up to the user or should we provide defaults which can be overridden. Do we have a default logfile directory? Some logfiles will be named with dates - how can we provide that functionality?

> Class methods:
> • Ordinary methods are functions called on an instance of an object.
> • Ordinary functions float in system-space unrelated to any object or class.
> Class methods are a hybrid. They are defined purely in the context of a class. They do not relate to instances of objects.
>
> | | |
> |---|---|
> | Ordinary function call | DoSomething(); |
> | Ordinary method call | InstanceOfClass.DoSomething(); |
> | Class method call | ClassName.DoSomething(); |
>
> Many languages support class variables as well as methods.

- Here's an idea. Have the current directory (whatever that might be - we may be able to execute code but not write files to it - we need to explore this situation in case users come across it) as the default but then allow it to be changed for all subsequent openings of log files.
- We could give automatic substitutions of date/time in special forms of filename. For example "[yymmdd]foo.log". What would be the easiest? What about giving substitutions for the name of the running program as well? eg "LoggingFor[AppName]"

2 How will we ensure any open files (or other resources) are closed at end?

This could be a bit of a poser. We can close a file (free memory buffer etc) as part of the destructor[250] or explicitly get the data written to file and close it with a call but what happens if the application crashes? (That might be why we're tracing in the first place to see what on earth is so very wrong.) We could do this with some exception trapping at the top level... ...But see 4.

3 How will we deal with 'cannot open log file'

Do we halt the application if this happens or proceed without logging? As tool makers we can't really say so we need to give the choice to the user.

4 How can we have a globally accessible log.

Some languages make it very difficult for objects to see outside their scope limits.[251] This is going to make it difficult for a method to log it's events unless we can find some mechanism to bypass this restriction.

This is probably the most difficult part of the exercise. Different languages have different restrictions and possibilities for circumventing them. Objects are designed to

---

250 As you expect, a method called when an object is destroyed. Normal variables within an object will automatically get destroyed but other resources often need special action and so a special destructor (that overrides the default) is needed.

251 As opposed to those, regarded as a bit dangerous, which have practically universal global scope.

be self-contained with only limited access to a few system-level features. There's a clue in that last sentence: If objects can't be shared what about *classes*? (Remember that a class is a definition while an object is an instance encapsulating private data.)

Notes

1 Very quick to grasp user documentation needed for TRACE

2 It looks like we're going to have a package containing:
- End user programs (3 - 1 for each class)
- End user documentation (3 - could be rolled into 1)
- End user example code (3) Could be same as test (exercise) code
- Own crash testing code (various)

3 The nature of this application is that it should have a long life and wide application. Therefore we also need to expect some maintenance.

Ready to code?

We haven't yet fully resolved a number of things:
- How to access logging functions from within objects
- Logging to File/Memory

But we're not going to get much further without some concrete code to work with.

The answer to this conundrum is to build a *prototype* to prove the concepts we're trying out. There will be all sorts of neglectful practices going on to rattle off the necessary code but just at the moment, providing we know it's for experimental purposes only, we're not particularly bothered.

Here is version 2 consisting of just two files very much like version 1 except there are dozens of lines of code instead of the original handful. (PHP specific notes are given after the code.)

ProgBook/Ch14/v2/ClassLog.php

```
<?php
//  Log - Proof of concept
//  The principle is to add any text we're given
//  to an array in the session


define('LOGARRAYNAME','LoggingArray');
define('LOGENABLEDFLAG','LoggingEnabled');

class Log{
// Issues with enabling/disabling logging
// Assume by default OFF so that it can be
// used for instrumentation and switched
// as required.

  public static function StartLog($ForceEnabled=TRUE){
  // If ForceEnabled is false then EnableLog() must
  // be called previously for anything to happen.
    if($ForceEnabled==TRUE){
      self::EnableLog();
```

```
    }
    if(self::IsLogEnabled()){
      if(self::CurrentlyLogging()==FALSE){
        $_SESSION[LOGARRAYNAME]=array();
      }
    }
  }

  public static function CurrentlyLogging(){
    return isset($_SESSION[LOGARRAYNAME]);
  }

  public static function ClearLog(){
    if(self::IsLogEnabled()){
      if(self::CurrentlyLogging()==TRUE){
        $_SESSION[LOGARRAYNAME]=array();
      }
    }
  }

  public static function LogText($Text){
  // main call to log text
    if(self::IsLogEnabled()){
      if(self::CurrentlyLogging()==TRUE){
        $_SESSION[LOGARRAYNAME][]=$Text;
      }
    }
  }

  public static function GetLog(){
  // return array of text
    if(self::CurrentlyLogging()==TRUE){
      return $_SESSION[LOGARRAYNAME];
    }else{
      return array();
    }
  }

  //@@@ we could put a write array to file
  //@@@ method here.  Options?

  //@@@ we ought to have some tidy-up
  //@@@ routine as we can't have a destructor


  // --- enabling/disabling logging ---

  public static function IsLogEnabled(){
    $rv = FALSE;
    if(isset($_SESSION[LOGENABLEDFLAG])){
      if($_SESSION[LOGENABLEDFLAG]=='Y'){
        $rv = TRUE;
      }
    }
    return $rv;
  }

  public static function EnableLog(){
    self::SetLoggingSwitch(TRUE);
```

```
  }
  public static function DisableLog(){
    self::SetLoggingSwitch(FALSE);
  }
  public static function SetLoggingSwitch($OnOff){
    if($OnOff==TRUE){
      $_SESSION[LOGENABLEDFLAG]='Y';
    }else{
      $_SESSION[LOGENABLEDFLAG]='N';
    }
  }

}// end of class Log

?>
```

ProgBook/Ch14/v2/TestClassLog.php

```
<?php
// Testing proof of concept with class Log
// 1 - OK to use class methods per se?
// 2 - CMs accessible from inside object


require_once('ClassLog.php');
session_start();


// -------- 1 --------------
// o Start and clear log
// o Write a couple of lines
// o Get log as array and dump to screen
print('Start<hr>');
Log::StartLog(TRUE);
Log::ClearLog();
Log::LogText("Hello");
Log::LogText("World");
print_r(Log::GetLog());
print('<hr>End of part 1 of 2<br>');

// --------- 2 -------------
// o Define minimal class
// o Instantiate
// o Append to log from within object
// o Get log as array and dump to screen

class FooBar{
  public function Fox($SomeText){
    Log::LogText('Inside method Fox');
    Log::LogText($SomeText);
  }
}

print('Start part 2<hr>');
$fb = new FooBar();
$fb->Fox('POC works!');
print_r(Log::GetLog());
```

```
print('<hr>End of part 2 of 2');

?>
```

PHP language specific notes:

- `define(...)` is PHP's way of establishing constants.
- Function arguments with the form $Foo="Bar" indicate default values if the argument is not present when called.
- `$_SESSION` is a system-level array available on a per-client basis if set up with `session_start()`.
- `require_once()` is a directive to the PHP interpreter to fetch the named file into the code at this point.
- `Foo::Bar()` [252] means the method Bar() of *class* Foo.
- `$foo->Bar()` means the method Bar() of the *object* $foo.

Notes on the proof of concept code

The overall concept is a little different to V1. Lines of text are being added to an array rather than written to file. (Arrays in PHP are really key/value lists. This is a really easy way to implement a *text buffer* which is a nice thing about this particular language.) The writing to file can be done as a separate exercise. This change means we don't have to write to a file at all if all we want is a report at the end of the program, or we can explicitly write the whole thing or we can write the buffer to file whenever we feel like it. So that gives flexibility... ...A good thing but a challenge for giving a fast briefing on use.

All the methods are class methods (indicated in PHP by the keyword `static`.) This ruse allows an instance of the FooBar class to access them. At present all methods are given a *visibility specifier* of `public` meaning we don't care what other code uses this method. YCPL will have its own visibility specifiers. You can see how in PHP these class methods are called.

> Visibility specifiers can be a little slippery. YCPL's will need long and careful study... ...Other languages may have overtly identical but significantly different specifiers.

There's no constructor or destructor.

Comments are sparse because we are not attempting production quality code at this stage.

I've used two flags to see what to do when being asked to log some text.

- Has logging been enabled
- Are we actually logging - have we started or do we need to initialise an array

The first test is a switch on/off which allows the logging code to be left in the source but disabled. The second allows us to initialise the text lines buffer if it hasn't been already.

---

252 "Paamayim Nekudotayim" if "double colon" is too prosaic for you.

Notice how there are plenty of user-friendly versions of calls that are really ways of doing the same thing.

I've noted in the code some additions but left them hanging for the time being. It is far better to write this sort of thing down than only think of it again after the production code has been produced. There is a temptation when converting prototypes to forget to add in the bits that were deliberately left out of the prototype.

At this stage we're testing to see if the concept will work so quite a trivial test program will suffice. It's in two parts :

- Will the calls work normally?
- Step up the challenge by calling from within an object.

Ready to produce the final version?

Tempting but definitely no.

Essential review

We need to review what we've just produced because there ought to be issues arising and matters that have been highlighted in the course of turning a collection of ideas into concrete code.

- Suppose we could find a way to store the text array in the class: Then we wouldn't need to mess around with that (very handy but one more complication out of our control) system-level $_SESSION array. We'd need something like a class method but a variable.
- The same would apply to other state such as the on/off flag.
- Why are there so many methods for switching the on/off flag? Do we need them? Are the users going to have any trouble if we excise EnableLog() and DisableLog(). I shouldn't think so. Also it is a single command call for them to get to grips with rather than three - so that should be easier.

A quick read of the manual to refresh our language knowledge[253] reveals that we can have *Class variables*. This might strike you as a bit strange - surely a class is an abstract collection of methods so having data inside a class is peculiar to say the least? Very useful though if used in moderation as we are about to find out. As there's only one class there's only one 'instance' of a class variable. (If we needed more than one log at a time we could implement a list of logs - still a single data object but with distinct sub-elements.)

> In this particular application there will never be an instance of Log. Often it is possible to mix class methods and variables with ordinary methods and fields. For example we might have a User class with instances containing user details and a few management functions such as keeping track of how many users are logged on at a particular time.

253 I've refrained from thundering on about the importance of reading the manual, following tutorials and continued research because if you can't cotton on to the importance of knowledge for yourself then shouting at you won't make any difference. However texts vary a great deal in their quality, currency and appropriateness so feel free to sample even more sources. Finally: Tools you may read about are not to be judged on 'importance' or 'fashion' or claims that all the top guys use no other - but simply utility.

Perhaps we should cull those unnecessary switching routines.

Second prototype

ProgBook/Ch14/v2a/ClassLog.php

```
<?php
//  Log - Proof of concept - Variation 1
//  Try keeping all flags and data as static variables

class Log{
// Issues with enabling/disabling logging
// Assume by default OFF so that it can be
// used for instrumentation and switched
// as required.

  private static $loggingEnabledFlag = FALSE;
  private static $loggingArray = array();

  public static function StartLog($ForceEnabled=TRUE){
  // If ForceEnabled is false then SetLoggingSwitch(TRUE) must
  // be called previously for anything to happen.
    if($ForceEnabled==TRUE){
      self::$loggingEnabledFlag=TRUE;
    }
  }

  public static function ClearLog(){
    if(self::$loggingEnabledFlag){
      self::$loggingArray=array();
    }
  }

  public static function LogText($Text){
  // main call to log text
    if(self::$loggingEnabledFlag){
      self::$loggingArray[]=$Text;
    }
  }

  public static function GetLog(){
  // return array of text
    if(self::$loggingEnabledFlag){
      return self::$loggingArray;
    }
  }

  //@@@ we could put a write array to file
  //@@@ method here.  Options?

  //@@@ we ought to have some tidy-up
  //@@@ routine as we can't have a destructor

  public static function SetLoggingSwitch($OnOff){
    self::$loggingEnabledFlag=$OnOff;
  }

}// end of class Log
```

```
?>
```

The test program is almost identical. As we are running this without the $_SESSION system variable we remove `session_start();`

Discussion of second prototype

v2a has 54 lines compared to 94 in v2. The difference is more pronounced if only active code lines are counted. What's happened? Unnecessary functions have been culled and we've streamlined the 'are we currently logging' business.

- We've proved the concept of Class variables works. Instead of writing lines of text into an array in the globally accessible $_SESSION we are now working with a locally declared array specific to the Log class.
- There is no constructor or destructor to a class (there are for *objects* of course) so this needs some careful thought. With PHP we can initialise a class variable at 'compile' time so that makes life easer. We will still have problems if there are any open files and we don't find a way to automatically close them if the program halts under duress.
- We've been able to do away with the business of seeing if we are currently logging. We needed that originally to be sure of initialising the text array. Now, by the feature of having the class variable automatically initialised for us, we don't have to worry. (Check YCPL documentation on this subject - there are probably restrictions.)
- In fact we can probably do away with StartLogging() as well. It doesn't seem to do anything that SetLoggingSwitch() doesn't do.

Review

If you read the texts on program design you'll get a lot of stuff that tells you about formal specifications and formal documentation. In engineering terms these are the stages to get to blueprints. But if you're not manufacturing from scratch the prototype might contain most of the information itself.

So far (it will come) we haven't dealt with the associated documentation to enable use and maintenance.

The important part of this design is the way it is implemented not the underlying concepts. The production phase will be "I'll have one like this to production standard please". This won't always be the case - after all we're dealing with something quite simple here without complex interactions and responsibilities - but there is a danger of equating 'blueprint' with 'design'.

We have learnt a lot from our prototyping and now have a good foundation for production code.

- Hone techniques by exploring possibilities. The more alternatives you investigate the better you'll be able to judge what combination of options to use.
- Simplify. Don't be afraid to cut out features or leave them as to be investigated further.

Ready to produce final version?

Not quite. Those @@@s shouldn't be left until we're into the production phase.

- Do we need to do any tidy-up with the code as it stands? As it happens not in PHP but some other languages might. WCPGW? If we've used some memory and fail to release it when we've finished with it we could end up with a ***memory leak***. Each time we fail to release a buffer it might remain unavailable for other programs and could eventually mean there is not enough memory left for them to run.

  This is one of those fundamental programming rules: Whenever you claim a resource - whether its creating an object, opening a file, opening a communications channel, or reserving memory - always, *immediately* code the guaranteed release method.
- We had quite a few things to say about the hassles associated with files when looking at the first attempt. So far we've ducked the issue but we do need to bite this bullet if we're to have proper logging functionality.

By the way, we also had things to say about timestamps but these seem to have been quietly forgotten.

- Correct. How did you find that out? By looking at the design. You'd be surprised at how often design documentation is write-only. It might be a bit unfair here where the information is buried in a long tutorial, but the design is your reference document. (In a while we will revisit it anyway in the light of our prototype to see if there are any lessons we can incorporate or important observations that a future developer might find useful.)
- Timestamps didn't seem to need much exploration. There won't be much difficulty in converting the current time into text...
- ...Except there might be user-friendliness issues. A typical procedure would be to knock up a dummy and ask a sample of users what they prefer. That's as important a use of prototyping as technical proof of concepts.

Completing code exploration

`ProgBook/Ch14/v2b/ClassLog.php`

Add three Class Variables with suitable default initial values.

```
  private static $filePath = '';
  private static $fileName = 'log.txt';
  private static $timeStampStyle = -1;  // none
```

StartLog() stays unchanged. Perhaps we could add in an argument to set the file name and path all in one go? That looks like the quick-n-easy functionality we want for *tracing*, so let's leave it for now.

ClearLog() stays unchanged.

LogText() gets the addition of a timestamp. The way this optional feature will work is the user will specify one of three standard formats to use beforehand. (Date formatting is rather tricky, so this simplifies the operation for the user.)

```
  public static function LogText($Text){
  // main call to log text
    if(self::$loggingEnabledFlag){
```

```
      // possibly add a timestamp
      if(self::$timeStampStyle!=-1){
        $ts = self::TimeStampStr(self::$timeStampStyle);
      }else{
        $ts = '';
      }
      self::$loggingArray[]=$ts . ' ' . $Text;
    }
  }
```

GetLog() remains unchanged. In fact this is a mistake because the log might be in a file not in memory. We'll see how this error gets spotted later... ...or perhaps the program will be released without any further thought...

Timestamps

Add a new function that returns a string version of the current date/time in a standard format. We need this for two purposes: Adapting file names in the case of repetitive logs and prefixing logged lines. The curious combination of letters indicate the sort of element to use as components. For example `M` for short month name, `m` for two digit month number. We can make this private, that is not accessible from outside the class.

```
  private static function TimeStampStr($Mode=0){
  //  Return a timestamp in pretty string.
  //  Default format is H:i:s
  //  1 ... j M y H:i:s
  //  2 ... j M y H:i
  //  3 ... ymd  (for filename)
  //  4 ... His  (for filename)
    switch($Mode){
    case 1  : $f= 'j M y H:i:s'; break;
    case 2  : $f= 'j M y H:i'; break;
    case 3  : $f= 'ymd'; break;
    case 4  : $f= 'His'; break;
    default : $f= 'H:i:s'; break;
    }
    return date($f);  // formats according to $f
  }
```

Add a new function so that each day we can have a new log file named for us automatically. For convenience we give the user special substitution codes to use "!D" where the date is to go and "!T" where the time is to go. Of course these are optional. We don't want to give the user access so it's private.

> Filenames which are going to be catalogued need to be carefully formatted.
> • Leading zeros to keep numbers the same length otherwise `3.txt` will sort after `10.txt`.
> • Most significant part at the front for the same reason. eg yyyymmdd...

```
  private static function FileTimeSubstn($PlainFileName){
  // Substitute !D and !T with date/time in file name
  // WCPGW - null string
    $f = str_replace('!D',self::TimeStampStr(3),$PlainFileName);
    return str_replace('!T',self::TimeStampStr(4),$f);
  }
```

Finally in the timestamp department we allow the user to switch the style used for stamping each line.

```
  public static function SetTimeStampStyle($StyleNumber){
  // -1 for none  WCPGW?
    self::$timeStampStyle=$StyleNumber;
  }
```

Writing to file

The overall method is for the filename and path to be established ahead of calling a write method.

```
  public static function SetFileName($FileName){
    if($FileName!=''){
      // WCPGW with time substitutions?
      self::$fileName=self::FileTimeSubstn($FileName);
    }
  }

  public static function SetPath($Path){
    self::$pathName=$Path;
  }

  public static function WriteBuffer(){
  // user function (probably more variants to come)
  // Dumps whole buffer to file in one go
  // returns null string or error message
    return self::WriteToFile(FALSE,TRUE);
  }

  public static function GetFullFileName(){
  // Let user know where their file is
    return self::$filePath . self::$fileName;
  }

  private static function WriteToFile($Append=TRUE,$Clear=TRUE){
  // This is the private method that does the actual job of
  // writing text to a file.  Return an error message
  // or a null string.
    if(Count(self::$loggingArray)==0){
      return '';  // nothing to do WCPGR
    }else{
      if($Append){$mode='a';}else{$mode='w';}
      // errors opening file contained (@ stops reporting
      // $fhandle = false if a prob)
      // WCPGR other file errors?
      $fhandle = @fopen(self::GetFullFileName(),$mode);
      if($fhandle){
        foreach(self::$loggingArray as $t){
          fwrite($fhandle,$t."\n");  // WCPGW? line endings
        }
        fclose($fhandle);
        if($Clear==TRUE){
          self::$loggingArray=array();
        }
        return '';
      }else{
        return "Unable to open log file: " . self::GetFullFileName();
      }
```

```
    }
  }
```

The method we've used to deal with file-open errors is to pass a message back as a function return value so that the user can take whatever action they feel is appropriate in the circumstances. We trap any exception and suppress it[254] and instead replace it with a more benign version of 'it didn't work!' message.

WriteToFile() is private. This means we have control over when it will be used and what arguments are given to it. This should make it simpler to protect against improper use.

WriteBuffer() is user friendly *wrapper* for WriteToFile(). There will probably need to be more versions such as perhaps AppendBuffer() in the final code.

`ProgBook/Ch14/v2b/TestClassLog.php`
I expect you can guess what extra bits were added to the test program.

Review

That's quite a few extra lines of code with lots of cross-calling and tricky bits that need explaining. What a good thing we didn't try to do everything at once otherwise we'd have got lost or bored or confused.

Notice there were some WCPGW comments. When you think of a potential problem scribbling down WCPGW and moving on is a much better method than hoping you'll have the same thought when creating the production code.

Although we've got a few uncertainties about writing to file it's pretty much clear that we've got a workable system. Also we can easily see how to add features like 'count to ten lines then append to file'. HTMLlog and Trace derivatives don't look like they'll cause us any problems, the HTML aspects are all just 'add-on' and Trace is probably just a few functions to simplify quick, temporary logging.[255]

Prototyping principle

The object of prototyping is to see *How* to make something work. This applies as much to a database or a protocol or a data format or a web site design as it does to a conventional program. If you need to confirm or clarify something then it's worth prototyping.

Building a prototype is stage one. Stage two is reviewing it. Don't forget stage two!

Question :
Hey but why build a prototype when I can just write the whole thing straight off?
Answer :

---

254 The @ in PHP absorbs exceptions. fopen returns false if an error so we still know about it.

255 At this point we could review our design in order to confirm these suspicions.

- What have you lost? Nothing. Your production code will be a spruced-up prototype.
- You're not committed to your first thoughts. Conclusions of reviews are easier to implement at early stages.
- You have a boundary between the essentially creative phase and the essentially mechanical phase.

Shall we code?

Let's review the parts of the complete project.

- Code itself
- User documentation
- Benign exerciser program (Test and example)
- Crash testing program
- Development notes and configuration

Where should we start?

We will be working on all of them in parallel to some extent so the first thing might be to create the filing system and shell documents. It is a lot easier to add to an existing document framework than stop, think about the format, think about the structure, then add a bit. By the time you've finished you've lost your original train of thought.

Document formats

If you're keen on blueprints such as *UML* then you'll need some editing tools, but at some stage you might want to make sure that the documents themselves are viewable without the tools.

Likewise user documentation needs to be accessible. This probably means HTML and PDF although plain old text is often quite sufficient. Unless you're developing a huge application you shouldn't need fancy navigation - just provide a good index and table of contents.[256]

Unless you have a really good and fast diagramming tool, try to leave pictures out or keep them on backs of envelopes. You can spend ages fiddling with graphics. On the other hand, simple graphics that give instant overviews can be good for quickly orientating readers.

A lot of documentation will be in the source code in one way or another. Some of this might get extracted automatically to generate an API. Check out the tools to do this and the variations on comments they use to achieve this. Typically the results are in HTML on a class by class or file by file basis so you may need a backbone to tie them together.

Document structures

There is no 'best' structure, the important things to remember are

- Ease of creation and maintenance

---

256 You'll need to write clearly, check spelling and details. Yor code mite be brill cept if you express it can't then everybody won't be crediting you!!!

- Utility
- Simplicity of access. That is a good filing system.

Plan of work

We're already half way through the plan of work.

1 Appreciate the task. Get a gut-feel for (a) What (b) How (c) Amount of work
2 Top-down design
3 Bottom up design
4 Proof of concept and prototyping
- Plan production *(Current task)*
- Finalise design and review estimates of amount of work
6 Finalise development environment
- Write user documentation
- Write code. NB Possibly in stages.
- Get code to work (Exercise)
- Check code works (Crash test)
8 Build finished package of deliverables
9 Consolidate development documentation and review

5 and 7 can get mixed up and there might be some grey areas.

> This process will be slightly different for each project. Small stand-alone programs will be done 'on the back of an envelope'; large applications by many team players supposedly coordinating efforts through a shared work environment. It can be applied in rough detail to a large project on the back of an envelope or with great precision (and expense) to a single critical component.
>
> You might be employed to start at 7 but you still need to be au-fait with the background and within your own little sphere do the whole 1-to-9.
>
> If you're ever parachuted into a project use this list as an agenda to find out where things have got to and which bits have an unpleasant smell about them.

User documentation - Now!

You don't have to write the user documentation before getting stuck into coding. In fact there might be hardly any user documentation but lots of technical reports on the results of prototyping, algorithms used and so on (*How*) - In which case make a start on that instead.

There is a reason for writing the documentation before the code if there is a good match between the final specification and the delivered documentation. Basically you can cheat by using the user documentation as your specification. You'll want to add side-notes of your own, for example exception conditions that you'll be wanting to check with testing.

There is another good reason for writing the user documentation at this stage. If you've been working hard on code while tinkering with the prototype then this will

- make a nice break from code-wrangling

- be a change of view from the *How* to the *What* which might uncover odd issues.

User documentation - components

For our project we can identify three components:

1 User guide - Overview, origins, purpose and scope.
2 API - To be mechanically extracted from source
3 Exercise (test) program

Item 1 is probably best written with a handy word processor then converted to PDF.
Item 2 will probably be produced automatically in HTML
Item 3 will be source code

Writing the user guide

Read some to find out what structure and style to use. Perhaps the most important things to remember are:

- Few people can be bothered with words these days. It doesn't matter how many times you shout at them to RTFM, they won't....
- ...So try to provide some instant gratification or make them jump through stages. Compare:

  (page 1) "*Before you start ...* " *(page 3)* "*Installing*" *(page 6)* "*Start the program*"

  With:

  "*There are only three steps: (1) Checklist (2) Install (3) Run*"

  And:

  "*Welcome to Super-Foo. You're only 3 minutes from seeing your first bar*
  - *Is my system ready? – See page 3*
  - *How do I install? - See page 6*"
- Why do you think books like 'Card-walloping for idiots' are full of pictures and big writing? That's the level of many users...
- ...But as technical documentation for technical people 'Learn Loxodrome in 12 minutes' books are pretty hopeless.[257] Instead they want rapid access to specific data in a concise format.

  - Crisp
  - Structured
  - Tabular
  - Indexed
- The general explanations and getting started type of approach may not be suited to the technical approach in which case you'll want to physically split 'User guide' from 'Reference'. It's probably a good idea to logically split these two anyway.
- Even if your program is just a module to be used in a bigger program there still needs to be some overview information (as well as the technical) in order that other programmers, (they're the users if you think about it), get hold of the right end of the stick, know what stage of development it is and where to take care.

Review

It's quite important to pause before launching on the production code. You'll have no bother restarting as your explorations have shown you clearly what's to be done, the difficult bits have been cracked and now you can concentrate on a combination of loose ends and quality of finish.

---

257 What does that tell you about that shelf full of 'You too can manage a mega corporation over lunch' books... ...and the 'managers' who buy them?

Your documentary tools are all set up for instant access so you can switch between them as you go.[258]

One things you might want to put in hand before embarking on production code is acquiring sample test data. This often takes time to arrange.

PRODUCTION_PREPARATION_EXERCISE

0. With the logging exercise as described above
1. Set up filing system
1.1 Create skeleton documents
1.2 Organise the necessary editors ready for efficient use
2. Write a things to do list
3. Write the introduction to the User Guide
4. Check you know how to use automated documentation tools

On the production line

Here's a flavour of production coding. There will be quite a bit of skipping about and loose ends to come back to later.

Plan

- After the code, the API will be the main thing. We can generate this with a documentation tool using in-code comments.[259]
- We'll concentrate on Log with HTMLLog and Trace tagging along. (We need to make sure Log is capable of supporting the inherited classes, but also we need to be confident it works properly before using it.)
- We can use an exercise program as a sample to be packaged with the code. Call this TestLog.php
- We need a way to stress test Log. This may need multiple programs and peculiar configurations. Perhaps we'd better start with a file called Testing.txt[260] for documenting this aspect.

Start production

- Create production directory
- Copy prototype code to production directory
- Create a file for our own notes

ch14\prod\DvntDoc.txt

```
Chapter 14 development documentation
------------------------------------
TODO
* ClassLog
```

---

258 I can heartily recommend a two screen system so you can flick between documents without covering up too many.

259 I wrote my own for PHP so my in-code comments that will be turned into a user-friendly API reference will be non-standard.

260 A word processed file might be as easy but it would need quick switching of editors.

```
      Smarten up in-code docn/comments
      Flag TODOs
    * Create ClassHTMLLog and ClassTrace and etc
```

As a real programmer you might not have bothered with details of the TODO unless it was approaching time to finish for the day. The important thing is that this file is instantly ready for immediate jottings as we think of them. As time goes by you can add STATUS and PROBLEM or any other headings. In theory, one day all that will be left is STATUS.

- Tidy up the header comments of ClassLog.

ch14\prod\ClassLog.php

```
<?php
/*
  LOG  (class)
  ===
  General purpose logging to memory buffer and file

  Peter Fox
  See Ch14/prod/DvntDoc.txt for status etc.
*/
```

  - /*...*/ is a block comment in PHP. Most languages have something similar.
  - My personal documentation standard here is name on line 3 and short description on line 5. Your documentation standards will be different.
  - The last line is a bit of a cheat. At the moment it's the best we can do otherwise we'll be spending all our time marking changes here when it's a very fluid 'work-in-progress' situation.

- Tidy up the class description comments

```
class Log{
#---------------------------------------
#  Class methods for logging text to memory
#  array with optional dump to file
#  (This class cannot be instantiated)
#  Usage is fully illustrated in TestClassLog.php
#---------------------------------------
```

  - Jot a note to copy the prototype TestClassLog and scheme some suitable presentation.

- Add documentation to fields

```
  private static $LoggingEnabledFlag = FALSE;
  # Master switch that disables logging when false (default)
  private static $LoggingArray = array();
  # Buffer
  private static $filePath = '';
  # If this is a null string we'll write the log
  # to the current directory ([T] may be execute only!)
  private static $fileName = 'log.txt';
  # This is the *actual* file name being used for logging
  # May be different to name of file *as specified by user*.
  private static $timeStampStyle = -1;  // none
  # Integer corresponding to TimeStampStr() argument used
  # to prefix text lines.  -1, default, is none
```

  - We have to consider who might be coming along later in a maintenance role and needs to know exactly what's what.

- [T] is a TODO tag alert. This will prompt us to crash test[261] the issue sometime. (There's no need to write it in the development notes as well - one knot in a handkerchief is enough.)[262]

Review

So far we've followed the conventional pattern of:

1 Identifying the file and its contents in a file comment
2 Describing a class with a class comment
3 Documenting fields

before getting onto the methods. We haven't changed any code, simply bringing the prototype up to production standard by means of decent comments.

Methods

The next stage is to look at the methods. Now we've had a little interlude we can do this at a number of ways:

- Have we got the right mix of methods for live use?
- Any missing? Any surplus?
- Is there any *How* that doesn't feel right?
- Which bits of the code are the core of the action.

Then on a lower level

- Are the methods arranged in the code with some logic
- WCPGW with method arguments
- WCPGW with logic
- Where should user documentation be clear? (*What*)
- Where should developer's documentation be clear? (*How*)

That's a lot of things to do all at the same time. I suggest two passes: Reflect on the first four items making notes as you go then get stuck into nitty-gritty coding one method at a time.[263]

Reviewing methods

- Mix? Difficult to tell. We've 'sort-of' covered all general logging bases. There's that matter of append or overwrite buffer that might be better cleared up. Perhaps we should give the user a one-shot method to write to a named file.
- Missing?
  - Let's add a 'WriteFile(filename,overwriteflag,clearflag)' method for general one-step use.
  - At some stage we're going to need to know the number of lines in the buffer in order to add a function to automatically force append to file after so many

---

261 I'm using the term 'crash test' for 'trying to break it' testing. As opposed to exercising which is a benign form of testing. The distinction is important. In this example we'll be handing over the exerciser to the user and keeping the crash tester for ourselves.

262 This is quite an important matter of self discipline and efficiency. There is no reason to be repetitious unless you need information collated in different ways.

263 If you're keen you could copy the first four items into a checklist format in your development notes. eg

Q : Any missing?
A : [ ]

Then fill in the answer boxes as you go. This is the sort of thing you might do as part of a formal quality system.

lines. We could count these in LogText() to trigger the necessary flush.... ...As it seems the work of only a few lines of code we might as well add it now.
    - There are some @@@s hanging around still telling us to look at a tidy-up routine. OK - Bite the bullet - What do we need to tidy up? We don't ever have any files left open so our code is well behaved from that point of view. We could clear the buffer in case it contained a massive amount of text, but we've already got a method for doing that. PHP will garbage collect¤ the buffer automatically, so it looks like the most we need to do is warn users of resource implications - but then our users will be developers and should know that for themselves. Conclusion: After consideration it's a non-issue.
- Surplus? StartLog() doesn't do anything that SetLoggingSwitch() doesn't do. Either it's surplus or doing the wrong thing. The only other thing we could use it for is to set up a file name/path, but since we've got perfectly good methods for that let's excise it. [264]
- Anything still not feel right?
    - SetPath() takes any argument at face value. Ummm.
    - Could do better with the arguments for SetTimeStampStyle(). This is a user function where they're expected to put in a magic number. Perhaps we should give them some help to save them having to grovel through documentation to find the numbers. Two ways spring to mind : (a) No-argument functions (b) Defined constants. (For pedagogical reasons we'll use the latter.)
    - ClearLog() and GetLog() are not really properly named as they work on the buffer array not the log file. We need to get a grip of terminology. Perhaps we should define 'Buffer' and 'File' and ban the use of 'Log' when we mean one or other of these. In which case these become ClearBuffer() and GetBuffer()... ...Which suggests we need ClearFile() and GetFile(). (This is almost prototype territory. Don't be afraid to revisit the prototype if things aren't right. It is far better to deal with lash-ups now rather than later.)

> It's quite normal to have to weed out and clarify terms that have developed during the earlier stages. You might want to add it to your checklist.

- Core code?
    - LogText() is where important action happens and is likely to be more involved with descendent classes.
    - TimeStampStr() contains non-obvious technical matters
    - FileTimeSubstn() happens at a certain time to fix !D and !T. Are we calling it at the right time?
    - WriteToFile() has a fair bit of logic in it and may have to handle exceptions and generate error messages.

---

[264] Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove. - Antoine de Saint-Exupery

You'd probably be making cryptic notes in the code with @@@'s[265] or writing stubs (methods that don't yet do anything) rather than typing everything out like I've done above.

> The more you can do in your head without 'dropping stitches' the better. That's why when you're working on developing production code you don't want any interruptions or distractions. If you share an office then make it clear you're not to be disturbed, divert the phone, and switch off intrusive 'new email!' notification. Much better, work alone. NB When you surface at the end of a deep code session it doesn't mean the others have as well!
>
> You should easily be able to double your productivity by decent isolation.

Review (Methods review)

This pass should have clarified what code and documentation to work on.

Now you're free to focus tightly on small technical areas at a time. This reduces scope is quite important, as for every method there is probably something to exercise it and possibly some testing of WCPGW to set up.

Computer programming at last!

Very many programmers think the next phase is the start and end of computer programming. (Lots of them leave out the exercising, testing and documentation parts as well.)

You've done all the hard creative, fluffy bits. Now you're putting everything in exactly the right position and checking the bits fit together. Your current task turns out to be lots of little jobs where the challenge is in the details not the underlying method. Until now we've skipped a lot of minor issues, but now they're identified we can (a) code (b) exercise (c) test them.

```
#-------------------------------------------------------------------
public static function LogText($Text){
#  Add Text as a line to the log buffer
#  This method is en/disabled with SetLoggingSwitch()...
#  NB ... Default is OFF
#  See also SetTimeStampStyle() and WriteBuffer()
#  If ForceAppendToFile() qv is used then this might
#  return an error message to complain about bad file name
#  or permissions etc.
#-------------------------------------------------------------------
  if(self::$loggingEnabledFlag){   // master switch [E][T]
    if(self::$timeStampStyle!=-1){ // any timestamp? [E][T]
      $ts = self::TimeStampStr(self::$timeStampStyle);
    }else{
      $ts = '';
    }
    self::$loggingArray[]=$ts . ' ' . $Text;
```

---

265 @@@ is the tag I use for 'Loose end'. Anything similar that is easy to search for would work.

```
      // Might be more lines in buffer than we want  [D][E][T]
      // So force appending lines to file
      // @@@ $arrayLinesLimit and ForceAppendToFile()
      if(count(self::$loggingArray)>self::$arrayLinesLimit){
        return self::WriteToFile(TRUE,TRUE);
      }
    }
  }
```

- We identified this as an important method so we've made a good effort with the method description.
- Adding that flushing logic on the bottom was only a few lines... ...but it's introduced a new Class Variable and function to set it which we need to follow up. Also, as we've documented the flush might produce an error message result.
- [D], [E] and [T] are flags to remind us we need to Document, Exercise and Test these items and logic.

ClearLog() and GetLog() renamed to ClearBuffer() and GetBuffer().

- Don't forget to change any other references.
- Add method documentation. Not a particularly important method from the user's point of view so we don't need much.

- Add [T] for GetBuffer() - Does an empty buffer cause any problems
- Do we really need to [T] for the $loggingArrayFlag test? Probably not as we've [T]'d it already in LogText() and we know that if we'd mis-spelled it the PHP compiler would complain.
- ClearFile() and GetFile() are stubs or ToDos in the file handling section eg
  `// @@@ public static function ClearFile() NB loggingEnabledFlag!`

```
  #-----------------------------------------------------------------
  public static function ClearBuffer(){
  #  Delete any lines of text in the buffer   [E]
  #-----------------------------------------------------------------
    if(self::$loggingEnabledFlag){
      self::$loggingArray=array();
    }
  }

  #-----------------------------------------------------------------
  public static function GetBuffer($Flush=FALSE){
  #  Returns an array of items in the buffer [E]
  #-----------------------------------------------------------------
  // return array of text
    if(self::$loggingEnabledFlag){
      $retVal = self::$loggingArray;  // [T]
what if array is empty?
      if($Flush){self::ClearBuffer();}
    }
  }
```

- It seemed so easy to offer the user the choice of flushing the array after getting the contents that it got added in just a few seconds...
- ...But it's introduced a basic programming error. Because in PHP `return` causes an immediate end of processing the current routine we can't

This is a classic example of how a 'trivial' change can clobber code that has worked perfectly.

Afterthoughts are good before testing but slippery afterwards.

just add another line after the existing return that reads the array. ie

```
return self::$loggingArray;
if($Flush){self::ClearBuffer();}
```

would never get to execute the flush. Obviously we can't clear the buffer before reporting it! So we've got to make a temporary copy, then flush, *then return $retVal.* But that last tiny step is missing![266]

The 'master switch' needs explaining and how many times will people try to log but be disappointed when nothing happens?

Perhaps we should set the default to ON. But then users would happily code without the need to think about where to put a master switch... ...until they needed it and might end up only partially disabling logging. It's a debatable point.

```
#-------------------------------------------------------------
public static function SetLoggingSwitch($OnOff){
# This method is a 'master switch' that enables or disables [E][T]
# all logging.  IMPORTANT : The default is OFF([D] FAQ?)
# $OnOff is a boolean
#-------------------------------------------------------------
  self::$loggingEnabledFlag=$OnOff;
}
```

Review

We've been caught out a couple of times by making quick changes. The better you get the less risky off-the-cuff alterations are... ...but if you're not 100% immersed in the code you are likely to slip up. So that's an automatic prompt for a '[T]'.

So far we've done one of the three segments of code. ('Basic logging to buffer' out of 'Timestamps' and 'filing'). This is a good place to stop to at least exercise the code we've just been messing about with. Even if we can't test yet we should be making notes or elements of testing ready for later.

> Logical segmentation of code is a good idea if you can arrange it. The boundaries between segments will often be grey but that doesn't matter, the main thing is that you can focus tightly on a small amount of code.

Exercising

The exercising program will be a cousin of the TestClassLog program used before. Remember that we've decided to kill two birds with one stone by making the program available to the users as an example[267] as well as a 'can we get it to work' tester.

- Before we start, we've got to identify the code, what it does and what it applies to.
- We also need an environment in which to run the code. In this case it amounts to a very simple print out. As PHP's built-in list an array function isn't very pretty lets

---

266 Keep reading to see if anyone will ever find out before the code is released!

267 Many people find that working from examples is a lot easier than tutorials or references.

write our own.

A simple method is

- list the methods we've just worked on in the code
- work them into a story...
- ...with ToDos for options

ch14\prod\TestClassLog.php

```
<?php
/*
  Test ClassLog
  =============
  This program illustrates the features of ClassLog
  and shows how they might be used.

  Peter Fox May 2006
  See ClassLog.htm for complete documentation
  See Ch14/prod/DvntDoc.txt for status etc.
*/

//  Always required
require_once('ClassLog.php');

function ListArray($AnArray){
# Convenience function to list lines from an array
# \n character becomes a <br> for screen display purposes
  if(count($AnArray)==0){
    print("<i>No items in array</i><br>");
  }else{
    foreach($AnArray as $t){
      print(nl2br($t) . "<br>");
    }
  }
}

// --------------------------------
// Basic logging to a memory buffer
// --------------------------------

// Do the work ...
Log::SetLoggingSwitch(TRUE);    // Switch logging on
Log::LogText('to be lost');
Log::ClearBuffer();             // erase all so far
Log::SetLoggingSwitch(FALSE);   // switch logging off
Log::LogText('to be ignored');
Log::SetLoggingSwitch(TRUE);    // switch back on
Log::LogText('Hello');
Log::LogText('World');
$hw1 = Log::GetBuffer(FALSE);   // get log *don't clear*
Log::LogText('Again');          // append more
$hw2 = Log::GetBuffer(TRUE);    // get log *clear buffer*
$mt  = Log::GetBuffer(TRUE);    // should be nothing

// ...report the results
print("<u>Hello World</u><br>");
ListArray($hw1);
print("<u>Hello World Again</u><br>");
ListArray($hw2);
```

```
print("<u>Empty</u><br>");
ListArray($mt);

?>
```

- That's nice and neat. Notice how we've been economical and logical.
- The presentation of the results should be self explanatory.[268]
- One objective of example code is for it to be read and understood from the page without having to be run.

When we try to run this program the first time there will probably be bugs and blanks. Because of the lack of trace features we might add *temporary* instrumentation to the code *for our own purposes*.

Running the exerciser

After weeding the inevitable typing errors the screen reports a distressingly uncompromising failure:

```
Hello World
No items in array
Hello World Again
No items in array
Empty
No items in array
```

Hooray! OK so it wasn't difficult to spot the 'nothing returned' bug in GetBuffer() but nevertheless it's one less bug for users to find. So insert

```
return $rv;  // Don't forget this next time!
```
[269]

- I had to do some trivial tidying of output formatting before being satisfied with the layout. That's quite normal.
- If we were writing a test program just for ourselves we might not wait until the end of all processing before displaying the results.
- If we're happy that we've exercised the routines we can clear the *appropriate* [E]s from the code and move on to crash testing.

Proper testing

I've been referring to the two sorts of testing as 'Exercising' or 'Checking' and 'Crash Testing' or just 'Testing'. The difference is that the Checking shows the code can work while Testing proves it always works.[270] (The exerciser program is called "Test..." because that's what most users will consider most appropriate.)

> Terminology confusion alert: Whilst we, in our development environment, know testing to be 'a good thrashing (that might break it)' users think of testing as 'a good example that will always work'. Hence the file naming conventions I've adopted.

268 PHP pretends to be a web page; hence the HTML. <u>=underline
=New line <i>=Italics

269 "Honestly! You computer programmers are always messing with silly remarks instead of getting on with the job." Actually we want to flag mistakes to help avoiding them again. Since in most cases formal documentation would be a burden and pretty useless (being divorced from the code) a semi-humourous quip is the next best thing - because it breaks up the pure logic and sticks in the mind.

270 For values of 'always'.

We have just seen that checking is relatively straightforward. If something gets left out it doesn't really matter. Testing is different!

- It's bitty
- It may need extreme circumstances that are difficult to reproduce and/or very unlikely.
- Some risks are very well hidden

For an example lets look at calling LogText(). What-if: (Comments in italics)

- There is no argument supplied? *Possibly picked up by compiler (run-time)*
- Two arguments are supplied? *Probably picked up by compiler (run-time)*
- The argument is a null string? *Should be OK*
- The argument is a simple type such as a floating point number?
- The argument is an object?
- The argument is an array?
- The argument is a function that returns a string?
- The arguments points to a null reference?
- The argument contains non-ASCII characters? *Which byte codes might be dangerous?*
- The argument is Unicode text?
- The argument is 10,000 characters long? *10,001,10,002 ... etc!*

Arghhhh! Definitely time to run away and hide. Clearly we can't actually test for every single possibility. So what's the answer?

1 Reduce the number of possibilities with checking code in the routine
2 Convince ourselves that certain issues are not a problem
    - By detailed knowledge and experience
    - By logical deduction
    - By experiment
3 Shrug off 'the sky falls in' and 'what do you expect if you do that' cases.

So far all we've done is listed some stimuli. What about the consequences, or more importantly how our code deals with the consequences. This is the core of the matter: *Does the code deal with difficult situations?*[271]

Back to the drawing board

If all we're prepared to accept as an argument for LogText() is a string then we can test for that first-thing.

```
if not-a-string(arg) { substitute "not a string" for arg }
```

That's knocked out a lot of possibilities working their way into the delicate parts of our code but we still need to be confident that this test logic actually works so we still need to bowl some bouncers at the routine.

It is quite common to develop code without worrying too much about difficult situations. Often you'll do argument checking as a matter of course but that may not include all possibilities and rogue conditions. If you can be tight and confident with your checking from the beginning that's a great help, but there may still be issues that

---

271 We're not interested in impossible situations but we may be interested in maliciously contrived ones.

need somebody with malicious intent to deal with.[272]

ch14\prod\CrashTestClassLog.php

```
<?php
/*
  Crash test ClassLog
  ===================

  1 LogText
  1.1  Arguments

*/

//  Always required
require_once('ClassLog.php');

//  Maximum error reporting sensitivity
error_reporting(E_ALL);

$RED="<font color=red>";

function ListArray($AnArray){
# Convenience function to list lines from an array
# \n character becomes a <br> for screen display purposes
global $RED;    // (PHP : Access to variable outside function)
  if(count($AnArray)==0){
    print("<i>No items in array</i><br>");
  }else{
    $lineNo = 0;
    foreach($AnArray as $t){
      $lineNo++;
      print("$RED <small>$lineNo</small>[</font>$t$RED]</font><br>");
    }
  }
}

class DummyClass{
  function HW(){return "Hello World (Dummy)";}
}

// report heading
$d = date('D j M Y   H:i:s');
print("<h1>Testing ClassLog.php</h1>$d<p>");

Log::SetLoggingSwitch(TRUE);    // Switch logging on

// --------------------------------
// 1 Logtext()
// 1.1 Test abnormal arguments
// --------------------------------
// 1.1.1 Build array of test items
print("<h2>1.1.1</h2>");
$args = array();
```

---

[272] It's quite natural to be thinking 'how can I get this code to work sensibly' at coding stage which is a different mind-set from the 'what else might break it' tester's motto.

```
$args['text']='Normal text';
$args['int']=567;
$args['float']=567.89;
$args['nul string']='';
$args['null']=null;
$args['boolean F']=FALSE;
$args['boolean T']=TRUE;
$args['object']=new DummyClass;
//$args['string function']=  Too complicated @@@
$args['array']=array(1,2,3);
// nasty binary characters probably in the low values
// so do high ones first.  Wrap with square brackets
// as delimiting indicators.
for($i=15;$i>=0;$i--){
  $binstring = '[';
  for($j=15;$j>=0;$j--){$binstring .= chr(($i*16)+$j);}
  $args["Binary string ($i)"]=$binstring . ']';
}
$longstring = "This will be a long string. ";
for($i=1;$i<=5;$i++){$longstring = $longstring . $longstring;}
$args['900 chars']="START".$longstring."END";
$args['text at end']='Normal text';

// 1.1.2 Run through array of possibilities
print("<h2>1.1.2</h2>");
foreach($args as $testName=>$testArg){
  print("<br>Test:$testName ");
  Log::LogText($testArg);
}
// 1.1.3 Report what's in the buffer
print("<h2>1.1.3</h2>");
ListArray(Log::GetBuffer(TRUE));
print("<p>$RED End of 1.1 $RED");


print("<p>$RED End of test $RED");

?>
```

You can see that this code looks quite different from the exercise test.

- More formal with a numbering system for reference
- ListArray() serves the same purpose but has been beefed-up.
- Additional bits and pieces we might need for testing and reporting.
- Proper report heading including timestamp
- Error reporting set to maximum sensitivity.

Setup - Do it - report pattern

It's quite handy, if possible, to split the setting-up from the execution. Remember that the execution may well result in an awful crash... ...but the setting up of extreme situations may be equally risky. We may be inserting temporary debugging code which might upset a report[273]. Hence this three-way-split.

---

273 If you can direct real output to one place and reporting output somewhere else that's quite handy also. All you need is a good trace utility...

- PHP gives us ready to use type-indifferent associative arrays which makes building an array of oddities and nasties quite easy.
- It's quite a good idea to put the more unsavoury items towards the end of the test. This gives you the confidence of reacting reasonably when normally stretched before a moment-of-truth.
- Of course our test data is only a sample.
- In this case the tests are pretty much self-documenting... ...but see below.
- It is standard test procedure to re-run a 'good case' at the end which can be compared with it when it was run at the beginning to confirm there hasn't been any internal upset.
- 3 is a generally accepted starting place for the number of items in an array or times to loop. If it works with three ('one at each end and one in the middle') then you've probably dealt with all possibilities.[274]

Stop and ...? (1)

It happened that none of the tests executed in 1.1.2 caused a crash or warning. What would we do if it did? There are four things we need to do:

1 Trap the exception
2 Report it
3 Flag was that good or bad, correct response or unexpected.
4 Continue after tidy-up if appropriate

To do this we need

1 Some of try...except code

```
Set was-exception-raised to false
try{ do some risky test }
except[275] {
  Set was-exception-raised to true
  capture result and any details
  manage consequences
}
```

2 A way to tell if what's happened was what we hoped for

Some languages allow you to set up specialised error handler routines. That's getting a bit deep for us here, but worth investigating to see where YCPL might be cleverly adaptable. The advantage of writing your own is that you can switch it in just for your development and at other times use the default or a user-friendly version.

Stop and ...? (2)

What would we *like* to do?

a Stop at error, report error, tidy up, shut down, or
b Log error, tidy up, continue, full report at end, or
c Some combination of a and b.

It's probably nice to have a test program that can be left to run then finally reports PASS,PASS,FAIL,PASS etc with some pretty traffic lights but is it practical? Quite

---

274 It can sometimes be surprisingly difficult to make up three test records in a database off the top of your head. Much easier to use real data... ...except using real data for experiments and then possibly training materials and so on is not such a good idea either.

275 This is a case where you'd want to be quite specific with certain exception types.

probably for 90% of the tests, with others needing configuration alterations that are too involved to trust to an automated system or are 'dead-ends' or too bizarre to bother with as a standard test.[276]

Review of test program

In view of the Stop and ... sections above how does CrashTestClassLog.php look?

- So far we've got away without try...except but it probably won't last
- We have reported *something* which we can inspect...
- ... but we haven't judged the results except on the grounds of 'no crashing'.

Are we going to have to repeat the argument thrashing we've given LogText() with all the other routines that need a string? Luckily the answer is no. What we're looking at here most of the time is 'standard behaviour'. How long a string can be, what are acceptable characters, what happens if we give a number etc. are characteristics of the language.[277] Out fluency with the language allows us to manage any limitations and quirks as a matter of course. It's still a good idea to keep up the pressure with Gotchas - you learn about these mostly by experience - embarrassing experience.

When you hear the phrase 'matter of course' that should make you stop and take note. What the person who says it often means is 'Who cares? Life is short. Let's not bother with boring details'. As a programmer you think: 'Looks like a case for a standard checking procedure here'.[278]

As far as it goes we've got a neat little program that isn't too difficult understand and won't be difficult to expand but is still at the 'poke-n-hope' stage. We are going to need to either document what should happen "You should see *this*" might work fine as a 'go-no go' test, but it still needs some documentation. The question is where should the documentation go: (a)code itself (b)test code (c)test data? Probably a combination of (b) and (c).

Improving the test program

Firstly it should be pointed out that there is always some danger of spending far more time on a test program than the results justify. The ideal person to do testing is a meticulous investigator and diligent documenter but there has to be a limit.

Are 1.1 results correct shortcut

HTML note: Here is the modified 1.1.2 code that adds little red sequence numbers. If you're serious about reporting in HTML then you'll want to be investing in styles and wrapping types of output in convenience formatting functions rather than the sort of complication shown here.

```
// 1.1.2 Run through array of possibilities
print("<h2>1.1.2</h2>");
$testNo = 0;
foreach($args as $testName=>$testArg){
  $testNo++;
  print("<b><small>$RED$testNo</font></small> Test:$testName ");
  Log::LogText($testArg);
}
```

276 Test 6.6.6: Try deleting all files with root permission. Result - Should be denied!

277 Most languages are more strongly typed where these issues are dealt with long before the code is allowed to run so PHP is not typical in this respect.

278 And then: "I wonder if somebody has been this way and already done the job?"

We've numbered the listing in 1.1.3 if we add matching numbers to 1.1.2 (see how handy those reference numbers are) we can match up test description tag with results. *In this particular case*, because these are not the most vital tests, we might be allowed to get away with this - clearly it isn't an ideal solution involving as it does cross referencing the 1.1.3 list, the 1.1.2 list and the source of CrashTestClassLog. This still doesn't tell us if what we're seeing in the results is acceptable.

Check for [T]s

So far our test has picked on one aspect of one routine. That's hardly comprehensive! Luckily we've been making notes (in the code in this case) where there are items we might want to test and marking them with easy to find [T]s. We'll ignore the file and timestamp ones at the moment.

- `if(self::$loggingEnabledFlag){ // master switch [T]`
  Does master switch (`$loggingEnabledFlag`) work in LogText()...
  ... and if it does, is testing here good enough to apply to similar cases?

  We have exercised this in the check/exerciser(TestClassLog). Do we need to repeat it here? Probably not, provided we make a note about what the exerciser should do.

  If this conditional test works here it should always work elsewhere. *This is a big statement to make. Sloppy, quick-n-dirty programming invalidates it. That's why most of this book is about good programming practice, and developing style and technique that makes such statements possible.*

- `$rv = self::$loggingArray; // [T] what if array is empty?`
  This is in GetBuffer(). The exerciser has a case of this where nothing went wrong. Job done! Err... almost.

  The exerciser calls GetBuffer() at the end of it's run. Everything worked as expected. However the array is a data structure and as a general rule *data structures need initialisation*. The question arises "how do we know the data structure has been initialised". Often programmers defer initialising data structures until they are needed. We could have written LogText() like:

  ```
  if(not buffer-is-initialised){initialise buffer}
  rest of LogText code
  ```

  (As it is there's a static initialisation done for us but that's a bit of a fluke.) If initialisation relied on making at least one call to LogText() and we never made that call then we're in trouble.

  So the `[T] - What if empty` has been dealt with by the exerciser but `[T] - Sure this is initialised` needs either crash testing or *logical proof*. In this case we have the logical proof because

  ```
  private static $loggingArray = array();
  ```

  automatically does the initialisation for us. The proof comes from a combination of

code inspection with how the language works.[279]

If we were feeling verbose, instead of just removing the [T] comment we'd replace it with `// (Statically initialised)`.

- We've duplicated the "does master switch work" [T] in SetLoggingSwitch(). It's difficult to think of anything more to be done.

Method arguments

We gave LogText() a good thrashing. What about the other methods? It so happens that although PHP, unlike many other languages, accept anything as arguments it is good at not getting hung-up about silly type conversions. So are we off the hook with GetBuffer() - yes ... ... SetLoggingSwitch() - no!

Here's the current code for SetLoggingSwitch():

```
#-------------------------------------------------------------
public static function SetLoggingSwitch($OnOff){
# This method is a 'master switch' that enables or disables [E][T]
# all logging.  IMPORTANT : The default is OFF([D] FAQ?)
# $OnOff is a boolean
#-------------------------------------------------------------
  self::$loggingEnabledFlag=$OnOff;
}
```

In GetBuffer() we only use the $Flush argument for a test to see if it 'is true'. But in SetLoggingSwitch() we *store* the $OnOff argument. Later on it will still be used for an 'is true' test but now we're going a bit too far. Suppose the $OnOff argument was something large or stupid. In the first case we're lumbered with carrying somebody else's rubbish and in the second, if there is a problem with evaluating 'is true' we won't find out about it until we come to use it by which time we'll have a job on our hands finding out what's gone pear shaped.[280] So the good practice rule is *never store unvalidated arguments in fields (even if there's no risk to the program logic.)*

So let's rewrite the SetLoggingSwitch() method:

```
public static function SetLoggingSwitch($OnOff){
  self::$loggingEnabledFlag = ($OnOff==TRUE);
}
```

Now we are guaranteed a boolean value in $loggingEnabledFlag.

Scripts

This is getting a bit beyond the scope of this book, but I'm sure you can see that with our test program we're only an ace away from the stage where we could read in a file with test commands, data and correct results which would then be executed and the actual results compared with those in the file to look for discrepancies. Bearing in mind that all we're doing with our code is checking some logic and that many programs have to work in real time, under stress, with simulated interactions, with thousands of

---

279 Who knows, in the next version of YCPL, or a variant of YCPL, certain features that you relied on are changed. - Ouch! That's why most programmers explicitly initialise everything.

280 The same applies to database data: Garbage in today...Garbage out next week.

possible variations and combinations of user input this degree of automation is very appealing.

Specification

I've mentioned before that specifications should be taken as guides. The blueprint says "To be painted" and it's up to you as completion draws near to find out what colour the client wants.[281] When you come to crash testing you will want to explore the *implications* of the specification. A typical case is where an "integer" is called for. Does that include negative numbers, zero or 500 million? Most of the time you'll make a sensible decision in the early days of coding but what about testing - when you've got to slightly rewrite the spec? There are two more big holes in specifications:

- Dates - Not only different layouts but possibly 'June', not June in any year or any particular day in June.
- "...takes one integer argument...." might mean "most of the time use the integer provided but there may be occasions when data isn't available - please work some sort of miracle"

There are whoppers in specifications also. Unrealistic performance promised once and never checked; or quantities of data that are pure guesswork but held with religious conviction to be unquestionable fact.

So when creating test data you need to look at the specification with your WCPGW glasses on.

Review - More testing

A lot of what we've just looked at has been about getting quality built-in to the original code...

- Logical proof by inspection
- Argument checking
- Keeping strict control over 'our' variables

...Which are matters we should be dealing with at coding time. (Can you see the importance of being able to focus on a small section of code in order to have these matters at the forefront of your mind when writing final code?)

When we started looking at [T]s we were on-a-roll with the crash test program, but as it turned out we were able to convince ourselves that the code was reliable by inspection. Inspection is only a partial answer, it tends to be better at showing why things can't go wrong than why they must go right, so exercising and crash testing are still required.

It all come down to the human skill of WCPGW being assisted by a combination of different tools.

Finishing off

From where we're at currently we'd look at coding to production standard, exercising and testing the timestamp

It seems to be taking a long time but really this is because we're discussing it. When you have done it a few times, and know where each of your tools is, you can rattle through.

281 Also there are some things left as 'understood'. When we say "The house will be painted" we probably mean the window frames but not the glass in them.

third and file handling third of the source code. Then we'd move to the other two classes in their turn.

User documentation

We've talked about documentation already. In this example we have decided the API will be the bulk with an introduction backed with TestClassLog. There shouldn't be much for us to do...
...But we did put some [D] markers in the code that we ought to clear up. For example in SetLoggingSwitch() we underlined the issue of the user not getting any logging because it isn't switched on. If we had a FAQ section this would be a candidate issue but in this case we could just highlight it either in a box or in the introduction with a four line get-you-started example:

```
include('ClassLog.php);
Log::SetLoggingSwitch(TRUE);  // *IMPORTANT*
Log::LogText('Hello Log');
print_r(Log::GetBuffer());
```

The shorter, simpler and more reliable the get-you-started section is the better.

Development documentation

Your working notes are none of my business. You should have seen all the Todos and 'Don't forgets' crossed off and vanish. You may have used your notepad for listing working files so that it becomes an index to the files in the project. It's up to you how you go about recording further development.

One of the tidying up matters is to clarify the status of source code files. Do this in a way that is recognised by the automatic documentation tools. In this case we'd replace the status comment in the header with something like:

```
Version 1.0 : First release 12 May 2006
```

Release

It's that exciting time when the code is about to be officially released. It may be late or you may be tired of the whole thing and can't wait to get onto something more interesting... ...Conditions are ideal for fatal bugs to slip in.

- Don't add finishing touches to the code without
  (a) Exercising and Testing
  (b) Ensuring documentation has been changed to match.
- When building a distribution package
  - Have a specification for the contents
  - Preferably have an automated method of building the package
  - Test the distribution *and it's documentation*.[282] This is surprisingly difficult to do because for a start you need some virgin systems and virgin volunteer clients to have a go in order to highlight the problem areas.

Code management

Just when you thought your filing system was tidy and you've got all your tools tweaked

---

282 I'd like a pound for every download that was missing important files or wouldn't install properly. If I can't get it to work out of the box then that tells me all I need to know about the quality of the programming. Life is too short for debugging other people's applications.

to fetch the right bits with a couple of keystrokes you need to start on another version with added Foo and extra Bar. Version N+1 can't be indiscriminately mixed with version N because (apart from other reasons) N+1 is at the development stage. That's just one scenario. There are all sorts of reasons why all of version 1 needs to be frozen. For example a new version of YCPL appears which your users may start to use. Does everything work as before? Who knows? How will you find out yet still have a master version of your original YCPL which other customers will continue to be use?

Many people find that a good filing system is sufficient but the more people are involved the more you need well understood and observed procedures and automated tools for spotting actual and possible dependencies between application elements and also to find the correct version of a file to work on in a particular case.

We've touched on the program `make` (and variants called something-ake) which could for example automatically regenerate API documentation from source if a source file was modified and could then automatically rebuild the distribution package.[283] Everyone, including the one-man programmer, needs to know about principles of *ake even if they don't use it often.

Review

Up until this chapter we've been concentrating on developing your mental abilities to see how your code works in it's environment. Even when designing programs we were exploring how the world works so we could arrange our programming forces to suit.

In this chapter, possibly as important as all the others put together, we've seen the how a development method can be applied to deliver high quality code.[284]

Here's the basic method in a nutshell. In this chapter we've concentrated on items 3,4 and 5. It's very important to recognise the change in working pattern at the end of stage 3. Until then we were creatively sketching in the drawing office. From then we're getting our hands dirty on the factory floor, operating machinery, testing and assembling components to produce a finished product.

1. Understand the design brief - explore the boundaries
2. Design top down and bottom up
3. Clarify your production blueprint
   - Overall structure
   - Prototype and proof of concept
   - Package of deliverables

---

283 make works out what needs doing as a result of changes. Then it calls other programs as required but it's up to you to provide and configure those programs. If you can't find a suitable 'handle the consequences of changing a bit of source code' program then write one. It's one less thing to think about if you can type "rebuild myproject" knowing that all the dependencies will be followed through and any loose ends highlighted.

284 If suitably adapted you can use this structured approach for almost anything from writing tutorials to procedure manuals - anything where methods need checking, testing and explaining.

4 Tool-up
5 Production
- Pre-production review
- Code in segments
- Exercise segments
- Test segments
- Keep documentation in step
- Release

6 Post-release activities *(To be discussed)*

If you've learnt anything by now it is to *interpret* such lists not to follow them with robotic precision.

Issues

- You won't learn the art of testing in a fortnight!
- The fact that until the end of stage 3 you can't really say how much time and effort will be required to produce the finished result doesn't stop people insisting on a fixed price before letting you get on with stage 1! The same applies to feasibility and additional resources required to run the application.
- There's always the temptation to skip on the testing and exercising. OK so it doubles the effort required in producing production code, but that's the price of quality.

Benefits

- At any stage you have a plan of work. You can see where you are and it's clear what's next on the list of things to do. This is a great aid to rationally planning your efforts and getting others to coordinate theirs.
- You can focus on particular matters without having to deal with broad issues and nitty-gritty code matters at the same time....
- ...In particular you can get stuck into hours of 'head-down' programming.

> HDP is a transcendental state only known to Real Programmers. All the interesting challenges evolve from the program-in-progress without having to put up with blunt tools or distractions.
>
> It is both intensely satisfying and draining. Fitness is required - but after a good session you'll feel like a god as you look down on ordinary mortals.

15. Serving

The last chapter was a tour-de-force of organisation and skill. There was nothing very 'difficult' or technical, simply applying brain-power in a logical and persistent manner.

Now we'll look at some common concepts worthy of discussion that are part of a programmer's stock in trade.

Protocols

> Procedures and protocols are very different.

We've been dealing with functions and methods of objects where there's a simple cause and effect involving one 'black box'. Protocols involve 'conversations' between black boxes. Not only is there a 'language' of interchange but also rules about the correct sequence and options.

Try writing 'buying a pint of beer' in *Beginner and y*ou'll get twisted into knots because there are *two* actors while *Beginner* is a procedure list for one. Here's a possible sequence of messages making a complete transaction:

```
(Publican-P is waiting for next customer-C)
C to P : Do you sell beer?
P to C : Yes
P to C : Are over 18?
C to P : Yes I'm over 18
C to P : What beers do you sell?
P to C : Nelson's Blood, Pucks Folly, Sweet Farmer's Ale, Hotel Porter
C to P : I'll have a pint of Maldon Gold[285]
P to C : Sorry.  Please select from the list.
C to P : I'll have a pint of Hotel Porter (in a jug if possible)
P to C : [Puts glass of beer on counter] £2.40 please
C to P : [Hands over £5 note]
P to C : [Gives £2.60 change]
C top P : Thank you [Takes glass of beer]
(Publican is now ready for next customer)
```

That certainly looks different from anything we've seen so far.

- Messages have a definite direction.
- The *state* of the publican is indicated at the beginning and end. In this protocol the publican's state is important; for example checking someone is old enough to be served alcohol needs to be done before pouring the beer. That is some messages are only appropriate at certain times. *Stateless protocols* (HTTP is an example) don't keep a record of state and so all messages are always acceptable.

---

285 All these beers are lovely and available in the best Maldon pubs.

- From a lifetime of study I can tell you there could be lots of other transactions. For example:
```
(Publican-P is waiting for next customer-C)
C to P : Pint of your strongest beer
P to C : Are over 18?
C to P : Yes I'm over 18
P to C : I don't believe you.  Go elsewhere.
(Publican is now ready for next customer)
```
  Typically a protocol will follow certain patterns but not a single rigid course.
- You'll often come across *negotiation* where the parties discuss what they can offer and what they prefer to arrive at a compromise if one is possible. In our example the list of beers is offered and the customer makes a choice.
- A good protocol has ways of dealing with confusion. What would happen if the customer keeps asking for a beer that isn't available while the publican kept up the same response? You have to be sharp about WCPGW because others aren't or they're using an older version of the protocol which doesn't support some feature. (You also have to be on guard for messages with malicious intent or side effects. Consider:
```
    C to P : (And a) gin and tonic please
    P to C : Ice and lemon?
    C to P : Err..  Hold on I'll just find out
```
  At which point C leaves but P can't serve anybody else. Result: Denial of service.)

Standards and libraries

The first thing to remember about protocols is they must be documented so that both actors 'speak the same language' and adhere to the same rules. So you should be able to track down the information you require to implement your bit, and to understand the limitations. For example HTTP is firmly one-client-request-returns-one-reply so you can't ask for an update to a bit of a page like you might do for a native application.

The second thing to remember is that many protocol definitions incorporate optional items, obsolete items and new items that may not yet be implemented by all the programs you might be conversing with. Also many implementations are either not precise, miss bits out or add extras. Expect some surprises when testing against other implementations.

Fortunately the popular protocols have already had libraries written for them, many times over. Unless you have special needs or you're not too happy about being locked into a fossilised library then save your efforts for dealing with your program and use off-the-shelf routines. You still need to read the protocol documentation to understand what it's all about.

If you have to write your own routines for a protocol then you'll almost certainly need to use separate threads and be very careful about your design. If possible look at the techniques others have used for this protocol and research methods people use for protocols in general.

Think protocol!

Humans are superlative at transactions[286]. We can negotiate the purchase of a pint of beer quite easily. Machines can go through the motions, use rules, even learn, but a huge amount of research then detailed work is needed to design and build working protocols for matters that humans find mundane. The question you may be asking is why should I be urging you to 'think protocol'? The answer is in the first sentence: We've learned to be good at flexible negotiation because it is an extremely valuable skill. It stands to reason that either for program to program communication or human to program communication we will benefit by thinking about "shall I offer Foo now" and "shall I accept Bar now".[287]

You might think of your Thermometer class having a protocol of:

```
Initialise(0)
0a  Ask : Please get ready (Method call)
    Reply   : Status and version message (Method response)
Setup mode(1)
1a  Ask : Which port to monitor. (Method call)
    Reply   : Port has a thermo probe attached or not (Method resp.)
1b  Tell    : Units to use (F or C) (Method call)
    Reply   : Acknowledge (Method response)
1c  Ask : Calibrate port probe to given value (Method call)
    Reply   : Error or probe model number (Method response)
1d  Ask : Switch to run mode  (Method call)
    Reply   : Number of operational ports...
                ... or zero and stay in setup mode  (Method response)
Run mode(2)
2a  Ask : Current temp on given port (Method call)
    Reply   : Reading (Method response)
2b  Ask : Max,Min,Average temp on given port (Method call)
    Reply   : Readings (Method response)
2c  Tell    : Reset Max,Min,Average on port (Method call)
    Reply   : Acknowledge (Method response)
2d  Tell    : Switch to setup mode (Method call)
    Reply   : Status message and number of operational ports
```

This protocol is described here in terms of software calls but might be a dressed-up implementation of a protocol defined by a semiconductor manufacturer which describes the electrical signals required to operate a temperature controller chip.

This is more than the APIs you've already come across that describe what each method does - it explains how to use them together in order that the calling actor can achieve its goals. It's the difference between telling you what the clutch pedal and the gear lever do in terms of mechanical action and telling you to press the clutch - change gear

---

286 Except perhaps monosyllabic teenagers and those who learn their transaction skills from the TV.

287 There has been a huge amount of research into game theory (we're talking about computers playing humans and each other here - not shoot-em up) which has done a lot to encourage research into AI (Artificial intelligence). The object is to look behind the mechanics of transactions to strategic 'thinking'. For example in practice a publican rarely asks "are you 18?" they remember or make an assessment on looks or don't care (or a combination.)

then release the clutch so you can drive faster or slower.[288]

User interface

When we need information from a user we typically ask them a series of questions. We might check these as we go and switch the direction of our questioning as we find out more about them. Or we might present them with a single large form to submit which we check, asking them to correct errors until it is right. You have seen hundreds. What about when some of the required information is already available? What about when that 'already known' information may be wrong, mostly correct or completely correct? What about when the user expects the computer to 'know' the details but for one reason or another is forced to do a lot of input, or is disconcerted that 'the system knows I exist but it can't find me'. You see - we (the computer via the program developer) are interacting and need to appear to have some intelligence. The classic case is where `Jim Davies` types in his name but it can't be found on the database because not only is it recorded surname-initials but also misspelt as `Davis,J`. (So `Davis,Jim` might not work either.)

So as well as the trick of being flexible enough to use message variations we need to be clear what the objectives of the parties are (NB possibly malicious) and their likely strategy. The traditional approach is to force the user to conform to the strictly limited and sequential demand of the computer. It's no so long ago that you'd be asked to type the number of a menu item rather than the initial letter or use a pointer.[289]

"Hey yes of course - the mouse!" what a flexible method of input that is. Flexible but a total dead loss if you want people to operate a typical business interface quickly. Our county library has just installed new software which requires the mouse to operate. So the librarian takes your book, puts it down to wiggle the mouse, picks it up to scan it, puts it down to wiggle the mouse, picks it up to stamp it, puts it down to confirm issue then picks it up to hand across. (And I haven't mentioned scanning the reader card!) Whoever designed this system either didn't know or didn't bother about the key to protocols:

> *Separate the objectives from the method so you can accommodate different ways of interacting according to circumstance.*

- At all stages make it clear what the current objective(s) are. For example `Where shall we send this order to?` How simple is that? `Delivery address` is normally an acceptable shorthand but `Address` is too cryptic and doesn't carry the idea of 'tell me this so I can help you'. This is important when you're offering a user choices or are trying to get them to follow a formal procedure. 'It might be a bit of a bore but at least we're getting somewhere'. The same applies to an API which requires specific initialisation. Your code is doing 'things inside' but somewhere in

---

288 Technical people tend to suffer from trying to explain things in technical 'how it works' terms rather than 'what it does' using concepts familiar to the user. (Further compounded by listing features rather than benefits.)

289 By the way it can be very efficient to bash in "567" from the main menu and arrive exactly where you want to be - but only if you're a hardened user of this particular application. Then some dippy programmer inserts a new menu option and the magic number changes.

the documentation you have to explain what the purpose of initialisation is to the user.[290]

- With program to program (or object to object) interfaces you can usually be restrictive, in the name of being concise and precise, although you often provide options for those that want bells and whistles.

- With an API you're often strict but allow a degree of polymorphism. For example your thermometer object might accept temperatures as numbers moderated by a previously set C/F flag or a string such as "15C" or "53 f". (Or even "15c+-2" for a range.)

- As we discussed many chapters ago when looking at user interfaces, we saw that sometimes it is a really good idea to have a step by step 'wizard' to work through a number of objectives. That's a protocol: 'Give me the order, (OK then) give me the authority, (OK then) give me the delivery address, (OK then) give me contact details'.

- It is now common practice to allow people to click, tab, and use shortcut keys to do the same thing. That's being flexible for the sake of 'usability'.

  This is the tip of an important iceberg: Generally it is recognised to be a good thing to separate display and user interactions from the application's underlying activities. To give a trivial example: Suppose on a screen we have an input box for a telephone number. Should the amount of space we've left ourselves on the screen dictate the number of characters allowed? - Should the fact that we've allocated 10 characters in the database be allowed to limit the number? No. We've put the phone number into the system for 'business' reasons so if 10 characters isn't enough then that means the database *must* be adjusted and some fudge may be needed on the display. We'll look a bit more at this three-tier-architecture later.

- Printed reports and correspondence rely a great deal on protocols that could be more accurately be described as conventions. For example totals come at the bottom of columns[291] while English addresses have house number before the street name - German the other way round. Use the Hot-Tap/Cold-Tap¤ principle to your advantage.

  I continue to be appalled at the awful design of bank statements and utility bills. Important information is hidden on corners, in small print or amongst other data. Why? There is no excuse for this except programmers who threw a bunch of fields at a page without considering properly how people would use the information.

---

290 And what might happen if they don't follow the protocol. (That's a [T] then!)

291 When you think about it there is no reason for a computerised system not to put the totals at the top with a breakdown underneath.

Review

A procedure is simply a mechanism. A protocol is a way of using mechanisms. That implies a 'user' with objectives.

Protocols tend to go through various states although stateless protocols are a good trick if you can do them. Plotting these states is an art, but one well worth working at whenever the opportunity arises because after a while you'll naturally be thinking 'what happens when' which is pretty central to good programming.

The other reason for looking at the world in protocol-tinted glasses is that you are continuously aware of the fascinating myriad of interactions being played out between all living things: How seriously worried does somebody have to be before going to the doctor and what governs that? How does a doctor allocates resources and persuade the patient to follow their plan? How well does the health service communicate as a whole? These are real life issues which everybody feels competent to deal with because of their 'naturalness' but in fact is no qualification at all - Just look at the results! As a Real programmer you'll be involved with making programs that help people step through phases, often writing piggy-in-the-middle programs between parties who in 'the old days' would have met face to face or employed staff to do the communicating and negotiating for them.

If you want to see how protocols are documented, explained, work and evolve then have a look at the SSL (Now TSL) protocol which is relatively simple but illustrates well the issues of negotiation that are quite common to initial establishment of trustworthy communications.

Recall a while back when we were looking at design we usefully took two views of a system the *How* and the *What*? In that case we used those two words to signify "how does it work" and "What does it do". In the case of protocols the *What* part is the same but the *How* part is "How do I get it to do what I want it to do", that is a user-based view.

Client-Server

I don't think you need much telling what a client-server architecture is. A server provides a service to one or more clients. They communicate via some protocol. The server waits for clients to contact it, services the client's requests then goes back to idling or looking after other clients. Classic examples are FTP, DNS, web servers and database servers.

Database example

Let's look at two ways of accessing a database: As you know the data are at the end of it just files. For a single program to update the 20th record of file foo.dat it needs to call some functions which tell it where to look and do the actual reading, do whatever alterations are required then call more functions to write back the record. You can imagine the sorts of functions: LookUpTableParameters(), ReadNthRecord(), WriteNthRecord(), LockNthRecord() and UnLockNthRecord().

- These functions and many others in full detail need to be compiled into the end-user program.

- Every access requires refreshing everything we need to know about the database - because some other program may have been working on the data at a the same time.
- When we need to lock a record it means getting shared access to a lock file.
- If there's a problem with one of the locked transactions from one of the applications and the lock gets set on a record but never gets unset, there is no 'hand of God' to clear the blockage.

These issues give a flavour of the problems with free-for-all, snouts-in-the-trough access.

The client-server solution uses a server program that talks to clients so that the end-user programs never call functions that manipulate data and meta-data.[292] Instead the end-user-program calls server access functions such as ConnectToServer(), DoDatabaseQuery() and GiveMeNextRecordInResultSet().

- You'll see these functions are at a higher level. The nitty-gritty work goes on inside the server. (This makes it a great deal easier to maintain the code.)
- There is no need to keep re-reading shared information about the state of the database. The server knows these things and will probably be keeping them in memory.
- The server is in complete control of locks. If a client 'dies' then it can release any frozen locks.
- When data management operations are to be performed the server is in complete control of interleaving conflicting operations and allocating priorities.

There are plenty more reasons why the client-server architecture suits multi-user databases.

Communication

C/S depends on some channel of communication which requires both ends of the channel to 'talk' the same data-level protocol ('take these bytes') and the same-information-level protocol ('this is what I want you to do'). Until relatively recently this was a tall order. Nowadays we all talk IP, all use ASCII[293], and all have reliable implementations of standard protocols in the form of off the shelf libraries and ready to use clients. Also modern languages are much more 'network and stream friendly'.

For example I can type in an IP address into my web browser to look at a web cam to get pictures, take snapshots and control it. That's a lot of function for very little effort - all provided by layers of standard protocols enabling me to be the client to the web cam's server.

Everything client-server?

One of the great thing about communicating applications is that you can stretch the communications link - either a long way to where the resource is located, or moved to a more suitable machine in a cluster or even on the same machine as the client. Suppose you were measuring temperatures as in the protocol example above. In a lab, where

---

292 "Meta-" means "about". So meta-data is data about data such as file formats and which records are locked.

293 Or exceptionally, Unicode.

the computer is next to the items being measured, it might be convenient to use the embedded-library-of-of-functions model, but what about a production line or communications equipment cabinet on the top of a mountain? You already have the pattern of the protocol - if only you added a communications layer and a bit of management software to make your program into a server it is now infinitely more useful due to it's remote capabilities without losing any of its local utility.[294]

It's worth looking at how you'd implement this in a bit more detail because it raises interesting programming issues.

Server Problem identification exercise

Assuming you've got the API/protocol as given above which provide the *How* for the server, and assuming any simple protocol for communicating with the client, what's stopping you writing a server right now?

PROBLEM_IDENTIFICATION_EXERCISE
0. With hypothetical thermometer protocol
1. Sketch the key messages of a client protocol
2. Sketch management functions
3. How will your program 'connect' with client?
3.1 At data transfer level
3.2 At information exchange level
4. How will your program service multiple clients?

I hope you found that an interesting exercise in pre-design discovery of unknown territory.[295] It looks like there'll be some prototyping going on as soon as we've discovered the techniques to use.

1 Client protocol

You shouldn't have had much problem with this. There's not much more to do than repackage the methods of the thermometer class. However the form this repackaging takes depends on the information-level protocol. One possibility is to take the web-cam approach and implement a micro web server with bastardised features. We might do this if we were building a weather station. Another possibility is to use a 'remote command line'. Simply accept text messages such as `1A 4(newline)` and reply with `1A PORT 4 IS READY(newline)` or `1A PORT 4 NOT RESPONDING`.

The Telnet protocol is specified in "RFC 854" RFCs are one of the series of standards documents for the internet. These are freely available on the net.
• 2616 is HTTP
• 822 is email address specification

294 NB You *will* be doing a really good job of testing this won't you? I expect to be sent up to the top of a mountain if your quality assurance isn't quite what it ought to be.

295 Any fool can list "what we know" but it takes skill to uncover missing links. "There are some serious uncertainties here" might be terribly badly received by the project's gung-ho promoters. Fine! - If there's no problem they won't mind you doubling your overtime rate as it will never be necessary will it? (PS Get it in writing.)

This is sometimes called the Telnet protocol[296]. It's very easy to implement, debug, log and exercise by hand. Furthermore ready made clients and libraries abound. Perhaps instead of `1A` we might use a command word such as `MONITOR` but that's cosmetic. (In the following discussion we'll assume we'll be using Telnet.)

2 Management functions

- Firstly, how will the server start when it's in an unattended cabin up a mountain? Will being added to the list of boot-up applications work in all circumstances? Possibly, but what if the attached equipment needs 5 minutes to stabilise after a power supply interruption? (I mention this because the real world is often lying in wait - So this looks like a [T].)

- Do we need a remote reset capability? Possibly yes. In which case do we need an administrator's password?
- Should we be logging data in case communications are interrupted? Perhaps this is a feature we could add later. Should we be logging usage, users and errors? If so what controls and restrictions should there be?
- Should the server have an alarm capability? Normally a server is passive, waiting for clients to contact it, but is that going to be good enough in the case of fault conditions? Perhaps it could send an email to a service desk or a pager. (In this particular case perhaps we could usefully add an overheating alarm. How much trouble would that be in the scheme of things?)
- Are there any diagnostic modes that might help remote fault finding?

There are lots of ideas here. Although we could be accused of feature-itis, two points are worth making:

- The marginal cost of providing features might be very small
- Most servers will have a practical identical set of issues.

3.1.1 Listening for connections

Ah umm... How does YCPL directly connect to the Internet? This varies from the easy and fully integrated (for example Java) to the impossible (for example Javascript) with most of those that are capable using add-on libraries.

Without going into details, when one computer wants to talk to another using the IP protocol it fires a 'speak to me' message to the IP address of the remote computer with a *port number* added on. (The port bit is like asking for a specific department of a large organisation.) Some port numbers are standard, for example web servers will be found on port 80. The server has to listen on the specified port. The next stage is to establish a conversation with the client. This conversation is called a *socket*. Multiple sockets can exist on a single port just as letters from more than one sender can be placed in the same pigeonhole.

Pseudo code:

```
Initialisation
loop(until some stop condition){
  if(incoming connection on specified port){
```

296 Not entirely accurately.

```
        create a new socket from this connection
        ... use this socket for commands and responses
      }
    }
    Tidy up
```

If we implement this code without threads then we can only accommodate a single client at a time. See below.

3.1.2 Send and receive bytes

There are two main ways of getting bytes to and from a socket.

1 Your code and the socket share a block of memory as a buffer. Typically your program is responsible for reserving and releasing the buffer memory. Your code puts what it want's to send in to the buffer then tells the socket where the buffer starts and how many bytes are to be taken. When receiving bytes you similarly tell the socket where to put any data it receives and how much so you can fish it out. WCPGW is of course that more data arrives than you have allocated space for resulting in *buffer overflow*. Not only is buffer overflow going to lose your data but it can also be used to corrupt your program code in malicious ways.[297]

2 *Streams* are implementations of a stream of bytes. The underlying logical model is a queue with bytes being added at one end while being taken from the other. Typically a stream with complex capabilities will wrap a less complex stream which will in turn wrap a simpler stream. The stream concept may be used to wrap a physical device or memory buffer.

A stream should already be protected against buffer overflow and will have ready to use methods such as Write...ToStream(), AnythingWaitingToBeRead(), Read...FromStream().

Unless there are special reasons, a stream is a simpler and safer bet than a buffer.

3.2.1 reading messages

Part 2 should have given us the method to acquire lines of text. Now we have to do something with those lines. For the purposes of this exercise we'll use the incoming command format of :-

```
digit letter whitespace optional_parameter (newline)
```

For example `1b C` or `2a 1` or `2d`.

Which brings us to the common programming task of *parsing* to split the command into *tokens*, and the related tasks of validating this sort of input and action on it.

The most straightforward method of parsing is to split the text up into sections using some marker, often whitespace¤ or a comma, but possibly a great deal more complex, if matching brackets, nested elements and optional elements are involved. Quite a common format to parse is `Foo = Bar` or `Foo: Bar` where Foo is a key and bar is some value. You can see this pattern appearing above where whitespace is used as the end-

---

297 This works because the overflow bytes can be crafted to replace binary code in nearby memory thus allowing a remote system to alter your program. This is a very common method of attack.

of-key marker.

There are three ways to go about parsing, none of which are quite as simple as you first hope.

1 Hand crafted code for string manipulation.

```
Find whitespace
if (no whitespace) {key=whole line, argument=nil}
else {key=bit before whitespace, argument=bit after}
Parse key into number-letter and validate
Validate argument
```

2 Use a *tokeniser* to give you one syntactical element at a time. A tokeniser is a handy way of being given the 'next bit' of the line. This works well when a lot of similar tokens are split by the same delimiter such as for example words of text split by spaces or data values split by commas.[298]

3 Use *pattern matching* such as *regular expressions*.[299] The following two lines of Javascript illustrate this.

```
regExpn=/^([012])([abcd])\W+(\w+)/i;[300]
arrayOfBits = regExpn.exec("1A 4");   //->1,A,4
```

Pattern matching gets very complex and tricky and there are variations and extensions. However if you can write a bullet-proof pattern (Not easy! [T],[T] and more [T]) the murky details of decoding can be left to the machine.

Advantage : Ready made patterns for validating and parsing standard items such as email addresses, URLs, and phone numbers are easily available in the public domain. There is some hope that most of the bugs will have been detected by quantity of use.

Disadvantage : The parsing either works or it doesn't. This can make it difficult to give informative error messages.

Once the elements have been extracted from the incoming message they can be validated. Often validation depends on context so it may not be possible in the first instance to get much beyond 'could this token appear here'. In our example we will only accept `2n...` messages in mode 2 so that means we can't fully validate without that bit of context being provided.

3.2.2 formatting messages

We *could* return temperatures in roman numerals but perhaps there are better ways. In this case simply formatting the number say with a sign, three digits a decimal point and a single digit to look like for example `+056.7` would be straightforward and in a

---

298 See glossary for WCPGW with CSV¤ files.

299 There are whole languages based on parsing. And you thought regular expressions were incomprehensible! Unfortunately you need to have some grasp of regular expression methods and syntax.

300 Decoded: Must be at start of line - Catch 0,1 or 2 - Catch character a - Skip one or more non-word characters - Catch one or more word characters - ignore case.

format that humans and machines could easily read.

Formatting is a lot easier than parsing, most languages have formatting functions built-in or easily available. Reporting, particularly aligning decimals, and web page display require special additional techniques which involve a good understanding of the mechanism used to produce the images and then encapsulating those in easy to use standard functions.

4 Multiple clients

The pseudo code in 3.1.1 would work providing only one client at a time was to be serviced. Why accept that limitation if we don't have to? For a small amount of extra effort[301] we can bud-off each socket in it's own thread to form the basis of a session per client.

Pseudo code:

```
Initialisation
loop(until some stop condition){
  if(incoming connection on specified port){
    create a new socket from this connection
    create a new session based on this socket
    add new session to list of sessions
    start new session in its own thread // which immediately
  } // returns to here so we continue listening
}
Tidy up
```

Hey presto! A mother 'listening' program with child 'session' programs[302]. There are some wrinkles to be dealt with such as what is acceptable in a multi-user environment and how to manage what's unacceptable. In our example possibly mode 0 and 1 should require administrator privilege which means we need another "can I be administrator" message in the protocol. Or we block 0 and 1 modes unless there is only a single client at the time? Whatever we chose there will be some server-related state to be shared amongst session-related state. ie Class variables.

Review

The techniques discussed above are fundamental to servers and much more besides, and for that reason I recommend you set aside a couple of hours to have a go at the simple server exercise in the appendix. @@@??? That time will pay-back handsomely.

Although, because it is too varied a subject, we haven't looked in detail at formatting output you can see that there's two sides to a communication. If you can define a simple set of easy to interpret messages then you make it easy for both sides of the communication to be programmed.

---

301 Providing threads are easy to use in YCPL

302 Father Christmas and his elves from chapter @@@ all over again.

We glossed over the issue of how to handle tokenised input. There's no big mystery the collected tokens are passed to a function that takes the appropriate actions after validating in context and taking precautions against malicious input. The obvious method is to use a switch (aka. case) control structure along the lines of

```
switch(commandWord){
case "RESET"  : DoReset(); break;
case "LIST"   : DoList(); break;
case "SELECT" : DoSelect(argument); break;
. . .
}
```

But hey! Why not keep a list of functions indexed by command word then we've got a flexible way of calling them and perhaps switching them according to real-time, real-world environment. (For example initialisation might discover that the hardware attached to our server is a particular type and needs to use a particular set of functions. The application as delivered contains functions for all known hardware but adapts as required.)

```
ix = arrayOfFuncs.IndexOf(commandWord);
func = arrayOfFuncs(ix);
ExecuteFunction(func,argument);
```

Many languages allow you to treat functions like any other type and execute them at will by name. Have a look in YCPL's documents as this feature is often buried.

But hey! The language environment itself keeps track of function names so you may well ne able to simplify everything to:

```
ExecuteFunctionByName(commandWord,argument);
```

WCPGW? What if commandWord was say `doOpSysCmd` [303] with argument of `delete *.*`?

It's not much good realising this huge hole after an unfortunate event. You wouldn't want to install software with this capability even if it was 'theoretically' not possible to use it. Not everyone is honest, decent and able to resist temptation.

---

303 Some languages allow you to run operating system commands via a simple function. This may be termed accessing the shell or shelling-out.

16. Security

Encryption

We shouldn't be dealing with encryption until we've looked at security in general, but since this is where most programmers begin we might as well kick off with a little bit of technology.

The basic principle is of course that only somebody with the special key can unlock a message. This is used in two ways:

1 To share information privately between key holders
2 To allow somebody to prove they have a key.

For example

- I send you an encrypted email.
- Then you use your key to read it (usage 1)
- When I phone you up later I can ask you what the message was. If you don't know then perhaps you're an imposter.(usage 2)

"But surely" you ask "Why bother with steps 1 and 2 when all I have to do is phone you up and ask what the key is?" Suppose it is Mr Fisher pretending to be me: "Hello. Just a formality to prove you're the real you, there's been a lot of impostering going on lately and you can't be too careful. If you tell me the key then I'll know you're *you*." Hmmm... With the three step procedure you were able to prove you had the key without showing it to anybody. This is typical of the knotty logic of establishing trust, proving identity and many other aspects of practical security.

Method and key

Which combination of the following is secure?

Armour plated strong room door	or	plywood door
	combined with	
Bicycle chain combination lock with three rings	or	precision engineered combination lock with 50 positions and 10 turns

You're right of course: The key needs to be strong (in this case many combinations) *and* the method (in this case the door) needs to be strong. A weakness in either is fatal.

In computing terms the strength of a key is given by the number of possible combinations it could be. A key of six digits has 1 million possibilities, a password of four alpha numeric characters has 1.6 million (36 to the fourth power) [304] A ten character key has 80 bits so (theoretically, on average) it would take 2^{40} guesses to find

---

304 Four digit PIN numbers on cash cards have 10,000 possibilities - a fact worth keeping in the back of your mind.

the right one[305].

> Electronic data doesn't show any signs of being tampered with so an attacker can work away at leisure. More bad news: You can't tell if the attacker has managed to read your message or not.

As well as good keys you need good *and appropriately used* methods. Luckily a great deal of research has been done and you'll find ready to use, verifiable, code to slot into your application. Typically you give a routine a key and a message to decode (or encode) and you get the plain text (or encrypted) result if it's the right key. Different methods have different characteristics so you'll need to research the subject of *block cyphers* and *stream cyphers* for yourself. If you're tempted to write your own for your very own private purpose

(a) it will be a lot of work to perfect
(b) it will be interesting if you like working through all the possible ways information might leak, and
(c) there are people who get even more pleasure from breaking your method.[306]

In short, if you need something hidden, don't rely on a custom method.

Security by obscurity¤

So what about hiding it in a place where nobody would ever think of looking? There are many practical problems. What about disguising it? Ok until somebody spots there's a funny routine that gets called at 'test security moments' which peeks at this disguised and out of the way data.

- *DON'T depend on security by obscurity*
- Assume your method is known (and therefore the likely attacks)
- Assume your encrypted data is available to the attacker

Therefore: *Protect the key at all costs.*

Check my password

Assuming you have a list or database of users how do you check their passwords and flag them as logged-in? (Stand by for more security trapfalls.)

Method 1:

Store user's passwords in a file (or database record) against the user's name.

```
Alfred : MySekret
Bertha : 123321
Charles : T5F3q$-%
```

WCPGW?

- A hacker manages to read the password file to find Bertha's password so the attacker can pretend to be her, or anybody.
- The password file is stolen. Alfred might use "MySekret" for other purposes. Note that this works even if this is 'last month's' list of passwords.
- The file is altered by adding a new user with a known password
- Charles' password is altered so he can't access the system (*Denial of service*)

---

305 To convert 2^{foo} to 10^{bar} : bar = foo/3.3(ish) (or / 3ish) so 2^{40} is about 10^{12}

306 Though they're often disappointed by the ease of subversion.

Method 2:

Store passwords in an encrypted form, decrypting to match with the one offered at the login prompt.

> *Passwords* of six digits are often dates which might reduce possibilities to hundreds or even one if it's a password for a person and you know their date of birth.

This stops simple exploitation of theft and prevents alteration to a known password *provided* the master key used to encrypt passwords is kept secret. But that's a *big* provided.

- The attacker uses a copy of the same system with the default security settings to break into or fake the file.
- The attacker sets up lots of users with passwords like AAAAAA, BBBBBB to see how the encryption works. Suppose for example all passwords were encrypted on this system by the Rot-13 method,[307] it wouldn't take long for the method to become clear so it's then possible to reverse the encryption *and recover the key*.

Method 3:

Don't store the passwords but a hash of them. You will recall that a hash is a randomish number derived from the data. A *cryptographic hash* appears very random and can't be used in reverse to find the original. By never storing the user's password you can't 'leak' them.[308]

- It doesn't stop somebody setting up say a guest account altering the unencrypted permissions for the account!

Review

Already you can see that encryption is the tip of a very large and wobbly iceberg. Not only does encryption not help against some threats but it can introduce a false sense of security and giver hackers a way into deeper secrets.

In method 2 we invite an attacker to discover the key we use to encrypt passwords. They only have to find out how one plain password becomes an encrypted password to extract the key. If you were in charge of a row of safe deposit boxes would you leave a copy of the master key inside each one?

The moral

Don't try to use security tools until you understand *security models*.

- You'd be amazed at WCPGW. A lot of money has been spent on protecting the wrong things in the wrong way.[309] (@@@where did the idea that pw should be

---

307 Each letter of the alphabet is 'moved along' by 13 places with A wrapping around after Z. So "explain" becomes "rkcynva".

308 You can but not by theft of the data file... ...providing you take precautions.

309 Matched only by the effort big organisations like banks put into covering up mistakes and mayhem (Even when they know a customer is wrongly imprisoned for fraud.)

changed regularly come from? ... effective?)

- I've tried to whet your appetite but there's so much a whole book could be written on the subject. I recommend you immediately purchase *Security Engineering by Ross Anderson*. It's what you need to know, it's very well written, authoritative and interesting. When you've read this catalogue of devious exploits, clever techniques, unexpected flaws and why-don't-people-learn-from-mistakes *then* you'll be ready to chose the appropriate security techniques for your applications. It's required reading and a cornucopia of WCPGW.

Safe applications

Malicious input

One of your tasks is to avoid inputs being used to subvert your program.

Suppose you give your users an opportunity to add a note about their hobbies, latest projects or whatever. WCPGW? What if the malicious user types in:

```
Hacking"; delete from user where userid=123;
```

When you run the SQL query

```
Update USER set hobbies="data from screen" where userid=567
```

the full query becomes (split onto new lines for convenience):

```
Update USER set hobbies="Hacking";
delete from user where id=123;
" where userid=567
```

Which SQL interprets as three separate commands. The hacker doesn't care if the first and third commands work the mischief is in the second. This could of course be 'delete everybody' or a 'give me administrator permissions' or a 'set my account balance to 2000' command. This attack is known as *SQL-injection* and is easy to try, relatively easy to protect against but often ignorance, or 'it will never happen to me' prevails.

Another common form of abuse is when you allow people to send emails. In the address line or message area they might put 1000 recipients.

Malicious input can be used to confuse a system into giving away information useful to a hacker. If your program crashes with the message "Superfoo version 1.2" then that tells me exactly what level of security patches haven't been applied. It might helpfully give a stack dump telling me that "Function ValidateUser failed at 202A5C:6FA024" so directing me to look at a certain place in the binary code for a convenient place to bypass the test by patching the binary. Typical attacks involve giving very long arguments or devious but valid input.

The moral is to validate everything that could be hacked and trap any weird exceptions in a security-neutral way.

Secure security controls

It's quite common to need to validate a user's credentials or licence validity.

Method 1 :

Check at start of program. If fail then exit.

WCPGW?

- The check is bypassed or faked by a hacker. The attacker runs your code inside a debugger stepping through until the entrance to the protection function. All they do is replace the 'jump to check function' with 'no operation' or replace 'if check

function returned OK do foo' with 'if NOT check function...' This can often be done in a few minutes. (Typically used to turn demo versions into full versions.)
- Some permission flag gets left over from a previous session (Often engineered by foul means.)
- Confirmed state for a different session is used in this one. (Equivalent to flashing somebody else's ID card.[310])

Once 'in' the intruder can proceed unchallenged.

Method 2 :
Check at intervals throughout the program.

Method 2a :
Check when specific authority is needed for certain tasks.

This just requires the hacker to either isolate the test routine to bypass it which they've presumably done already or fake it more often. So hardly any advantage over method 1.

This illustrates the futility of adding quantity instead of quality. Five easily breakable locks are no stronger than one easily breakable lock. In this field quality means quite a bit of savvy and appropriate technology.
- If your code can be hacked then you're on shaky ground. But how will you guarantee that your code hasn't been hacked? Do a checksum¤ of the binary code... ...but the checksum part of the program can be subverted. - Gloom!
- If your code is secure (say it is being executed on a server under your control)[311] then you can validate users and their permissions using tokens. There is still the matter of the forged ID.
- What's to stop an eavesdropper discovering somebody's password? It's quite easy over a network. A trojan¤ keyboard snooper might do the same thing.
- A popular method for validating users (and weeding out bots on web pages) is *challenge-response*. The system gives a puzzle which only a genuine user/client can solve. For example : What's your mother's maiden name?[312] What is this number plus the random number we agreed on earlier?

There are very clever technologies such as SSL which protect against some things. I can't tell you which bits of the system you're going to have to look after yourself because of the complex interaction with various security technologies.

Deterrent

Because your latest program is so useful and you're so public spirited you've decided to release it as shareware. Jolly good. There's a message in there which says something like: `Please support shareware - Send ten dollars to me@mydomain.com`. Before

---

310 Read *Security Engineering* for the reality of the effectiveness of photo cards.

311 Secure as far as executable code is concerned. But there's not much to be done to stop people copying HTML web pages and hacking them for their own use. PDF is a bit better as you can theoretically stop alterations.

312 Discuss.

long your very useful program is being used in many places but strangely the message reads : `Buy the professional version for 10 dollars from www.malwarehouse.com`. If you look at the binary of most programs using a hex editor you'll see messages like this in plain text just waiting to be hacked.

A way to combat this is to have a table of messages which are encrypted, being decoded as required. This makes it quite difficult to hack but far from impossible. (If you're considering *internationalisation* then you have message tables anyway.)

Installation security configuration settings

For the deployment of any application there are likely to be well known bad practices which increase the risk of data and source code being leaked. On the one hand each configuration issue required detailed technical knowledge, but on the other they are widely documented.

If you're deploying a server based application then the management of file access permissions depends on the security management features of the server's operating system and any other services (such as a web server). You may find it convenient to run your development system with lose controls, but at some stage these need to be tightened up, tested, and probably most importantly - because they're slippery - documented. A common ploy is to keep data well away from 'executable' code and only allow access to the data directories by the server 'user'.

How do you protect confidential data on a system that's not under your control? By making sure only your application can read the data and that there's some key required to run your application. (Well you've protected it against physical theft - for a time at least.)

> All manuals have a tedious section 'that doesn't apply to your development environment' about how to deal with various threats. You need to investigate, experiment and probably research further.

Review

The management of keys and levels of access to privileges is a knotty problem well beyond the scope of this book. Your job is to do the other bit, making sure it's 'no-key=no-see'. This is a slippery problem which you need to address diligently and persistently.

One of the ways to improve security is to make it easy for users to operate the system in a secure manner. Suppose your program has to be run with full administrator privileges - Not only does this open up the possibility of abuse of your program to do naughty things on the system, but it encourages everyone to know the administrator's password.

17. Assisted development

So now your workshop has been kitted out with raw materials, good practice manuals, and useful tools. There's a database engine sitting in the corner, an old but reliable compiler over there amidst the quiet order/disorder of a craftsman's bench. You've a code of self discipline and a sign on the door `Please do not annoy the genius as withering scorn sometimes offends`. What more could you need?

How about some:

- Jigs and moulds to help construct often used parts
- Ready made sub-assemblies which just need a tweak

In this chapter we'll take a look at some ways you can use to develop applications quickly and reliably. Firstly these are not to everybody's taste. Secondly they can be over-used. Thirdly they can require significant development infrastructure investment and overhead of third party code. Fourthly they may not be mature and reliable. Fifthly you may be locked-in to a code-base with no certain future.

The purpose of this chapter is to make you aware of the possibilities and ideas so that you can consider experimenting. Even if you don't 'buy-in' to what follows it should illuminate the ways you evolve to make development fast and efficient.

Libraries

The first port of call for a programmer in search of a quick solution is to look for ready-made code. This is great if it works.

There are three main problems encountered with libraries:

- The background code base which they work with often isn't exactly the same as yours. For example the library might be written for a later version or different dialect of YCPL, or require further libraries.
- There may be known limitations with the code which is considered a very minor issue by the library author but could blow up into big trouble for you.
- You're often in the dark about the quality of the library code and may have to cross your fingers. Fixing problems may be impossible.

Third-party code may be your only option where you need to operate a specific piece of equipment or work with a proprietary file format. Nowadays there are often wrappers which can make it easier to access the function calls in these libraries in YCPL.

Of course during the course of your development you'll build your own library of useful functions and objects. Recall the extra effort we put into tracing and logging in chapter 13 @@@ because it was obvious that this code would be used over and over again.

Standards

Although not in themselves tools for making software, standards can be useful ready-made specifications.

- They provide a reference for inter-operation
  - Specifications for your software to meet
  - Specifications you're expecting 3rd party components to meet

  It's quite rare for any software to completely implement every single feature of a standard. This doesn't stop the marketing wallahs making claims.

  You also get a reference to the non-standard bits that you might like to exploit for good pragmatic reasons. So long as you know where standard ends and proprietary extension (or fudge) begins, you can make an informed judgement on the significance of this deviation.

- They expand availability of inter-operating software. For example a program that relies on SQL should be able to interface with a dozen databases with minimal alterations.

- There are often ready-made tools for testing compliance, for example HTML checkers. You may be able to quickly put together an exerciser for a 'standard' component.

- With any luck there will be many other users of the component (attracted to it by being convenient to interface with) who have already done field testing. Where there is a definite standard a bug report of 'it doesn't do what it should' is more likely to have to be addressed.

Standards are not essential, often they aren't even standard. Some seem to be more fashionable than useful and can easily be used as a magic (but inappropriate) answer or for the purposes of adding confusion.[313]

> Special note : 'Quality standards' of the `Foo:9000` sort are practically a scam. Either
>
> - you understand, live and breath quality
>
> or
>
> - you go through the motions and tick the boxes .

Design patterns

If you want to build a go-cart, you immediately get the picture of some box with an old pram wheel at each corner. If you want to build a sledge you immediately get the picture of some chassis with two curved runners beneath. These are *design patterns* for

---

313 Many years ago at a job interview I was asked out of the blue what the ISO character set was. Result : A complete blank. Of course later I twigged what the twit was talking about.

having fun going down hill. Once you've looked outside at the weather you have enough information to select which is appropriate. You can now start on the details, look for construction plans catalogued under 'Go-cart' or 'Sledge', and discuss the matter with others knowing they have the same outline model in their heads. The same wouldn't apply if you tried to build a rocking horse on roller skates. Not only would you have to build it without any obvious guidance but it might turn out not to work very well either.

Design patterns are engineering sketches - not quite blueprints - that have been found to be handy solutions for common problems. Well... umm... more like ideas that designers have implemented over and over again and so to save worrying at the details (such as they may be) you can use this template of ideas.

It's always a good thing to look at how others have tackled similar problems and this is what design patterns allow you to do. Researching them should broaden your knowledge of possible approaches to problem solving and the more choices you have, and the better informed you are about their relative merits, the better. If I was to give you a single example of a design pattern it would confuse you. You need to see a herd of them gambolling together across the dry plains of computer-science-space to get the flavour.[314] Good luck with that - You won't find the abstractions and cross references straight forward. Persevere.[315]

Layers

You will probably come across the Model-View-Controller (MVC) pattern.[316] This is an example of a layered architecture. Think back to when we were looking at design methodology. We had a top-down and bottom-up approach, with the former leading us to split our solution into logical components. Perhaps the most obvious way of doing this splitting is by application function; for example the administrator's tasks and the day-to-day tasks. Another way, which is not so obvious, is by nature of the software mechanisms; for example (a) the database bits, (b) the user interface bits and (c) the business logic. (This is like splitting the engine, luggage and passenger compartment of a car - It's a lot easier if the mechanical gubbins are put in one place and you don't have shopping bags round your feet while you're driving.) These are not alternatives - you can sketch a table with application functions in columns and software realms in rows to consider both.

Let us consider an object that displays itself on the screen as a button. Let us say in our

---

314 My reticence stems from the religious fervour with which some people promote design patterns - some of us have been programming quite successfully for many years without a needing a pantheon of text-book lessons. (In case you haven't yet twigged, I belong to the 'an engineer is an artist who will use every bit of technology to solve real problems really well' not a 'make way I'm a scientist - I have the technology and I'm not afraid to use it' school.)

315 Technical terminology (Technology) is always important where science meets engineering. The same would apply if you were looking at the chemistry of dyes, or physics of high temperature ceramics or forensics of archeology.

316 There are variants as people find their own ways of doing approximately the same thing.

application we want to use it to delete the displayed record. Here is what we could program in Fudge:

```
class DeleteButton extends Button{
  integer : jobID;   // we'll delete jobno this if clicked
  constructor(){
    this.caption = "Delete";
    this.jobID = 0;
  }
  method SetJobID(integer:ID){
    this.jobID=ID;
  }
  method OnClick(){
    db = new DatabaseConnection();  // fudge! No parameters.
    db.Execute("delete from jobs where jobno=" + this.jobID);
  }
}
```

With care this code might work but is it really an example of good programming?

- jobID/jobno : Shame there's no consistency in naming
- There's a fair chance of forgetting to call SetJobID()
- At first glance it looks like a good thing the database code is snugly integrated with the clicking event. There's no mistaking looking at the OnClick() event what's going on...
  ...Some quite serious database initialisation (the password authentication arguments have been left out for simplicity)... some important business logic... and updating the database.
- We may need to replicate this code if there are shortcut keys or automated procedures.

From a design point of view we might want to *decouple* the display of a button from the database and the business logic. For example if we wanted to use a different database we'd have to slog through all the code of all the screens making alterations. If we wanted to be able to switch between databases (quite common and a good selling point) then either we're going to have to have multiple sets of source code which will need keeping in step, or put switching logic into every control of every screen. On the other hand, if we can treat the database operations as entirely separate from the visual display, just joined by a standard protocol or interface 'glue' layer we don't need to be messing around with screens when we're really working on the database. The converse applies. If we found a lot of deletions going on 'by mistake' then we might add 'are you sure' logic. Now we've got to worry about display stuff and also database stuff.

Dividing designs like this is often called the Model-View-Controller (MVC) pattern or three tiered architecture. The designer develops the database as required to support the application without reference to the display. On top of this fundamental layer is how and when and under what restrictions the data will be manipulated. Finally the bit the users sees is added.

Patterns such as MVC can be anything from ideas to keep in mind, a way to organise your development process, through to a formal structure for automatically building code. Don't fall into the trap of the latter without the former.

Templates

Code templates

One of the best database programs I ever used was based on code templates that generated code for common activities such as displaying a table then providing standard keys to insert edit and delete records. Not only could the resulting code be tweaked if there was something special on a particular screen but the template itself could be altered so that code generated would incorporate that alteration.

Nowadays with OOP and frameworks (next section), there should rarely be a need for cut-and-paste-then-edit in code.

Aside : Working templates

You may find cut-and-paste templates useful for

- Developing documentation by hand. People like a standard format. Often there will be a lot of similarities between pages of help screens.
- Re-using development administration tools.

Aside : Data templates

With some simple text replacement routines you can save a lot of coding and provide flexibility when

- Implementing many variations of text messages.
- Allowing users to define layout of reports etc.
- Defining report layouts etc. yourself.

For example your application might send out a number of emails. You'd expect the bulk of the text to be the same but with some specific insertions. You could hard-code these message texts but if for some reason the body of the text needs to be adjusted it requires all the delay, expense and paraphernalia of a programmer grovelling in the code instead of a simple hack with a text editor, or even better providing a routine for the template text to be stored in a database with a simple editing interface.

Frameworks

If, using our workshop analogy, libraries are brought-in sub-assemblies, and design patterns are engineering concepts, then we still haven't got jigs. (Jigs are used throughout manufacturing as a scaffolding to hold components as they are being machined and assembled. It's a sort of specialised tool which is customised to fit the article being manufactured. For example window frames are assembled by being clamped in rectangular jigs with sides of adjustable length. Imagine the difficulty of trying to keep all corners at exactly 90 degrees by hand - now add to that having to locate drill for the fixing pins to the nearest half millimetre.)

Frameworks are a software equivalent of manufacturing aids that:

- Do the boring and repetitive stuff
- Speed production by rapid preparation and assembly
- Make sure the components align properly
- Can be fed specifications to produce finished or semi-finished parts.

Although in theory the whole development process could be supported, a framework usually limits itself to automating code production. As I write, in 2006, frameworks are becoming very fashionable with a host of evolving schemes. "Evolving" being a

euphemism for 'not yet mature enough to bet the farm on'. The big trouble is that you need to make a large investment to get going and another large investment to discover the limitations and how to work around them.[317] Once you build a real application using a framework you are committed to maintaining that scheme in good working order during the lifetime of the application.

> My personal preferred approach is to keep all code live and not to use code-generating frameworks. I put the 'plumbing' in classes then create applications at the highest level possible with the appropriate constructors and parameters. In practice this means a large class library and coding the UI with parameters taken from the database rather than specifying the database and getting the framework to create the CRUD¤ screens.

Frameworks are intended for building business applications from business process and business data specifications. For example you might spend a morning talking to a client about their operations, a couple of hours knocking up a database and by tea time have a prototype.[318]

> In your early years as a real programmer you have to establish good standards of workmanship before you allow a program make your code for you.
>
> (In the early days, code generators will do a better job than you will for straightforward business applications, and they're getting better all the time.)

Review

Frameworks: Automating repetitious tasks is second nature to programmers so it's not surprising that a lot of effort has gone into automating code production. It's certainly impressive when you hand over a database specification to an application builder and in 30 seconds you have dozens of fully functional CRUD¤ screens and a working database. (That 30 seconds is the tip of hours of setting-up, configuring and hair-tearing but you get the idea.) Frameworks can be thought of as automated templates.

---

317 Warning: Some frameworks assume they are the only kid on the block. Even if you can get them properly configured on your system to the Hello World stage could have trouble getting more than one to run at a time. I found good intentions (if you share the framework developer's view of the ideal design) let down by poor implementation.

318 The value of a prototype is that it gets the client involved in the problem identification and solution exploration. Immediately there is light at the end of their tunnel. Immediately you get a vote of confidence and allowed to see the really important details that they don't talk to strangers about. Tip. If possible do this over a weekend - It gives you a bit more time to mull over issues and the client will be impressed that you worked over the weekend just for them. Tip No 2: Don't let yourself get carried away also. The prototype is just a broad sketch for the purpose of communicating with the client not a design in itself.

Templates of data: Try to avoid hard coding anything that might be subject to alteration. The alternative may be a database table of configuration parameters or (the conceptual opposite of) so-called constants such as messages, email shell texts, or data layout specifications.

Templates for development processes: I've deliberately avoided tabulating the steps involved in developing software.[319] There were been plenty of hints in chapter @@@ and I want you to evolve your very own set of considered rules.[320] I expect you can see that there are obvious connections between code and testing which should at the very least lead to "new method Foo() suggests there ought to be either a test for Foo() or an inspector's[321] approval." Use templates like a checklist to save you having to remember all the steps and snags between idea and implementation.

Design patterns : It's well worth browsing an encyclopaedia of design patterns from time to time in order to refresh yourself about ready-tried solutions. With any luck this will raise useful questions about your current sketch outline. Patterns can get a bit academic which is why they are often illustrated with examples; which are there to crystallise your thoughts not as instant-solutions.

Mature Real Programmers are unlikely to get on well with 'instant code generator' frameworks. However they will probably have three things to help them produce applications efficiently:

- Development tools created and collected over time and proved in use.
- Extensive libraries of powerful code modules
- A realistic appreciation of what should, could and shouldn't be automated.

---

319 Because there is no 'one-right-way'. Even if there was emphasis and detail would vary enormously. For example in chapter @@@ I walked through a lot of testing of code from a whitebox, and blackbox point of view but nothing at all about testing in the field.

320 CONSIDERED RULES: I've walked off a site - that's serious - when basic safety measures were being ignored. I did that because I'd taken trouble to be informed about the shambles and evaluate the consequences. In a software development context you might insist as a junior that a senior properly checks your work, or conversely that nothing 'going out under your name' does so by a series of shortcuts that "we'll pretend never happened". This is part of being a Real Programmer - You have the confidence that comes from knowledge to stick to your guns no matter what. (You'll come across plenty of ignorant idiots - if you can't beat them, leave them - immediately - believe me you have nothing to gain by compromising with a snake - leave and explore greener pastures.)

321 Literally - somebody who inspects the code and is satisfied nothing-CPGW.

18. Get a life

That's the end of the technical part, there are some more morsels in the glossary and of course a huge amount of specialist and overview information on the Internet. (So long as you don't believe everything that you read there it's a fantastic resource and continues to develop at an amazing pace.)[322] Quite likely there will be big changes in the way programmers work as a result, already in 2006 we can see a great deal more of cooperative projects. Perhaps, who knows, this will lead to the virtual software factory becoming a standard model for development, deployment and employment.

This chapter looks at the human attributes you need as a programmer and how to exploit them. On one hand you need to be quick witted, knowledgeable and have the confidence to stand your ground; but on the other relaxed, be able to listen and to work with other people to get the best out of them.

Is being good enough?

An ordinary decent programmer(ODP) will have a certain fluency in one or more languages, basic design ability and diligence[323]. Typically they will adopt techniques and stick to them once they've found something that works.

All hail to ODPs! The world needs lots of 'fairly reliable - if limited' programmers. Dear reader, if you've got this far you're probably ahead of the game already: The extra you have is that you can see 'fairly reliable - if limited' is not a high enough target to aim at. You are champing at the bit to do research, pick up experience and experiment with techniques and sample projects.[324]

Recall in chapter @@@ and appendix @@@ we discussed Bad Good Best I told you that 'Good' was the 'Average OK' category but 'Best' was the cream. Even if you expect to program as a hobby you'll still benefit from the programmer's 'super-hero' skills discussed in a little while (as well as having cute and vice-free programs).

Brains

Programming at the higher levels requires a trained and fit mind. Your brain must be regularly worked hard (and similarly relaxed) so you don't crumble under the pressure of the tough intellectual demands that go with programming. Anyone training for sport will tell you that the hard bit is starting when everything aches and you soon get exhausted... ...But if you keep it up you'll soon find the introductory tasks easy. (That's what this book has done. If you go back to the early chapters you won't find them nearly as challenging as they were to begin with. Congratulations.)

---

322 It's so difficult to remember what it was like just a few years ago without an Internet - A bit like trying to recall when you didn't like beer.

323 At least they'll *try* to produce quality code.

324 Another way to put this is that ODPs wouldn't have bothered to read the book.

Real Programmers are professional intellectual athletes, and stand out from the crowd of couch potatoes in the same way as other athletes. You'll be told 'how brainy you are' and considered odd by those that don't appreciate your powers, endurance and particular principles. "Sorry I can't work late - I've got a training session for next weekend's race" is something you might hear a marathon runner say. "What! You cycled 10 miles here today!" is something I get all the time.[325] The RP equivalents could be "Sorry. I'm going sailing tomorrow - There's no point in me starting until the user comes clean on their real requirements" and "What! You worked through the night!"

The trouble with brain-exercise is that you can't sensibly measure kilometres covered or best times. Don't be fooled by simplistic IQ testing. For a start it doesn't measure your comprehension and knowledge acquisition technique and endurance. Some people are good at teasing out small scale detail while others are good at quickly assessing big issues. (Obviously you should try to be adequate at both. For example it helps if the words in your neat summary are spelled correctly.)

Checklist of brain fitness tips

- Get your eyes tested : Poor vision makes long reading tasks very fatiguing.[326]
- If you find memory tasks difficult use a technique. Many people don't need detailed memory much for programming or design - That stuff gets written down for the very reason that detailed memory is subject to error. Mostly you'll need to remember people's names and jobs. Practice whenever you meet a group.[327]
- Mental arithmetic and back of envelope calculations are good 'keep in trim' exercises. Being quick, providing you're careful, can be very impressive and also trap disconnections with the real world and wacky estimates. Division is particularly important as estimates thrown at you such as "The total daily will be N"' can be looked at on a per hour or a per product or per district or per person basis which might raise a few eyebrows.
- Distinguish between closed and open ended challenges. In the first you're trying to get a 'right or best answer' while in the second you're creating possible solutions. These are somewhat contrary as one is trying to restrict answers while the other is trying to expand them.
- Some puzzles rely on finding the right method of solution - Once you hit on the method you can derive the answer, so the puzzle *is the method*. There are some awful quiz and 'test-your-IQ' books that don't realise this.
- Just because you're a professional problem solver doesn't mean you have to be good at working out it was the butler all along. Problem solving ability is very

---

325 10 miles each way to the pub is a pathetic distance - And I'm NOT an athlete.

326 On a general note if long hours give you back ache, wrist pain, headaches or similar symptoms then *do something about it* - Don't take a shrug from management or "If we let you have a trackball instead of a mouse the everyone will want one" as an answer.

327 One of the reason for naming conventions is to be able to translate the idea into a likely name. For example the popular 'functions start with a verb' convention will lead you to look first in the manual for ReplaceString() rather than StringReplace(). One less thing to remember... ...Except in my experience this is honoured more in the breach than the observance. Arghh!

specific to realms - You may be good at converting an outline design into layers and components but hopeless at working out horse racing odds.

- Playing (and inventing) games is a traditional programmer activity.[328] There are three useful brain skills here:
  - Efficiently converting a lot of data into significant patterns.
  - Developing strategies.
  - Getting inside the minds of the opponents.
- Try to be conscious of your degree of focus and also the number of things you're juggling with at the same time. *Don't be afraid to offload some items onto a scratch pad, there are no medals for forgetting.*[329]
- Alcohol and caffeine will affect your mental abilities: Be aware of your actual reaction and their delayed effects. There's nothing 'wrong' with these substances. On the contrary, you may find a bottle of wine makes initial discussions more relaxed and creative. As a professional should you have a tool-box of a few tricks for 'thinking around the subject', but others probably have no experience of this at all. So you may find blunting their prejudices with alcohol and bonhomie necessary. Alcohol is a fun way to switch-off the strictly correct analytical mind and watering the what-if, who-cares-about-details, creative mind.
- Prescription or over the counter drugs can have unexpected side effects. Watch out for them and be prepared to wind down your work schedule a notch or two.
- Physical fitness and good health is a big help for the person who wants to apply their mind to problems in detail for long periods.
- *Study*: Your maths teacher wasn't really expecting you to be spending the rest of your life working with sines, cosines and tangents; their cunning plan was for you to learn how to accumulate and assimilate knowledge with precision. Then your history teacher made you write essays in order to acquire the habit of analysis, and synthesis. *These are basic knowledge manipulation skills for professionals - If you're not fluent then practice by blogging or writing a diary or editing a newsletter.*[330]
- Even top athletes like Tour de France riders have rest days and spend most of the race doing the minimum necessary to keep up. It's a fantastic feeling rattling through problems, finding the 'right' answers and cracking worrying issues but when brain-fatigue sets in there's simply no point in you carrying on. You must learn to pace yourself and stop before your brains turn to mashed potato.
  - *Don't be afraid to call 'time-out' - It's one of those things that is difficult to get across to ordinary people because mental exhaustion doesn't show up like sweat and muscle cramps.* Obviously it is better to have planned breaks where convenient, but why not work through the night when you've got a 'following

---

328 Games are often used for programming practice - leading you to explore aspects you wouldn't usually come across. A lot of pioneering work with artificial intelligence and languages supporting flexible data structures for knowledge representation was stimulated by gaming.

329 Absent minded professors don't get Nobel prizes for being absent minded.

330 Motto: Read lots of complicated stuff - Write a little clearly.

wind'?[331]

- Relaxation can be difficult - Try doing *nothing* if you don't believe me. Without a doubt you need an environment where the distractions don't require you to apply your mind to 'matters in hand' - so no phone calls or colleagues dropping by for gossip. Leafing through a seed catalogue and planning your garden, inconsequential chit chat with strangers in a pub, doing the crossword, bricklaying[332] and hiking are all examples where you are simply doing something else of no particular consequence.
- Take special notice of the precise use and mis-use of words. Not only is this an amusing intellectual exercise but a vital part of survival. Does the user *want* or *need* a feature? How often do you see *activity* confused with *action*? *Often* - What does that mean? You can use a mental highlighter when listening to management-speak, sales-speak and geek-speak then go back to these items when, after due consideration you are asked for your professional opinion. Do this a couple of times and you'll soon be on the bullshitter's blacklist.[333]

Coding

What goes wrong

There are many situations where a programmer given a 'complete' specification and told to turn it into code with the design aspects limited to the odd internal data structure. This might be good use of your talents if there was a particularly tricky algorithm to implement but otherwise just hours of mind-numbingly tedious typing. Soon the 'as soon as I can get this to work I'll declare it finished' mentality sets in.

There are many programmers who would rather spend their time polishing code, looking for the most complex algorithms and object models, creating spiffy development tools and giving geeks a bad name rather than delivering a finished result.

Then there are top-heavy development environments where programmers are expected to use baroque object models and over engineered tools in order to 'produce quality code through corporate standards'.

To understand why these are lopsided ways of doing the job means understanding the skills of the programmer, needs of the project and suitability of the development environment.

Coding in perspective

All the other activities described in this book are peripheral to creating finished code.

---

331 When I was nine-to-fiveing I always had at least a 45minute lunch break and in the 5pm to 9pm evenings when everyone else had gone home I'd switch to something different and do a proper day's work.

332 I can vouch for this soothing pastime as could Winston Churchill. Physical achievements can give great satisfaction when your everyday work is abstract.

333 Take it as a compliment. See how quickly you can achieve the high ground of fact-based decision making. "Excuse me can you tell me where that figure comes from?"

Having said that, those other activities are essential to good programming and together will take roughly the same amount of time.

(@@@ add a table on the end of the code/test chapter ??? somewhere else? to illustrate sequence of events.)

Of course coding shouldn't happen in isolation from the problem identification-specification-design-feasibility stages and the check-document-test-deliver stages. But some programmers are not competent to tackle all these aspects on their own, often the size of the whole task is too much for one person. Discussion amongst technically experienced colleagues can be useful to develop and refine ideas. So some assignment of skills to tasks will be required. Also there will be some sharing of work in progress, and the need for good communications, if there is to be efficient cooperation. Clearly this requires a controlling mind and teamwork - A subject we'll look at in more detail in a moment.

Coding standards and infrastructure

Ask a dozen programmers to produce 'the same' code in the same language and you'll get a dozen different programs, taking different times to write, with different strengths and weaknesses. Now swap the code around so that each of the dozen has to work with another's code. Uproar ensues as each reacts to 'stupidity', 'unnecessary complication', 'poor this', 'bad that', reinventing the wheel, and generally *being different*.

Coding standards are an attempt to make it easier for programmers to share each other's code. As you've guessed there is no standard standard. For example I've used `[T]` as a 'needs to be tested' flag but you're unlikely to see that used elsewhere. (@@@APPX my coding standard??) Naming conventions, comments and layout of braces are usually at the heart of these holy wars¤. However there are good reasons for establishing and sticking to conventions in a development environment.

- Making code easier to work with
  - Automated tools are able to read and interpret the code. For example I like to have some code libraries sorted alphabetically but when adding to them I work at the end of the file and use a sorter once I've finished messing about with the new bits. My PHP documenter can extract all sorts of useful information if it's carefully formatted.
  - Programmers can share meta-information (such as the [T] for test-me!)
  - Programmers can interpret embedded information. For example knowing immediately that (in *my* standard naming convention) `Foo()` is a function or method, `Foo` is a parameter and `foo` is a local variable.
  - Programmers can quickly navigate through code. For example it might be your convention to put constructors at the top of a class followed by Get...() and Set...() methods and so on.

- Standard workshop practice for code quality
  - Using standard routines rather than hacks.
  - Using (or not using) certain types.
  - Using certain levels of code checking as a matter of policy.
  - Using (or not using) some techniques or programming strategies. For example

it might be 'house policy' not to embed message strings in order that they may later be internationalised in a modular fashion.
- Avoiding, or exploiting, or automatically switching at run time, or manually selecting at compile time, dialect-specific or version-specific language features.

- Common tools
- Common filing system

Working with others

Real programmers will be streets ahead of dumb users, clueless colleagues and idiot managers. Err... There's a fair bit of truth in that statement, but while they may be dumb, clueless and idiotic you still have to work with them, so letting on might not be the most successful strategy.

Most of the time you just have to be patient while making encouraging noises to help them to catch up with the analysis you arrived at a while ago. Avoid getting embroiled in discussions as a general bod to bounce ideas off. Instead be sure what purpose you are serving at a meeting and stick to that field.

> "Hey Peter. What would you do then?" can be quicksand. Be firm enough to finish off the discussion along the lines of "X didn't work last time - what's changed?" or "It seems you can't go further until you have more data."

On one hand I make it a rule never to attend a meeting without an agreed agenda particularly meetings where there's a danger of decisions being made. On the other hand informal contacts are vital. These really work if you can start by being helpful.[334]

Users

- If you get users talking about themselves that makes them happy and you may pick up some key phrases which when fed-back makes them think you really care.[335]
- Do as much listening and as little talking as possible. Make it easy for them to tell you things. How do you do that? By nodding, saying 'yes', by sympathising with their personal problems and congratulating them on their achievements.
- Your notebook is a solemn badge of office. A fancy fountain pen goes well with it. Try a hat, bow tie and silk waistcoat. People love cheerful eccentricities because

---

334 You can of course be helpful on a personal as well as professional level. "I've got the name of that show you were asking about" or simply "borrow my umbrella". A little bit of social respect goes a long way. In the days of typing pools and data entry departments you were *always* helpful and courteous to the lady in charge - Not only did your work jump to the front of the queue but the look of disbelief on other people's faces when your work was hand delivered with a smile was worth a fortune. Go out of your way to be charming - Practice.

335 I once got a job by 'being interested in' Hydrogenation having seen a "No naked lights Hydrogenation plant' notice on my way in. Fortune tellers use the same tricks.

you're probably the most interesting thing that's arrived in their office for ages.

- If explaining something to users then use their language. Try not to use abstract concepts, instead use practical examples and props. Always set the scene, or better still get them to set it.
- Try to get the users to express their issues themselves with open ended questions. Suppose you were considering where to put the 'chase late payers' screens and what degree of automation was appropriate and what sort of blacklisting to use.
  Bad: "I suppose you have problems with bad payers?" For which you get a 'yes a bit' a 'no not really' or a disagreement.[336] That's not very useful is it.
  Good: "Does that always work smoothly?" Which opens the way for an examination of the issues in general. Is credit being given to the wrong people? Is follow-up slack or difficult? ... "How could that be improved?" This then leads naturally onto a quantitive exercise of how many, how much, and so on; so *between you* you can clarify the priorities and how much the computer system should do.
- When things go wrong, even (or especially) if it's not your personal fault then
  - Never panic, but if there's something *practical* that ought to be done immediately then get onto it. Informing the management can come later.
  - Make a realistic assessment of the actual consequences before discussing it.
  - Make it clear that your job is to clarify and correct. Others will do damage limitation and look for blame.

> It's really useful to be trusted and approachable. You'd be amazed at how difficult it is to get users to report bugs and other problems - even when you've got a formal bedding-in period.
>
> There is no better way to achieve this than being a knowledgeable team player 'on our side' when the going gets tough.

Colleagues

Your colleagues may have the same opinion of you as you have of them! There's a lot of 'my way is the one true way' and 'I haven't got time for that fancy stuff', 'we've managed without a crash test procedure until now' not to mention pure stylistic bitching.

- Geeks soon cluster together and evolve petty status symbols and codes, often based on arcane trivia or expertise, which make you wonder how pointless the rest of their lives must be. So long as you recognise this knowledge isn't wisdom and reaching the 53rd level of *Dungeons of Zog* tells you more about someone than they'd be proud to admit, this badinage is harmless. In fact you can use it to weigh up the calibre and characters of your colleagues quite quickly...
- ...As can asking relevant technical questions. Be careful not to fall into the geek-trap of evaluating technology or techniques aloud. There may be plenty of programmers around who are as critical as you are,[337] and if you subtly signal

---

336 Some of the players may be very sensitive about bad debts - How they occur and who'll get the blame if the true extent is found out. This is a good example of X-Ray vision. You might have to deviously put some 'stop this silliness' into your program.

337 Quite possibly because they have their own, different, prejudices.

you're thinking this over might show their thoughts, but there's no point in opening a can of worms unless you have a fully thought through alternative that you can articulate or present as a fait accompli.

- Don't be hasty: 'Not the same as what you're used to' doesn't necessarily mean wrong. Check the BGB¤ scale. While you want to be in the *Best* category and need the environment to perform at that level you can make a case for that fairly easily.[338] Where you see *Bad* you'll need a considered, and if possible easy (or look here's one I've done to show you) solution.
- The majority of you colleagues will be in the *Good* category. Adequate. Don't be afraid to call out the *Bad* ones when you've got evidence of utterly unacceptable performance. If you're supposed to work with this person make it clear what your conditions are in writing.
- One way of showing how smoothly you integrate and calmly you analyse issues is to keep written records. Not to excess but the sort of stuff that is useful for quality systems and project management. (Even, or especially, if there is no quality system or project management to speak of.)
- Remember that for the most part programming is a solitary intellectual pursuit. Demand an environment where you can get on with your job. In my experience I can do twice as much work in my study at home as in a client's office. Not only do offices provide distractions and encourage interruptions but you can't pace your work according to your mental schedule. When you want to do 'head-down' programming make absolutely sure that nuisances including the management know that DO NOT DISTURB means what it says. If necessary turn up early and leave early so you can get in a couple of hours on your own.[339]

Hone your inter-personal skills - Programmers are a difficult bunch to deal with and there's no escape if you can't cut the mustard yourself. Most of the above only works if you really are good and obviously in the *Best* category. Typically you'll have been drafted into a project for some purpose. That purpose is what management will be focussing on even if your wisdom is well applied elsewhere.

Management

Managers are like children:

- They demand attention
- They don't understand the meaning of 'No'
- They make unexpected demands
- They won't be put off by reasoning
- They assume your purpose in life is to work for them

Unfortunately "Go to your room" [340] doesn't work - while smacking is so undignified.

- Don't under any circumstances be pressurised into giving an opinion or commitment until you have had (a) the relevant facts and (b) time to consider them.

---

338 But heaven help you if you try being a prima donna without delivering what's on you bill material.

339 Or save up some tangled stuff for after the others have gone home.

340 Management anger therapy occasionally works a treat - but be careful!

If you're being asked for a professional opinion of feasibility or suitability or cost etc. then insist on access to the facts in writing so you can study them, and in your written response quote them back. If you're being asked for a rough first guess opinion then make sure your response is in writing marked "FIRST GUESS" and list the unknowns. (@@@APPX paper tools O-M/Factors) *This is good practice anyway of course - The point is not to be bullied by management into guessing.*

- If you're being asked "Am I right? - I have this meeting in a minute and I know how particular you are" (Subtext: "I'll be spending your hard-earned trust to save my face and I want to go through the motions of consulting you first.") If you have reservations then bluntly say "I have reservations. There are issues still to be addressed." If you think the manager is poorly briefed (or hasn't bothered to read your report) then brief them in writing (or suggest they hand round your report) so they at least have a crib sheet. *Whether they came to you out of deviousness or genuine ignorance the professional thing is to make a clear summary...*
- ...Ensuring you distinguish between uncertainties, facts, opinions, and recommendations. *This is standard practice but even more important when under pressure with the possibility of fudging and fumbling about to happen.*
- Make it clear that you're not on-call out of working hours to deal with things that should have been sorted out before. *ie Don't be bullied into accepting feeble time management by managers.*

Q: As a real programmer you will have discovered out how things work, how they should work, WGPGW and split the *What* do we want to achieve from *How* will we achieve it. Also you're quite likely to know there's 'boggy ground ahead'. When who should appear but someone of that tribe of institutional ignorance, collective optimism and perennial panic called management, to persuade you that you've got it all wrong. Worse still you're invited to attend a meeting. What do you do?

A: Use the written word. With any luck you will have prompted this situation with a well written report which has ruffled a few feathers. If you haven't then ask to see the report not the waffle.[341] If there is no writing then you can choose between an *informal* discussion[342] or putting something down on paper. Note: If "there's nothing to put on the paper" then what are people basing decisions on? Tarot cards?

Avoid an inquisition at all costs unless you are fully prepared. Firstly you need an agenda before the event[343]. Second you'll want to circulate your paper in good time. Of course these two matters are standard *good* practice... ...so ask yourself why, and who's going to lose out by *Bad* practice. The obvious practical benefits of prior circulation are that your audience should be informed, at least they may have

---

341 This is vital. It will tell you a lot about the authors and their view, knowledge and competence. Also of course you may find the report is rubbish and be able to say why.

342 Trick: Certainly amongst men, giving corrective advice and compromise is easier when walking along or in a public place.

343 Try to get 'your' item near the top of the agenda then you can leave the meetingophiles to it and get on with something else. You can save hours of ennui until everyone twiggs your game.

read the summary, and you don't need to waste meeting time going over it all again. Now you're in a position to

(a) Put the paper into the context of the meeting. For example "The user has requested feature X. This report describes the programming resource implications for the three options being discussed today." (Note. Don't tell people their job.)

(b) Add-in late breaking news. eg "Since the report was written, the estimate on page 3 has been revised to "Acceptable (providing foo)".

(c) Cover any queries raised before the meeting that might indicate confusion or matters on the edge or beyond the strict terms of reference. eg "For clarification staff holidays will not be an issue".[344] Here (or in (a)) you might also want to stress the confidence of the conclusions. Some people have a nasty habit of ignoring other variables used as assumptions. For example fixing on 46 days as a shorter period than 50 when the uncertainty might be measured in weeks and then taking that as the only thing that matters.[345]

Which leads to the most important and clever part:

(d) "Any questions?" It's their meeting, let them do the talking, try to see if they have a hidden agenda. If your facts are questioned that's no problem you know their provenance. If it's your opinion that's being questioned then make sure they appreciate the difference between analysis of the problem and recommended solution.

Review

Dealing with users and colleagues requires good inter-personal skills combining psychology and simple pleasantness. You must be easy to talk to, appear interested in other people's problems and 'bring something to the party'. This last item might be entertainment[346] or a distracting natter about football to begin with, but that's only a 'way-in' to trust, picking your brains and asking your assistance.

The sure sign of a nerd is someone who thinks the number of gigawotsits you need to play the latest hacked release of Kingdom of Zog is interesting to anyone else on the planet. Get a life! If you can't talk to ordinary humans about ordinary things, or at least strange things that are real, then you're a sad waste of DNA that needs to go bareback bronco riding, ballroom dancing, bell ringing[347] or all three.

The better your managers are the more you can allow yourself to be drawn informally in order to expedite work knowing that everyone is playing for the same team. You can give an opinion based on little more that thin air knowing that your audience appreciates you're busking it a bit and might change your mind after a good night's sleep. But be aware that managers have different agendas, knowledge and expertise

---

344 Notice how you're an ambassador who may have to deal with ultra vires - non programming matters.

345 There are a lot of useless decision makers about. Some so awful you wouldn't believe.

346 It works for me. I take fooling very seriously.

347 Campanology should be ideal relaxation for programmers as it combines physical exercise, precision, team work and tricky mental logic to follow the change pattern.

which can lead to abuse of trust. *In any case* you will be relying on the written word (and possibly the occasional graph and diagram, but particularly tables) to communicate. If this causes stress then ask yourself why standard practice should be abandoned and who might suffer as a result!

> Managers have their problems too : Programmers!

Teamwork

> A committee is a parley of private interests. A working party is a group with a single agenda. A team is a group that starts from the premise that we're all on the same side working together with common interests.

Although coding is a solitary occupation, most other programming tasks involve other people. Even if you design everything yourself you can still benefit from floating ideas past people to see what they think. Often there will be a horde of various flavours of programmer in an office in some amorphous mass, typically operating on the 'ant and twig' principle: Give enough ants enough twigs and they can build a nest. Each ant scuttles around doing its bit in an amazing example of the power of cooperation. But that doesn't really qualify as teamwork, just group efforts.

Building teams

Millions have been wasted on sending groups away for the weekend for team building exercises. Walls are not built by throwing bricks together and neither are teams.[348] Teams are built by *team builders* who set out to organise a pool of useful, trustworthy and loyal participants. As a technically competent programmer with half-decent social skills you are in an ideal position to say "Hey here's a good way to share the work and put our heads together. You're good at X and you're good at Y and I can do a bit of Z at a pinch so if we pool resources we can crack the whole project with maximum efficiency"

> There's a great deal of satisfaction from creating a harmonious team capable of covering all the bases. Once you've got a bit of experience you'll be forever getting other people networking in order to discuss issues before they turn into big problems.

You don't need formal qualifications or a job description to be a team builder. Once you've got the knack you never loose it, and you are always looking for opportunities for getting talent to participate in group activities.[349] Of course you need to assess people's strengths and weaknesses and match them to your overall vision of what

---

348 If you can't skive off participating in some barmpot team-building weekend then do some *real* team building: Secretly collect a handful of subversives and make unorthodox preparations in anticipation of your own party, manufactured crises and problem solving exercises. Much more worthwhile, more fun and more adrenalin than the cheap thrill of walking over hot coals.

349 Also spotting hidden talents and coaching. As well as spotting character flaws - and in time become disillusioned.

needs doing. You should have these skills so put them to good use. Normal people enjoy working sociably but many programmers need extra encouragement because of irritating experience with ant-and-twig which can lead to disjointed work schedules. I suppose it's a matter of being intelligently positive and avoiding mindless coercion.[350]

Team structure

Everybody in a team knows the people to approach with suggestions or difficulties. One of the important characteristics of being a founder member or driver, is redirecting such queries that come to you (because you're well known as 'knowing what to do') *by introducing people so they can talk directly*. There should be at least one facilitator and you should have the skills and contacts for that.

Can everybody play in goal? No. Should everyone be chasing the ball? No. Right then, at any one time different people will be doing different jobs. In a programming colony there's the traditional way of doing this, pretty much along the lines of various sorts of programmers all doing programming and chipping in their specialisations and the racy, avant garde way (that's been around since Fred Brookes described it in 1975 - See Books¤) where coders are outnumbered by the other people in a team who support these lead programmers and make the whole product. You can think of it like airline pilots who drive and command the plane supported by other staff on the plane, or as Brooks describes it as Chief and Assistant surgeons supported by well organised specialist assistants in an operating theatre.[351] *The mythical man month* is essential reading which should be a revelation.

Leading a team

The official post of Team Leader is often just 'senior programmer'. If you're not careful you end up with grumbling responsibilities and tedious administration without the power to do anything. If you've evolved into a team leading niche you have the power that comes from respect and shared experiences without the administrative chores.

Similarly don't get conned into an administrative role especially chairing or facilitating a committee or working party. That's what low level managers are for. The consequences of eschewing committees and building teams is that you have a more efficient parallel communications network which is capable of triggering helpful actions in a very short time. "Hi Sue. Would you mind if we reversed the ...".

The grief from being a real team leader comes when a whole team develops a different view of things to the management. "We don't care. Our holidays are booked. Not our problem. The management should have thought of that before rescheduling." At this point you, personally, are seen as 'the problem' and management goes behind your back to try and divide the team up.

Why not be a manager?

You'll have seen enough managers to know that the entrance qualifications are not very

---

350 Have fun taking the mickey out of mission statements.

351 Would you trust your colleagues to operate on you? If it's any consolation, many surgeons don't trust the majority of their colleagues either.

demanding[352]. Have a look in the glossary for OM¤ and Factors¤ which are two basic tools which should remove any remaining mystique.

At age 45 brains start to slow down and the experience that comes with maturity should be the natural prompt for a move into management. If you have problem solving skills, vision and decent technical knowledge (which you should as a RP) and if you have proven team building skills then shouldn't you be in charge? A lot of very skilled people ask themselves the question then decide against joining the existing management because of the meetings, politics and pressures involved. As a result they leave to set up their own businesses and be their own boss, or retire early having made a nice wage in the meantime.

Large organisations should be worried about this loss of highly competent people. There's very little cross-over between the commandos and base-wallahs which applies at all ages - although there is a suspicion that failed programmers find comfortable management billets.

Can you avoid being a manager?

But what about the new dawn of remote collaboration? If you want to work your own hours at home then you need a good helping of team awareness and understanding what management needs from you. Essentials are:

- Writing clear reports. Also judge the scale of the work: Should that be a "Yes OK" email, a paragraph or two, a page or two, or ten pages?
- Be extremely well organised
- Keep *ahead of* deadlines
- Keep the need for people to contact you to the minimum. Try to get things right first time. Discourage last-minute kerfuffles which might interfere with your days sailing etc.
- Demonstrate your good self discipline to others.[353] Maintain your reputation for doing what you've said you'll do. (Even if it's only an apperance.)

I've bored you with management so you can spot management related issues, then experiment with management related opportunities. How do you deliver projects where the participants rarely, if ever see each other? It can be done as Linux and Firefox prove. At the time of writing there is very little experience around of a management model. *The Cathedral and the Bazaar* is an essay by Eric S. Raymond[354] available on-line as an essay or with others as a printed book of the same name.

The team you can't see

When asked "Who are the people you work with?" most programmers forget to include users. Oh dear. Programmers can be very self-centred and easily fall into the trap of believing their views must be superior to the users - almost by definition. (This is

---

352 Why are there all those Management for idiots, and Management in 15 seconds books?

353 This doesn't mean be a boring old fart, but it does mean being trustworthy.

354 One of the names in the Internet hall of fame.

particularly noticeable where geeks retreat into their technological world.)[355]

So you as a RP, interested in people, recognising that if you 'do a good job' you'll be asked back to do another, realising that users are not 'up to speed' in how to sort their problems out take the time to find out what they say they want, what they actually need, advise and liaise during development. Job done - All down the pub for a drink?

Err... No. What about the users that actually use your program. There may be thousands, all over the world, some may not even be born yet! There will be a mix of cultural backgrounds for example Windows/Linux or BASIC/Lisp/Java or Bookkeeping/Finance or Retail/Wholesale. There will be the full range of personalities, from those that won't read the manual on principle through to those that evaluate software by the number of help screens, hints, tips and cartoons. Some people will spend the necessary time working to find 80% of the features and follow the tutorials while others need it to work within three minutes without any thought or they give up. So, they give up and call the help line with unbelievably stupid questions.

When you're World President you will be able to control the User Environment and User Motivation as well as the User Interface. Until then you've got to make assumptions and cover most bases. This is an imperfect art.

User documentation basics

Users are not necessarily *stupid*, but almost by definition - they wouldn't be reading the documentation if they knew the answers - *ignorant*. This book starts at a point most people in computing might think of as far too simple.[356] There's not much harm done in that; in fact people like to see things in writing they have picked-up already.

> General purpose advice for any user documentation:
> Distinguish between the *What* it does and *How* to make it do what the user is trying to do.

Where you are documenting procedures that involve your program you have the problem of keeping the version of the user documentation in step with the development of the program. This is mighty tricky in itself and gets worse if you have, say, a user guide, a tutorial, an animated walk-through[357] and reference on the documentation side and various versions of your application on the other. The difficulty is that there is no formal linkage between your code and the usage instructions. There are some ways to

---

355 And managers retreat into their business world.

356 If you go back to the first couple of chapters you'll see how 'ridiculously tedious' and 'unnecessarily basic' they appear. The good news is that any bits of this book that appear hazy should eventually appear as 'obvious'. The bad news is that they'll be obvious to you but not to others and you'll have to persuade them to read the book for themselves.

357 Animated walk-throughs look pretty, require no effort on the part of the user but tend to cover lots of *whats* at different levels which confuses. If the purpose of the presentation is to show how easy it is to do foo, or *just* to name the elements then that's fine but trying to mix a bit of everything without focussing on one purpose will be a pretty mess.

ease the problem:

- Write the documentation first and build your code up to it. (This is very difficult when at the design stage you haven't got anything to show and the documentation will be have to be adapted.) In this way the code can be linked to the documentation:

```
// 5.6 : Creating a new user account
// ******************************************************
// Validate users credentials - [D] Assumed!! admin user
// Display blank input screen [D] Screen shot 5.6.1
// Validate input - [D] Table of allowed inputs
// Trigger follow up actions - [D] Not clear in user guide
// ******************************************************
```

  I've never done this from scratch although writing the user guide after code has made me look again at the UI. My suggestion is that you try writing a user guide draft after the Proof Of Concept stage, before the head-down coding, and use it as a specification for the UI. If you've been given a formal specification to code which will have any sort of UI then it might help to attempt the UI first as part of discovering what the program is all about.
- As well as *What* and *How* there's a *When*. When could be called 'What-if'.

```
When the red light comes on ...
If the message "foo" appears ...
When the defaults settings are not to your taste...
```

- Distinguish different levels of *How*. For example there will be main tasks, supplementary tasks, ubiquitous operations (eg standard keystrokes) and assumed skills.[358] Often the last two of these are dealt with first to get common activities out of the way.
- *What* comes in different flavours as well. Overall purpose and capabilities, main items, subsidiary features, arcane features and specifications.
- You don't need to keep the *What* physically separate from the *How* but you must distinguish clearly between them. For example you could have a table with objectives in the left column and methods in the right, or use the FAQ style, or indicate procedures by typographic and layout means.[359]
- Normally user documentation employs a undirected language such as "Keep out of direct sunlight". (This book uses a different style - talking directly to you as if you're in the room with me.) Sometimes you may want to make a situation the user might be in
  - particular to them "... *your* preferences..."
  - more immediate "... *you* must not...", "If *you* find..."

  If you're good you'll be able to get other ideas across such as "*We* can't afford a mistake at this stage".

> In my view one of the most important goals of an application is to make it easy for people to be diligent. If you're writing a clerical protocol to go with your program then, for example, how accurate should the input figures be? If you don't tell them they don't know. If they appreciate the purpose of accuracy then there's some hope they might make the appropriate effort. If you make it easier for them to do the job the right way instead of the wrong way there's some hope of getting it right.

358 You might assume that medical professionals would write clearly and in the right box even when you make it 'impossible' for there to be a mistake by having a form checked. So did I. When the procedure was computerised there were squeals of complaint when they couldn't put anything in the conclusions box without something in the observations box.

359 Such as the way Beginner has been used in this book. You might consider using the Beginner-ish style for your instructions to users. (There is no formal syntax - It'll work so long as it *looks* different and formal.)

- Here is a simple checklist you can use to evaluate the first few pages of your documentation. The first page (don't forget this could be a web page) is extremely important in encouraging users to make the effort to read more.
  - Can the user *do something, quickly*?
  - Can the user do something *easily*?
  - Does the user know how much gold is in the pot at the end of the rainbow? That is why should they bother with wading through all this writing?

  These three can often be dealt with by a walk-through of a 'here's one I did earlier'. Possibly starting with the finished article and working backwards. For written documentation of a private business system consider an encouraging foreword by the managing director which puts the effort of getting used to the new system into the context of the benefits all round.
  - Is it obvious that learning and understanding comes in stages? Many applications make the mistake of having lots of help screens and other on-screen pages but there's no structure or 'way in'. For written documentation a contents page and section numbering give a clue in this direction.[360]
- Always put what the user *wants* to know at the start[361]. What they *need* to know next with *may be useful* following. A review/glossary/index/quick reference often goes at the back.
- Obviously, use the user's language. Where a term is used in particular way (perhaps your program has caused something that was vague to be structured and so names for data items need to be clarified) then you'll have to make sure three times over this message gets across:
  1 A definition
  2 How this is different (and why) from traditional use
  3 Deprecation of traditional use

Testing

- Again and again. You'll be amazed at what people find difficult or confusing. Just because one person finds a difficulty doesn't mean going back to the drawing board. Learning *is by it's very nature* difficult.
- Try to give out documentation before letting the user-tester look at the program. This lets them evaluate the approach, style and structure separately from the 'click here to do this' bits.
- Make sure you get proper feedback.

---

360 Many programmers are good at unstructured learning but generally it's a disaster.

361 Which means you need to have researched what the users want. It might be reassurance of nothing bad or 'difficult', or promises of benefits. Don't be shy, do some selling.

Review

Communicating with humans is an essential skill. Most people are not programmers and view you as an expensive nuisance. If you're not relevant to them you won't get their cooperation.[362] Writing has to be clear and purposeful - This takes practice. Talking needs to observe the social conventions that make for fluency. - This takes practice and benefits from preparation.

Self management is something you'll need as a matter of course. The more you do it yourself the less you'll be pestered and the more you'll be respected as competent under pressure.

You may be the person best qualified to build teams. You find out who are the people you can trust and those that consistently fail to meet expectation. In the first instance your objective is to reduce arguing time and increase problem solving time. As you get more experienced you should find yourself encouraging members to take more active roles and the team will 'take-off'.

Team and business management may not be for you. Take note of the anti-bullying remarks earlier in the chapter to insulate yourself from management pressure and politics.

You will have to deal with lots of people and organisations many of which do not share your views. Many you can educate using charm, authority and simple teaching techniques, but you'll come across some hard cases where you need to walk away and leave them to stew in their own juices.

> Although computer programming can be a fascinating intellectual exercise, that's nothing when compared to the challenge posed by real people in real situations.
>
> Start with the person called you.

362 The quality, quantity and choice of communications is a huge segment of systems analysis. That's beyond the scope of this book, although as you may have gathered you need to have some idea of the real world context in which your program will live.

19. Review

I'm looking backwards over thirty years of trial and error, progress and development, investigation and achievement somehow condensed into these few pages. You're looking forwards to expanding what you've just read into another thirty years of top-level challenges.

The best way to review the content of this book is to read it again. You'll be able to whizz through the early chapters possibly thinking critically about the content and style. If there were early exercises you struggled with originally you might want to have another go to prove to yourself how much you've progressed. The later chapters have more prompts to investigate the computer science related aspects of programming which you do need to follow up so you know what sort of mental models and technology (not to mention fashions) are available for your use *if applicable*.

Of course you'll be developing your fluency with YCPL and continuing to experiment with development environments and tools. When I was working full time nine-to-five I tried to put aside Friday afternoon for private investigations - As a result productivity doubled every four months.

If you take the view that programming begins and ends with writing code then you will not make a good programmer. It's not technology that's important, it's not showing off with complex code that's important but the utility of the end result.

Chapter 14 should have given you an insight into how good code is developed. There are lots of items in that chapter that require you to

- appreciate how they fit into the whole scheme
- decide how you're going to implement them.

Chapter 14 is not a works-out-of-the-box coding technique but a pattern that you will develop according to your own circumstances. It might be a good idea to run through it checking for bits that you should be implementing, listing them and getting down to building your own workshop and workshop manual.

If you're intelligent enough to be an ordinary programmer then you should have what it takes to join the elite. Many fail by concentrating on screen-related activities instead of the real world. It will take years of serious training, re-focussing and continued application to become a great programmer - but you're already facing in the right direction, and you've got the first few steps mapped out for you - which is more than can be said for many.

I hope you've picked up some of the fun of problem solving, the enjoyment of learning new things every day, and the satisfaction that comes from doing a job better than others. Good luck.

Parting thoughts

- Design is the creative bit. Errors get lost (bad thing) in the fuzz.
  Coding is the precision bit. Errors should stand out.

- There are two ways to be a useless programmer: The first is to forget to see the real world in which your code runs. The second is to fool yourself you can do it without a method.

- There are plenty of ways to be an inefficient programmer. The most serious is to let others tell you what to do.

X-Ray vision

A few days ago I explained to a lady who was trying to bamboozle me with spiritualism that "I could see through brick walls" because that was my profession. Water off a ducks back in this case... ...and I suspect you'll get the same response.

But you can!

Metaphorically.

Just as physics shows how molecules and masses and waves work to beam a concert from California to Crewe, and economics shows us why somebody bothered to send samples from Santiago to Salford, so practising real programming shows you how people really think, what their agenda is despite what they claim - even if you ask them directly.

If you can differentiate what people say from what they think from what they do then you can become a modern day priest.

- People will interpret your good listening skills as evidence of your natural goodness and trustworthiness
- You will be able to put people's troubles into context.
- Although they know you can't work miracles [363]- nevertheless they have faith in your abilities and admiration for the unfathomable difficulties you have to deal with on their behalf.

Postscript

Isn't it wonderful to be human after hours of rigorous development, cynical nodding and creative problem solving.

The price of WCPGW...

Sorry to say, but all the "WCPCW" references were originally spelt "WCPGR". How embarrassing is that! Let that be a warning that you can spend a couple of months blissfully unaware of a stupidity that's staring you in the face.

---

363 But the occasional rabbit out of the hat won't go amiss and is very satisfying.

...is worth paying

As late one night I invented the word "Iterature" to mean:

> Literature in the scientific sense, meaning published papers, on the Internet. As in "Go and have Google for the Iterature on the subject"

Glossary

<A

*nix
Unix, Linux and variants.

@@@
Tag in code that says "must come back here later to finish-off". You can use any arbitrary string so long as you always stick to that one string. You can use it in any electronic document you're working on, and of course search for it to see what loose ends are still outstanding. @@@[T][D]

A

Algorithms
I don't want to duplicate the excellent references easily available on the Internet, for example Wikipedia, so this is a short entry. At one time algorithms were seen as essential study for programming. Nowadays the mainstream programmer should be able to get by with a general knowledge of standard algorithms and a recognition of the old adage 'there's many a slip twixt cup and lip'. See chapter @@@.

Array
Indexable list of elements. In general, exceptions apply, arrays are *dimensioned* to be a fixed size. Say 12 months of the year. There are gotchas:

- Many systems use 0 for the first index which can be strange to people used to things starting at 1. This also means that the highest valid index for an array of 12 elements is 11.
- Sometimes a programmer will dimension an array 'with room to spare'. "20 will be ample for the number of live bug reports"...and so it was until one day it wasn't ...which meant another bug report ... and so to meltdown. See constants¤.

Dynamic arrays and vectors¤ get round the issue of fixed size.

Assembler
Assembler code is a 'readable' version of the lowest level processor instructions. An assembler is a compiler which turns this text file into *machine code*.

B

BGB

Bad-Good-Best model of competence. See appendix G.

Books

You can learn a lot from the Internet but there are some books well worth a good read which will alter your whole outlook. Wikipedia has a good book review section.

The Mythical Man-Month: Essays on Software Engineering

Author: Fred Brooks. First pub: 1975 and 1995
This is a classic about the management of software projects. Amongst other useful and insightful things it contains:

- Adding manpower to a late software project makes it later. (Brook's law)
- The over-design that goes into somebody's second system (2nd system effect)
- The system architect should write the manual (draft but in detail) as the specification for the software developers.
- The first attempt will be a prototype whether you intend it or not.
- Surround a couple of really good programmers, which he estimates are 5-10 times as productive as the average) with a team that supports their work with tools, testing, admin and so on. He likens this to a surgical team.

Security Engineering

Author: Ross Anderson ISBN 0471389226
A wide view of security engineering including computer security. As well as being an excellent introduction to security techniques and ideas, it really highlights the practical difficulties and is a feast of WCPGW. Essential reading.

Programming Pearls

Author Jon Bentley ISBN 0201657880.
Key programming techniques to go into your brain. Lots of case studies which highlight the role of creativity and insight. It covers program design and coding and will show you why some programmers are so much better than others.

Brittle code

Liable to break either due to confused coding or age. A bad thing which you'll learn about when making minor alterations that cause melt-down, expose old bugs or introduce new ones. As your tools evolve over time and people who understood how the code was glued together move on so your development environment can be a contributory factor. See Legacy Code.

Bug

A bug is a fault lurking in a program. Like flu, the symptoms are unpleasant, may strike

anywhere and be easy to spot, but the cause is invisible to the naked eye and nobody knows where it came from. Although there is plenty of advice, scientists haven't yet found the cure.

Bug-free code
A myth.

C

C
An incredibly influential programming language which gave programmers the right mix of efficient use of a machine at low level with high-level language features. At one time C was ubiquitous and available on just about any platform. This made it the language of choice for a lot of people and companies. Nowadays you'd need a very good reason for learning it.

Can never happen
But you'd be surprised how often it does.

This comment is often seen in code, and as a signed-up member of the WCPGW club you'll understand why. We're not talking improbable or unlikely but *impossible*. Being a defensive programmer you may not trap impossible errors but from time to time you will comment your code to explain that a particular logic branch can't happen because you validated-out all the loopy options or possibly the data feed you've been given is specified with certain restrictions and you check those restrictions.

```
if ((month>0) and (month<13)){
  ...process normally...
}else{
  Die('invalid month');      // can never happen
}
```

This isn't paranoia[364], just the real world being its normal self.

Case(1)
UPPER CASE, lower case, Mixed Case, CamelCase. See Crash Case¤

Case(2)
Synonymous with the `switch` logic construct which performs one of a multiple choice set of options. Read YCPLs documentation for gotcha¤

CASE(3)
Computer Aided Software Engineering. If you listen to some people CASE is a utopia where requirements are typed into the a computer which then produces finished,

---

364 *They* are saying you're not sufficiently paranoid. By the way, it's not a question of 'conspiracy or cockup' but 'conspiracy and cockup'. Shields up!

tested, documented and optimised code. It is fairer to say that since the dawn of computing software engineers have recognised that computers can help them produce better software faster and cheaper, and a lot of effort goes into rationalising the design and manufacturing processes. Even if the integration of tools was perfect they would still be expensive to buy, expensive to install and maintain, require a lot of training and have to be kept alive long after the last new bit of software came off the production line. Tools are a brilliant way to improve productivity but in my opinion, as discussed in chapter @@@ we are not at the integrated production line stage yet.

As a programmer you need to be aware of what sort of facilities are available and 'waste' some time getting hands-on experience to find out for yourself (a) what the practicalities are and (b) how you might implement the ideas behind the tools in other ways.

Code

- To — To sit at a screen typing in a computer language.
- Source — What the programmer typed
- Object — What the compiler produces (almost) ready for the computer's processor (or virtual machine¤) to run.
- Executable — Immediately runnable object code
- Pseudo — Outline sketch of how the final source code would work

Champion

A powerful person who shares you vision and is able to make change happen even against opposition. Rare. Many projects will end in failure because there isn't a champion to provide leadership and face-down the moaning-minnies and fatal-compromisers. At the start of every project ask yourself:

- Who cares enough about it?
- Who understands what will make it a success?
- Who has the power to make it happen?
- If the going gets sticky who will see it through?

If there's the risk of confusion, objection, change of policy or interference and you can't answer these questions then look around for something more worthwhile.

Collection

A generic term for a data structure. Common derivatives are

- bag - duplicates allowed
- set - only one of each item
- array, vector - linear list
- dictionary - keyed list
- hash table - specialised form of dictionary
- linked list - items chained together
- tree - multi-level linked lists (or lists of lists)

Many OO languages come with a class hierarchy which allows you to call collection methods such as enumeration, store and retrieve without worrying about the exact implementation.

Command line

Text interface through which commands can be typed. A Command Line Interface requires knowledge and keyboard skills to use. CLI may not be fashionable but for many aspects of programming it is very practical. See chapter @@@

Comments

Comment-free code belongs in the bin. Follow local style or develop your own. Firstly automated tools may be used to extract certain comments to save doubly documenting your code. Secondly you and your colleagues need various levels of pointers to the workings, assumptions and tricky bits. Thirdly you can use comments to tag your code during work-in-progress either as reminders or a handy place to put associated information such as parameters to test with until you get round to that phase.

Compiler

A program that converts source code into object code. See code¤. See appendix E.

Computer Science

Computer scientists like to deal with the *How*-do-they-work of computers. Software engineers (ie programmers) follow the creed of *What*-can-we-do with computers. The former is easier to teach being a collection of techniques which students can be asked to enumerate and master while programming is more fuzzy and can't be learnt from a book.

Programmers need to know something of computer science but there are arcane areas that are far less important than hours and hours spent practising the art of writing the right program. In my opinion a good programmer will explore available technologies to the extent of knowing what they may do *then salting this concept away for future reference*. Knowing in the back of your mind that there might be a clever way to tackle the knotty problem in front of you, and knowing where to look, are the key skills.

Console

A console is derived from Teletypes and later VDU terminals which controlled a program (including the operating system) by sending and receiving characters 'down a wire'. (Modern systems paint a whole screen and have built-in keyboard and mouse handling.)

Originally most programs worked by receiving characters typed at the keyboard via a port¤ typically given a system name such as TTY: and displaying progress, or results by sending characters back to TTY: (TTY being short for 'Teletype'.) (CON: SCR: KBD: are some of the variations on this port naming theme.) Nowadays if you want to receive from the 'console keyboard' you'd probably read `StdIn` and conversely write to `StdOut`.

There are a lot of programs that still use a console for I/O¤ because
(a) You can operate them from the command line
(b) StdIn and StdOut are just about universal and simple
(c) You can operate them remotely. eg by Telnet¤
(d) You can make really tiny programs uncluttered by display gubbins
(e) StdIn and StdOut can be used to talk to the StdOuts and StdIns of other programs

Conspicuous relaxation
One of the privileges of being a Real Programmer is self-organised total rest periods. Others don't have the mental strength to stop completely, they bumble along making distracting interruptions and trying to look busy while gazing out of the window. As a mental athlete you can knock the spots off them any time you like, that's why you're well paid... ...except it can't be done for hours and days on end. See Nervous Exhaustion¤

Crash case
To force all characters in a string to either upper or lower case.

CSV
Comma Separated Variable text file format. This is the original classic data transfer format with one record per line and fields separated by commas. It is simple and practical for basic data types that can be represented as text. If you need to get data out of or into a spreadsheet you can probably use this without any worries.

```
North,2,Spades,,
South,4,Spades,,
West,,,Double
```

- Don't use for binary data!
- WCPGW? Put strings in double quotes if there's any chance of a comma appearing within the string. WCPGW? Even within all strings being quoted you forget to check for comma-within-quotes gotcha.

D

Dates
It may surprise you that many programming languages do not automatically support the following dates:

"1066" or "2006"
"May 1999"
"1st June 1676"
"Unknown"

And many others.

Dates depend on two things: The facilities provided to the language by the operating system and the features provided by the language itself. This raises platform-specific issues¤. See appendix D.

Development environment
Your workbench and office in both real and virtual senses. You won't be a good programmer unless this is well organised and suited to the work you're doing. If you don't have the right tools or are being interrupted in the middle of head-down coding sessions then productivity and accuracy will drop off and you won't be happy.

On the other hand with the right tools, the right physical environment[365] and good team work you can work at five times the average pace.

E

Enumeration
Working through the items in a collection¤. Typically starting at the first item and repeatedly fetching the next until the end

Error
A fault of any sort. It is the programmer's job to prevent errors happening in the first place and to deal with any consequences of lurking faults.

Exception
The recognition of an exceptional situation that arises while a program is running. This could be a division by zero or exceeding the time allowed waiting for a response. The good programmer will plan for such events and have worked out what action to take as a result.

- Exceptions propagate up the calling stack and may be caught at any stage...
- ...unless there is no catching when the program crashes or does some other rude thing, possibly causing invisible damage that lurks to affect something else.
- The programmer can decide what action to take and where it's most appropriate.
- A frequent task is to ensure a clean-up always happens.

F

Factors
Back of the envelope project pre-planning tool. See APPX@@@

Fencepost error

Filename
Name used to identify a file. Source of Gotchas¤.

- On a Windows system `MyFile.Txt` and `myfile.txt` refer to the same thing, but on a *nix¤ system case matters so they are different entities. The Gotcha is that your program might work fine while you develop on your windows machine as you refer to "myfile.txt" but fail to find it when ported¤ to another system. A classic case is a

365 Do something about it if you're not happy.

web page with the code `<img src="mypicture.jpg">` with MyPicture.jpg being uploaded.

- *nix¤ filenames beginning with a period do not appear in normal directory listings.
- Some people are in the dreadful habit of putting spaces in the middle of file names. Bad bad bad![366] Space is often used as a delimiter to split arguments such as on a command line. If you give `my naughty file.txt` to a command line program it will probably think you meant just the file "my" or possibly "my" and "Naughty" and "file.txt". Tip: Work out a filenaming convention that doesn't involve spaces (or minuses or extra periods) How about MySpaceFreeFile.txt. *Then stick to it*.
- If you are generating file names for log files arrange them so that the year is first then the month then the day. This makes sorting them a lot easier - providing you don't forget padding with zeroes as required.
- Watch out for alias extensions such as `foo.htm` and `foo.html`
- Slash(`/`) is used on *nix¤ systems as a path separator while Windows used Backslash (`\`). Especially over the phone, this wart can be invisible.
- Personally I abhor filenames without extensions. What program should I be using to look at "`design`"? - Or is it a program in its own right?

Formatting output

In a GUI or if you are outputting HTML then your formatting methods for layout and general presentation will be quite complex. However for plain text as well as these you will probably want to format numbers so that for example they have exactly two decimal places. See printf¤.

For values of...

Hackish way of indicating that some given number, statistic or even boolean is not to be taken at face value. It can be used for "insert any of your favourite values here" or as a polite way of saying "bollocks" when someone gives you a 'fact'. "We always deliver on time" ... "For values of 'always' ".

A witticism to bore people with: Pi equals 3 for small values of pi and large values of 3.

Framework

Fudge

Almost a programming language which shouldn't need much conversion to YCPL. (I use it here because I don't know what YCPL is. Normally use real code or pseudo code.¤)

---

366 Ta very much Microsoft for calling the root for programs "program files" - not.

G

Garbage collection

Gotcha
A trap. For example mixing = and ==. Although gotchas are generally documented they can cause baffling errors - some of which might stay dormant. A typical scenario os where you 'upgrade' to a later version of a programming language, and the things you could do quite happily before are now falling on their faces. *You* haven't changed anything... ...probably - but can you be sure since the last time you tested it. The other common scenario is where you switch from one language or dialect to another. Perhaps in the on you've been using you can guarantee that uninitialised variables will always be zero, false or null but now you should do the initilisation explicitly. See Scope¤ and Filename¤ for examples.

Grep
Hacker-speak for search. The name of a widely used utility program for searching (and replacing).

H

Hack(1)
Quickly whip up a solution

Hack(2)
A practical rather than pretty patch to a system.

Hack(3)
(As in 'hack into') Find a way round security or modify maliciously.

Hacker(1)
An elite programmer that could demonstrate superior technical knowledge and extreme ingenuity to get a quart out of a pint pot, without spilling a drop, and only paying for the pint. Always rare, now almost extinct.

Hacker(2)
Nowadays "Hacker" usually refers to a programmer-gone-wrong.

Hexadecimal
See example in Type¤

Hex Editor
An editor that lets you look at (and if you're clever, edit) raw bytes in a file. Normally the screen will be in three columns: Address offset from the top of the file, 16 bytes shown in two digit hexadecimals, and an ACSII¤ representation.

Hot tap - Cold tap
Something you know but are not conscious of. Taps (in the UK at least) are always arranged

H C

Hot on the left, Cold on the right. Not many people ever think about it. That's the whole point. Humans quickly learn to use patterns to indicate shortcuts. The more usual label for this is Conventions.[367]

- code conventions
- private conventions
- user interfaces
- user guides and other communications

I

IDE - Integrated development environment
A programmers workbench which combines, one hopes seamlessly, a number of tools. Strangely enough this may be *part of* your development environment.¤ IDEs are often purely *coding* environments rather than supporting the full scope of development tools.

Idempotent
Only ever happening once. For example if a user tries to access a function for which they need permission a check will be made to see if they are logged-in. If not then the login process is kicked off. But once they're logged-in they don't need to do that diversion again.

Inter-personal skills
Good interpersonal skills are a requirement for good programming. You can learn them from books and are worth practising if you're shy, don't know how to talk to people, and people don't know how to talk to you. See chapter 18.

- Programmers can be a bit blunt as babble isn't their forte. If you think about it all they're really interested in is the shortest line from A to B and may be streets ahead when considering some matter under discussion. Waiting while other people beat about the bush, go off at tangents and get hopelessly lost is just one of those things you have to get used to. Look upon coaxing them back onto track as an interesting

---

367 But the word is so boring and does nothing to convey the hidden power of something that should be one of the sharpest tools in your kit.

intellectual challenge.

Iterature
Formal documentation found on the Internet. As in "Go and look at the iterature".[368]

J

Jailhouse
There are legal issues with programming. Don't go hacking other people's systems is an obvious one. The details of licences for the use and re-distribution of software are best left to others. You may have privileged access to customer's data which raises issues of commercial confidentiality and personal privacy.[369]

The scariest moment I've had was when I became certain that a colleague was involved in hacking. Eventually proof was found, but for 48 hours it was just my strong hunch that triggered all sorts of hairy security activity. OK, I could give good reasons for the alert but it would have been distinctly embarrassing if the case was not proven. The moral of the story is that you need to be very careful and very sure and very delicate in how you 'alert the authorities'. (That's ODA¤ in practice.)

K

K
Prefix for units of 1024. Also 1000 or 1000-ish. (Using k to replace a decimal point as in 6k2 for 6,200 is standard in electronics but a bit of an affectation in computing.)

- M is K K's (For values of K)
- G is K M's (For values of K)
- T is K G's (For values of K)
- Stick to upper case Ks, Ms, Gs and Ts in computing. ("m" indicates 1000th)
- 10 bits are required to express 1024 possibilities.

---

368 This word was created during a long late-night writing session for this book.

369 When finishing a job where you've been given access to private areas, make a point of 'handing back the keys'. This might mean telling the system administrator that it would be best for everyone if your user account was disabled. Or you might want formal written permission to retain client's data for specified purposes and under certain guaranteed conditions.

L

Legacy code

Old code that can't just be thrown away because it is still in use, even of only occasionally. There is an interesting trade-off between patching from time to time and completely re-writing in your personally preferred, ahem - a more modern and efficient language. It is a real bore to look after but many times there will be one application for which old hardware and operating systems are kept going for because nobody is quite comfortable with the upheaval that could ensue. Another reason is perhaps it was designed as a robust system by true professionals and the currently available skills don't inspire the same confidence.

M

Management anger therapy

Some stupid managers only take notice when you get cross with them. Don't play this card unless you have a handful of trumps. Give them an early opportunity to back down before you lay into them. Public MAT has the most therapeutic value. Best done on a Friday afternoon. Make sure the ball gets left in the manager's court. "I'll be in my office if you need me!" Never apologise.

N

Nervous exhaustion

This is a real danger. Real programmers need to be mentally fit and need to know how to pace themselves and work efficiently. Rest periods are absolutely necessary. Performance will vary from day to day. Having read chapter 18 you'll be taking relaxation seriously, but watch out for signs of stress in colleagues.

Newline

For historical reasons there is a confusion between the control characters CR and LF. For example, Line feed physically moved the paper in a Teletype down one line but didn't move the print head to the left, for which Carriage Return was needed. LF then CR would have the same effect (but more slowly on a Teletype¤). Then some engineers thought it would be a good wheeze to just use LF at the end of each line and hack that into CR+LF at the terminal end. Which is fine unless you mix feeds with CR+LF and LF alone.

- "Newline" means either LF or CR+LF
- "End of line" is a mythical character[370]
- *nix environments tend to be LF-only
- Windows environments tend to be CR+LF

Research ASCII control codes for more.

Notebook

A programmer's badge of office, filing system and lifesaver.[371]

Nothing's changed!

As in "I haven't changed any code at all" to which the reply is "So it fails just like it did before then!" Users are just as creative in their use of this phrase as they change their setup, don't tell you they're trying to print over the network to a different printer, and different operating system. See also What's changed?

O

ODA

Observation - Decision - Action. These need to be separated for transparent and auditable decision making. See appendix G.

OM - Objects/Methods

Qualititive project management tool. See appendix H.

P

Passing arguments by reference

There are two ways arguments can be passed to a function or method.

- By value : A *copy* of the argument is given to the function
- By reference : A *pointer* to the actual object is passed to the function

Why should you care? Because if you're working on a copy any changes you make will not be reflected in the original. For example if you were sorting an array by passing it to

---

370 Although the Ctrl+Z character was often used at the end of a file.

371 Until reviewing this entry it didn't occur to me that I should specify this as a paper-based device not a battery powered one. The permanence of paper and the ease of sketching make it the medium of choice. (Cheaper and less nickable too.)

a function you want to make sure it was passed by reference. Most of the time arguments will be passed by reference by default but there might be cases where you need to pass by value or some features 'break the rule'. Check documentation.

Parity bit

In the evolution of pulsed telegraphic data transmission and punched paper tape it was found that sometimes noise on the line or dirty contacts would cause incorrect characters to be received. So an extra bit was added to each character which might be set to 1 when the count of the 1s is the other bits was even and 0 when odd. This could be checked at the receiving end to see if any bits had been corrupted during transmission. This extra bit is known as the parity bit.

When 7 bits were used for character data the 8th was used for parity. As technology progressed and equipment became more reliable and other methods were found for detecting and *correcting* (parity only detects) bit errors so it was found more useful to hijack the parity bit for additional characters, symbols and line drawing shapes; more than doubling the size of the displayable character set.

- Look up *Hamming codes* for cleverness in computer science.

Pipe

A stream connecting two processes. This can be programmed so that process (or program) A feeds bytes into the pipe and process(or program) B catches them. This can happen asynchronously. See chapter @@@. Programs that use StdIn and StdOut ports can be joined together on an operating system's command line so that the StdOut of one gets fed into the StdIn of the next. You might see something like the following

```
>dir | find "EXE" | sort
```

where a directory listing is piped to the find program which only passes lines containing EXE to the sort program which sends its results to the screen.

Platform

Term used to describe the foundation environment in which a program runs. This may refer to hardware (particularly processor), operating system or both. For example the "*nix platform".

Platform-specific issues

Wouldn't it be nice to write a program that everyone in the world could run? Unfortunately different computers work in different ways. The first level of incompatibility is hardware and operating system.

The second is the amount of resources available. For example your shoot-em-up game might need to come in two versions, one for those people with high-power graphics cards and another for those with lesser hardware.

One of the major problems is remembering that other users will have different display resolutions. You have to test a range of resolutions and design a working compromise. I have to admit to some gaffes on this front.

Pointer

An location in memory where an object or data item can be found. A horrible source of

bugs in those languages that allow you to use them.

Port(1)
To port something is to get it to work on another platform or rewrite in a different language or dialect.

Port(2)
A named I/O¤ connection. For example TTY - to a VDU console, LPT: to a line printer (or any printer). Port names are operating system dependent. They also require the appropriate hardware or emulator¤. From a programming point of view ports are always used with serial byte/character data such as can be represented by a Stream¤

Port(3)
A sort of sub-address of an IP address. It works like telephone extensions do. For example web servers tend to 'listen' on port 8080 or 80 and so that's where your browser tries to contact. In real-life terms that's like all companies having their sales department on extension 456. Ports in this sense operate in the port(2) sense as serial byte/character channels.

printf()
In C¤ there is a print formatting function called printf on which most number formatting has been based. You'd use this to strictly control how many digits numbers have and how much space strings take up. You do this in printf (and similar functions in other languages are almost identical) by giving the function one 'mask' in the form of a string which sets out the formatting and then a bunch of variables which are formatted according to the specifications in the mask. For example:

> `printf("%d %3s %-2.2d",day,monthName,twodigitYear)` says Any number of digits for an integer. A space. Three characters (always/only) for name of month (truncate if necessary). Another space. An integer always with two digits, adding a leading zero if required.

Practically all reports need to show a fixed number of decimals and keep strings aligned and this is what you can use to tidy your display accordingly.

Pseudo code
Sketch code concentrating on what happens leaving out coding details.

Quick and dirty code
A tempting trap. There are often situations where you want to quickly cobble something together. This comes with certain risks - in particular: making a mess of existing code, doing it wrong and getting incorrect results, and leaving this lash-up in the production code. It is definitely a question of more-haste-less-speed.

This isn't to say you shouldn't get stuck into doing small jobs quickly... ... but before you start to code do three things:

- Make sure you have a back-out plan if hacking existing code.
- Use some framework of comments such as pseudo code. This will help focus your mind on the objectives, traps and method overview.
- Put in some @@@s to remind you to do standard housekeeping procedures afterwards.

R

Real time

A program that operates in real time must deal with inputs as they happen and respond within a certain time. There are all sorts of issues which are made more pertinent by the fact that real time systems are often hooked up to expensive and safety critical equipment. Keep all mediocre programmers away and only work with experienced managers.

Reassuring noises

Hand-waving, sales brochures, suits¤ making strategic commitments, management taking you into their confidence, and promises of payment. Anything with "on" as in -time, -target. Should ring alarm bells.

One of the biggest culprits are programmers who are congenitally optimistic in their estimates of difficulty, need for resources and performance.

> Unless you have a really good reason to think otherwise you should at least double your estimate of the time it will take to do a project.
>
> 90-90 rule: "90% of the project takes 90% of the time - The rest takes another 90%"

Record

A collection of data items. Often the items are called *fields*. Although strictly speaking incorrect, a row of database table or a row of a query result might be referred to as a record.

Regular expression

Hieroglyphic spells used in pattern matching. An example is given in chapter @@@. You might use a regular expression to check that some input *looks like* a properly formatted email address, or split lines from a log file into component parts.

- You'll need to read the documentation for YCPL very carefully.
- If at all possible look at other people's efforts rather than trying to invent the wheel from scratch.
- Test, test and test.

Reference

A reference is a Pointer¤. See passing arguments by reference¤.

Root(1)

The 'highest' level of the filing system. Normally indicated as a single stroke/slash depending on operating system. Sometimes this will be "the highest level of the bit of the filesystem that you can see". It is generally good practice to keep out of the root and work in sub-directories. Try not to let installers install directories directly off the root. See appendix F.

Root(2)

Term in *nix environments for the user that has master administrative permissions. So it is bad news to let just anyone or *anything* have this level of permission because then they or *it* can do whatever they like with the system. This what hackers are looking to do.

Programmers should try to ensure that their program doesn't need to have administrator (root) privileges. Often a developer will have lots of access that an ordinary user won't, or shouldn't have. This can be a source of grief when the customer can't get the program to work on their site while it works absolutely fine on yours. Moral: Test with client access permissions before release.

Run-time error

See Exception¤

S

Scope

The range of code in which a given variable name retains its identity.

- Typically variables defined within functions are local to that function and have no connection with any variable with the same name outside.
- Often for-loop variables are not valid outside the loop...
  ...but (Gotcha¤) may still be accessible in an undefined state. eg

```
var j : int;
for(int i=0; i<10;i++){j=i;}  // last time i=j=9
if(j==i){  //  Gotcha! i may be undefined outside loop
```

  The equality may be allowed by some languages, might always be true in testing but fail sometime in the future once in a while, or always on another system.

Scribblything

A writing implement.
Your whole business is getting others doing things in new ways. Don't be afraid to introduce new terms for ideas and float them past people while watching their reaction. It's an old trick to see who are (a) listening (b) bothered (c) don't like anything new (d)

are really (sometimes pathetically) keen[372].

Segway

To slide imperceptibly from one thing to another.

- Dangerous if you're trying to focus hard on a difficult thing.
- Dangerous if group attention is being diverted from key matters.
- A useful method of bringing others onto 'your ground' without frightening them.

Mental discipline note:

> If you're in design mode then you're quite likely to be skipping between ideas all the time. That's good But when coding you need to be disciplined and restrict the items you're juggling in your mind to the absolute minimum number. Leave a marker to come back to later.

Serialisation

Writing (and reading) a complete object to a stream¤, often a file. The magic is that objects 'know how to save themselves' and can be reconstructed from a byte stream. (The diary exercise in chapter 9 illustrates this.)

Sorting

There is a science to choosing an appropriate sorting algorithm¤ and an opportunity for many mistakes in coding.

- Use ready made facilities if possible
- If rolling your own, research the methods and test the code thoroughly
- If sorting more than a few dozen, items pay attention to the performance issues

Stream

A stream is an abstraction which represents a port¤, buffer¤, file, or pipe¤ as a channel through which bytes/characters can be sent or received. See Chapter @@@.

Strong typing

Some languages insist you define the type of every variable and function argument and check that you're not mixing apples and oranges. As well as stopping you from making silly mistakes the compiler can produce more efficient code.

T

Teletype

An antique terminal with simple mechanical keyboard for input and a roll of paper bashed by a noisy little drum that popped up and down as it printed one character at a time across the page. Often the characters were mono-case and always mono-spaced.

---

372 Actually the people who lap up your jargon and simple to grasp key phrases may be completely lacking in judgement and even be an embarrassment. Make a note for when you've got some rubbish you want to sell.

You'd be lucky to get 20 characters per second output. These were *brilliant* compared to other ways of programming simply because they were hooked-up on-line via a standard serial interface so you could write a 10 line program and get the results in five minutes instead of a day waiting for your job to get punched, chucked out by the compiler, amended then run (the first time might work but who knows) when the denizens of the computer room see fit to give it a whirl.

- Teletype terminals always had a bell. There's an ASCII character to ring it 0x07.
- ASCII 0x08, Control-H, written `^H`, is backspace. This explains the occasional message which jokingly lets you see thoughts behind the words. For example: `It will never^H^H^H^H^H be ready by Friday`.

Time left

A handy progress indicator if you know how much of a task has been done is

```
timeLeft = ((amountTotal * timeSoFar) / amountSoFar) - timeSoFar
```

WCPGW?

- amountSoFar or timeSoFar are zero.
- progress message is not properly cleared when completed.

Type casting

Converting one type into another. This might happen silently and automatically (either by design or nature of the beast) or you may need to explicitly convert from one type to another. For example some languages are not keen on you trying to divide one integer by another in which case you may have to convert the numerator and denominator into floating point numbers first. Silent casting, which often happens when operators are trying to do the best they can with arguments, is fertile camouflaged bug country.

Sometimes a similar thing is done with objects. For example you might have an array object which allows all sorts of object to be added. When you come to retrieve these objects you may need to tell the code what class of object you're retrieving.

Adding method definition : `Add(Object:anObject);`

Fetching definition : `GetNext();` returns Object

Usage :

```
someFancyObject = new FancyObject();
myArray.Add(someFancyObject);  // FO is sub-class of Object
. . .
sfo = (FancyObject)myArray.GetNext();  // cast to FO
```

Types

Most languages come with a built-in selection of ways of storing numbers and character strings. These trade-off accuracy, size range, speed of processing and number of bytes used for storage.

- Integers are characterised by the number of bytes used and whether the number is signed. Sometimes unsigned integers are called *words*. It is very easy to mix *signed* and *unsigned* by mistake. If you try to store an excessively large number in an integer you will get an *overflow* exception. Bitwise logical operations should be done on unsigned integers.
- Reals or Floating points are typically available in two flavours *single* and *double* precision. They take longer to process than integers and although accurate enough for most everyday purposes can occasionally exhibit pathological tendencies if misused.

- Booleans are sometimes a separate type and sometimes an integer 'dressed up'. For the purpose of interfacing with low level APIs some languages will define 8,16 and 32 bit boolean types. See typecasting¤.
- Special values such as 'Not a number', Nul (or Null), may be included within a type by a testable value or may be types in their own right.
- *Characters* are single bytes used for alphabetic purposes.
- *Null terminated strings* are one way of storing a sequence of characters where the first byte with zero flags the end of the string. Here is "Hello World":
  ```
  0x48 0x65 0x6c 0x6c 0x6f 0x20 0x57 0x6f 0x72 0x6c 0x00
  ```
  This form is more often used with system programming and libraries written in C. When calling a system function requiring a string you'll probably need to use this style.
- Fixed length strings simply allocate a number of bytes. They are used a lot when defining records. Some funny things can happen *silently* if you try to save a string that's too long to a fixed length string. One possibility is that the string is silently truncated. Another is that whatever occupies the next bit of memory will get overwritten - once again silently. This is one of the most common causes of security flaws called a *buffer overflow*.
- *Variable length strings* are fine for working within a program, fine for reading and writing to text files and streams, but no good for passing to third party libraries or fixed length records that others should be able to read. They work by having their length specified at the start followed by the appropriate number of characters. There are two things to remember:
  - There will be some maximum length depending on language
  - When sizing storage space you have to leave room for the bytes used to tell the length.

More exotic types

User-defined types shade off into objects. A crude distinction is that types are just ways to represent data while objects have methods and the other OO goodness. One use of user defined types is to package fixed length data fields into a data *structure* or *record*. This might be used for accessing random access files or possibly to allow a lot of data to be returned (or rather a pointer to it) by a function.

Record gotcha : Often languages will automatically pad-out record structures so that each new field starts on a 'word boundary' - That is if the last field was specified with an odd number of bytes an extra one will be added silently. You may be receiving data that gets garbled because of this. If the documentation is vague, use a hex editor to view the data directly.

- *Date*, *time*, *date-time*, *timestamp* are variations that may co-exist in some languages and may need converting. For example your database will support dates and times of various types but you may need to present them in a particular format (full of gotchas) in order to communicate with it from your program.
- *Currency* is often found as a (non-standard, not transferable) type with the advantages of a fixed number of decimal places and rounding rules to suit monetary calculations.
- *Exceptions* may be offered as a separate type.
- *Events* may be offered as a distinct type.

- *Functions*, *methods* and *event handlers* may be offered as a distinct type.

U

Unicode

A standard for encoding text that breaks the 255 different character limit imposed by one-byte-per-character. If your program will be used beyond the English speaking world then you should be looking into this and making policy decisions as a result.

Upgrade

A traumatic episode that occurs by

- enthusiasm getting the better of experience,
- the need to exchange known fatal bugs by surprise painful ones,
- stealth and subterfuge.

If it's not broken don't fix it! This is a deadly serious subject. Upgrades cause all manner of grief - often expressed in peculiar ways - often completely clobbering vital operations.

Warning

- Never be the first to upgrade - let others find the snags
- Research the problems other have had before you follow
- Don't be fooled by the "everybody must be 'standard' cry". That 'standard' will soon be last years model and you're on an escalator. Take an objective look.[373]
- Don't be conned by "It's free" - So is a fight with an angry bear
- But all the new features! - So? - Is being able to spell check a spreadsheet in Swahili worth it?

A rational upgrade strategy is

- to stay aware of the state of play and where it might be leading
- work towards upgrading in your own time
- if possible, have a guineapig
- only upgrade when 'necessary' or 'the major changes' are worth it.

Precautions are

- Always - Always - Yes, Always! have a roll-back strategy and keep it in place for as long as possible.
- Test all the little used things as well as the obvious.

---

373 Or it might be necessary to take a devious look: List all the tools you have on your development machine, any one of which failing to work will incapacitate your work, and say in writing to your boss that you don't think it is a good idea unless the upgrade supplier will indemnify against all possible scenarios.

V

Vector
A list or dynamic array.[374] The valuable feature of being able to expand if the need arises comes with some slight overheads of computer time and needing to focus your mind on precisely what's in the list and how to access items and do housekeeping.

W

Watchdog
Also Watchdog Timer. An alarm that will trip and interrupt sometime in the future used to catch things that should-have-but-didn't. The object is to prevent infinite loops or hanging when resources are not available.

```
Set watchdog timer for 1 minute ahead
try
  loop until connected to database server via network
  WCPGW - Server n/a.  Network is a notwork  ... HANG!
except
  if watchdog-alarm-exception terminate with message
```

WCPGW?
What Could Possibly Go Wrong? Just as clocks go 'tick' and cows go 'moo' so programmers go 'wcpgw?'.

What's changed?
A really good question to ask. If it used to work but doesn't now then *something* has changed. Be persistent! See Nothing's changed!¤

Whitespace
One or more non-printing characters.

WOMBAT
Waste Of Money Brains And Time.

X

---

374 You might still see Vector used for a plain array. Originally it was the indexed listness which made a vector. The emphasis on dynamic sizing is a modern usage.

x

Used to indicate the following is a hexadecimal number as in `xFF` or `0xFF`.

Y

YCPL

Your chosen programming language.

Z

Zzzz

Meeting on a Friday afternoon. Nobody wants to be there and everyone will have forgotten the key action points by Monday.

>Z

(See previous entry. Nothing of note happens after Zzzz.)

A. Using Javascript

Javascript is available for free, built-in to every web browser. As a general purpose language it is very limited and is difficult to debug but just sufficient to run the exercises in chapter 5. It is important for programming web pages, and being able to use it for that purpose might be a very useful skill.

The purpose of this appendix is to show you how to program the exercises in chapter 5. You probably wouldn't use Javascript in everyday web pages in the style described here and this is not meant to be an 'all you need to know tutorial'.

Development process

1. Create a document using a text editor
2. Save as foo.htm
3. Point your web browser at foo.htm
4. Debug by repeatedly editing foo.htm, saving and clicking refresh on browser.

How to write a HTML document is covered in chapter 2. What that didn't cover was how to include a block of Javascript. Here is the first Hello World code:

```
<html>
<head>
  <script LANGUAGE="JavaScript">
    document.write("Hello World")
  </script>
</head>
<body>
<hr>
</body>
</html>
```

- Type this into a new document using a *text editor*.
- Be very careful with the quote marks
- Save in a suitable place with the name `hw1.htm`
- 'Point' your browser at hw1.htm - either by double clicking on the file name if using a file manager[375] or typing `file:///whereveryousavedit/hw1.htm` into the browser's address box.

You should now see Hello World displayed and a line displayed.

- If you can't see the line do View-Source to see if you're looking at the file
- If you can see the line but nothing else then either
  - You have Javascript blocked on your browser - Turn it on
  - Or you've made a typing mistake

Note : Each time you change something you have to refresh the page. Ctrl+R does this on many browsers.

---

375 I did tell you to make sure file extensions were always displayed when using your file manager.

The place of Javascript

As you can see from the code, we are putting the Javascript into the head block of the document. This code automatically gets executed before the document displays. Normally the bulk of a HTML document is in the body, but as all we want is a little framework for trying a few simple programs we can hijack the document and leave the body block blank.

A note about Javascript syntax

You will need to look at some reference materials which will also clue you up on the other things that Javascript can do, ways it is used and how it interacts with the structure of a web page.

What will seem strange is the use of periods in the middle of 'names-cum-functions'. This is something that we haven't dealt with by the time of chapter 5. Briefly (and not 100% accurately) `foo.bar(buz)` is a function bar, with argument buz and that function is (a) defined as a method that applies to whatever the type of foo is and (b) when called applies to the variable foo. `document.write("Hello")` writes Hello to the thing called document. `document` has been pre-defined. The function `.write()` is termed a *method*. `document` is an *object* (as in object oriented programming).

- Javascript is *not* Java
- `;` is *not* used at the end of statement
- All the code goes in a `script` block
- The results get interpreted as HTML.

hw2

Assuming you have got hw1.htm to show properly, we can look at hw2.htm.

```
<html>
<head>
  <script LANGUAGE="JavaScript">
    tot = 1.1 + 2.2 + 3.3
    document.write("Hello "+tot+" World")
  </script>
</head>
<body>
</body>
</html>
```

Javascript is not very type-sensitive. You can get away with murder... ...Or be mugged if you're careless.

hw3

```
<html>
<head>
  <script LANGUAGE="JavaScript">
    document.write("Hello World<br>The time is ")
    today = new Date()
    document.write(today.getHours() + ":" + today.getMinutes())
  </script>
</head>
<body>
</body>
</html>
```

`today = new Date()` means create a new object of type Date and call it "today". `.getHours()` is a method (ie. function) which returns the hours extracted from a Date object. In this case the particular object is `today`.

Framework for running examples

The hw series proved the system works and gave you a bit of experience with the edit-save-refresh cycle. Now here is exercise 5.1 with a top and tail sections.

```
<html>
<head>
  <script LANGUAGE="JavaScript">
    // ******************
    // ***** Heading *****
    // ******************
    document.write("<h2>Exercise 5.1 - Average</h2>")
    document.write("<hr>")



    // ----- fill array with any old data -----
    COUNT = 8
    a = new Array(COUNT)
    for (i=0;i<COUNT;i++){   // js arrays start at 0
      a[i] = i + 1           // when i=0 1st element value is 1
    }
    sum = 0

    // ----- step through array adding up -----
    for (i=0;i<COUNT;i++){
      sum = sum + a[i]
    }

    // ----- results -----
    average = sum / COUNT
    document.write("<br>Average of " + COUNT + " items is " + average);


    // ****************************************
    // ***** flag end of program reached OK *****
    // ****************************************
    today = new Date()
    document.write("<p><font color=green><small><i>")
    document.write("["+today.toLocaleString()+"]")
    document.write("</i></small></font>")
  </script>


</head>
<body>
<!-- all the work goes on in the javascript in the HEAD -->
</body>
</html>
```

The top identifies the program and tells you it has started. The tail tells you it has completed and displays a timestamp so you can be sure that you're looking at the most recent refresh. Obviously the middle bit contains the operative code which you can

replace as required for the other exercises.

The reason for this top and tail stuff is that if Javascript encounters an error it just stops! No handy messages or even a line number to give you a clue. If your code doesn't complete then copy or move the line (which draws a horizontal line)

```
document.write("<hr>")
```

down the page a bit, see if it gets displayed: If yes move it down more, if not move it up. this should help you locate the line the error appears to occur at.

If you were wondering...

- Can Javascript read and write files? No.
- Can Javascript change a page without reloading? A little bit which can get very complex to implement. Have a look at examples.
- How are things made to happen when for example a button is clicked? Buttons (and other elements) have built-in *event handlers* that react to events such as mouse clicking which normally do nothing but can be assigned to Javascript functions. In the following HTML example, a button reacts to being clicked by calling the a function called DoReplace().

```
<input type=button value="Replace now" OnClick='DoReplace()';>
```

B. Binary logic

Binary is about manipulating 0s and 1s. We do this is two ways

- Logically : eg 011 AND 101 gives 001
- Arithmetically : eg 011 PLUS 101 gives 1000 (3 + 5 = 8)

You won't have any difficulty getting a working knowledge of binary but there are practical traps we'll investigate.[376]

Boolean

A *boolean* variable can only be either True or False. By definition True is NOT false (and False is NOT True). Sometimes languages and data formats will treat 0 as False and 1 as True but don't rely on it. If you are only dealing with Trues and Falses you can use the following table to tell the outcome of combining two variables in different ways.

Boolean input arguments		Result when combined using operator		
		AND	OR	XOR (Exclusive OR)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

Brackets

By far and away the most important thing to learn with all binary operations is to put brackets round every operation to be absolutely sure you control the sequence of operations. Never forget this. When you do - remember I told you!

Base 2 Binary	Base 10 Decimal	Base 16 Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

Arithmetic

In case you don't know binary arithmetic operates using base 2 instead of our normal base 10. The only reason we get involved with

376 There was a time when schools became very keen to teach everyone the binary system 'because we'd need that for when we all grew up into the computer age'. With slide-rules and log tables going out of fashion binary arithmetic seemed the very thing to go in it's place. It's a bit like saying we'll all be driving cars so lets teach primary school kids thermodynamics.

binary arithmetic in computers is that deep, deep inside, computers store and manipulate their numbers using binary and sometimes we can take advantage of or be tripped up by this fact.

Because generally nowadays computers work with 8 bits at a time the largest unsigned number is `11111111` or `255` or ff or `FF` or `0xff`
(Don't worry about case in hexadecimal. If it isn't obvious that you're working in hex put a zero-x in front.)

If you were programming a volume control you might calibrate it in 255 steps from off(0) to max(ff). You'll often see this sort of thing in lower level programming. If you were getting the answers to a million questionnaires you might limit yourself to 14 options a "didn't answer" and a "not relevant" to save space and make each record compact by cramming two answers into a single byte. This is the tail wagging the dog! Sometimes it is justified to set your real-world limits on these artificial boundaries. Would there be any point in sticking to 8 possibilities, using three bits? The answer to this is 'it depends'. If you have an absolutely huge amount of data then tight packing is a very good thing, otherwise just be sensible.

What if you have a remote control for your model boat where you want the rudder to go from full left, through zero (ahead) to full right? You need -127 to + 127. The highest, leftmost or most significant bit can be used to flag the sign. There are other ways of representing signed numbers which most programmers will never ever need to worry about.

Sums

1 + 1 = 10 (1 + 1 = 2 decimal)
10 + 10 = 100 (2 + 2 = 4 decimal)
11 << 2 = 1100 (3 times 2 twice = 12 decimal)
11 << 3 = 11000(3 times 2 three times = 24 decimal)

The << says "shift-left" Since one shift left is the same as multiplying by two, two shift lefts are the same as multiplying by 4. What is 11 << 6? Adding 6 zeroes on the tail gives 11000000. 3 times 2^6 in decimal is 192... Or WCPGW? If the MSB is being used as a sign this could be -112 in decimal.

You can usually say whether you want to work with ***signed*** or ***unsigned*** numbers. But remember these three things:

- There will be limits on the size of number you can store for any number of bytes.
- The sign trap we've just seen.
- Somebody may be sending you data in a slightly different bit or byte scheme. Check to see if signs are being used and if so how and check to see what the byte order is. It may seem strange to you but some systems store multiple bytes with the most significant byte before the least significant one, and others the other way round. 0xabcd being [ab][cd] or [cd][ab]

If you need to know this level of detail you will know where to find out the specification. A large proportion of programmers probably never touch binary arithmetic.

By the way, to divide by 2 shift right. (But quite frankly it will only be very exceptionally

that you can't get the programming language to do the work for you.)

Logic

Binary logic is simple to understand on the page as described below but when it comes to code you may need a quiet and darkened room for a few minutes to be absolutely certain what's going on.

The following riles apply to results of conditional tests, single bits and whole binary numbers. Here are the rules with some examples:

NOT
Flip 1 to 0 and 0 to 1
Often symbolised as `!`
NOT 1 = 0 NOT 0 = 1 NOT 00010101 = 11101010

AND
Result is 1 if both operands are 1
Often symbolised as `&&`
1 AND 0 = 0 1 AND 1 = 1 11110000 AND 10101010 = 1010000

OR
Result is 1 if either operand is 1
Often symbolised as `||`
1 OR 0 = 1 1 OR 1 = 1 11110000 OR 10101010 = 11111010

XOR
Exclusive or. Result is 1 if one *and only one* of the operands is 1
1 XOR 0 = 1 1 XOR 1 = 1 11110000 OR 10101010 = 01011010

XOR has a very useful property which makes it 'reversible': If foo XOR bar is buz then buz XOR bar gives foo. By treating each character of a message as 8 bits we can XOR them with a series of secret numbers to encrypt the message. When we want to decrypt it all we have to do if repeat with the same secret numbers and we get the plain message back again.

Set, flip and clear a bit

You might come across a specification like this: *`Set bit 3 in the control word when data is available`*. It looks like you're going to have to poke 1000 (or 100 if the specification is written with the LSB being called "bit 1") somewhere to flag we have data ready. So how about:

```
ControlWord = ControlWord OR 0x08;
```

Is this guaranteed to set bit 3 without upsetting anything else? Let's see if it does bit 3 correctly first. There are two possibilities bit 3 was 0 or it was 1:

0 or 1 gives 1 - correct
1 or 1 gives 1 - correct

Now what about the three zeroes (bits 2,1, and 0). Once again each of these bits starts out as either 0 or 1 and we mustn't change them

0 or 0 gives 0 - correct
1 or 0 gives 1 - correct

WCPGW? Are we absolutely sure that there are no spurious 1 bits? In this case yes, but what if we were ORing with a variable which represented a signed number - it

might have that top bit set.[377]

What about when we don't have any data to send and have to clear bit 3. Once again we need (1) a magic binary number and (2) the right operation. To force bit 3 to 0 we can AND with 0. If we're ANDing what do the other bits need to be? All 1s.

```
ControlWord = ControlWord AND 0xf7; //11110111
```

Now what about the three 1s (bits 2,1, and 0). Once again each of these bits starts out as either 0 or 1 and we mustn't change them

0 and 1 gives 0 - correct
1 and 1 gives 1 - correct

WCPGW? Are you sure that ControlWord is a single byte? It could be 2 or 4. (The generally accepted terminology is *unsigned* or *signed* applied to byte, *word* or *double word* for 1,2 and 4 bytes respectively.) If ControlWord is a 'word' then it is 2 bytes, 16 bits. All those bits might matter. Let's see what happens:

Say ControlWord is	**1111000010101010**
and we AND with	***00000000*11110111**
We get	**0000000010100010**

Oops! The top-end zeroes implied in our 0xf7 have zapped any top-end bits in ControlWord. Now we know about the problem we can of course use 0xfff7 if ControlWord really is a two byte unsigned word..

There is a handy way to get 0xfff7 from 0x08. NOT. This will work for 1,2 or 4 bytes, and is a clearer way of setting our bits. So to clear bit 3 we can write

```
Bit3Setter = 0x08;
Bit3Clearer = NOT Bit3Setter;   // 0x(f's as needed) f7
ControlWord = ControlWord AND Bit3Clearer;
```

or if we're in a hurry

```
ControlWord = ControlWord and (not 0x08); // brackets!
```

WCPGW? Watch for any signed bytes, words or double words creeping in.

There is an easy way to flip bit 3: Just XOR with Bit3Setter.

Testing

How can you tell if bit 3 is set? Firstly by ANDing with a 1 in bit 3 position (and 0s in all the other positions). This will give us

00001000 if bit 3 is set
00000000 if bit 3 was clear

All 0s is zero which is easy to test for (and may be defined as False)

Traps

In some languages minus 1 is taken as false. You might see a function like this

```
function foo(){  // returns integer
  if(something fails){return -1}
  ...
}
```

It is returning false the only way it knows how. foo() might be saying how many

---

377 For a small fee I'll show you my scars.

characters of data have been read from a file and return -1 to indicate the end of the file. Sometimes -1 is just a handy distinctive number, called *out of band*, which can't possibly be a real value and sometimes it is defined as false.

In some languages there are very particular rules about what is 'false'. For example in PHP a number of 0, and also null string are 'false'. This leads to shorthand in the code you could be reading which might not be obvious. Don't expect any shorthand true/false techniques to copy verbatim to another language.

Don't mix logic operations (the sort we were doing with ControlWord) with conditional logic (the sort in `if` statements.) You can get away with it by careful use of brackets but you are on very thin ice. Suppose we are closing down our program and want to see if we have any unsent data remaining which we ought to wait for. We can test to see if bit 3 is set (data ready in our hypothetical example) and take some action as a result:

```
    if (ClosingDown and ControlWord and 0x08){...
```

This code might possibly work, and pass your tests, but if it does it is a bit of a fluke, depending as it does on the precise way conditional logic, logic operations and arithmetic are mingled. A better way would be

```
    Bit3Set = ((ControlWord and 0x08) == 0x08);
    if (ClosingDown and Bit3Set){...
```

With typed languages (as those that deal with bits will be) Bit3Set would be declared as a boolean just so there wasn't any mystery and the compiler can help us if it spots us trying to use it for anything except true of false.

C. CD collection

This is the SQL code used to create the tables in the CD collection database in chapter 10. This works for MySQL but you may have to modify the code slightly for other DBMS. It is probably a good idea to use this as reference for manual entry so as to get experience with whatever database administration tool you're using.

```
CREATE TABLE artist (
  arId int(11) NOT NULL auto_increment,
  arName varchar(60) NOT NULL default '',
  arBirthYear int(11) default NULL,
  arDeathYear int(11) default NULL,
  arNotes mediumtext,
  PRIMARY KEY  (arId)
) TYPE=MyISAM;

CREATE TABLE cd (
  cdId varchar(20) NOT NULL default '',
  cdTitle varchar(50) NOT NULL default '',
  cdSubTitle varchar(50) default NULL,
  cdGeId int(11) NOT NULL default '0',
  cdNotes mediumtext,
  PRIMARY KEY  (cdId),
  FULLTEXT KEY cdNotes (cdNotes)
) TYPE=MyISAM;

CREATE TABLE cdar (
  caCdId int(11) NOT NULL default '0',
  caArId int(11) NOT NULL default '0',
  PRIMARY KEY  (caCdId,caArId)
) TYPE=MyISAM;

CREATE TABLE genre (
  geID int(11) NOT NULL auto_increment,
  geGenre varchar(30) NOT NULL default '',
  PRIMARY KEY  (geID)
) TYPE=MyISAM;
```

E. Compiling and linking. Libraries

The subject of this appendix used to be (and may still be for all I know) a core aspect of practical computer science. The modern programmer can get away without knowing a single word of this because nowadays things happen by magic.

However, if I should ever come across any person claiming to be a Real Programmer who can't be bothered to look beneath the surface then scathing remarks will be passed - in a very loud and sarcastic manner!

Summary

- A *compiler* normally creates an *object* code module from *source* code. (It may create 'byte code' which is run on a virtual machine.)
- Object code is only suitable for a specific target processor
- Different compilers can take the same source for different targets
- A *linker* creates an executable from object code modules
- A library is a module of object code or source code comprising a number of functions or access to resources (even perhaps objects)
- Module : Handy generic term for a unit of code that can (sort of) stand on its own.
- A dynamic link Library (DLL) is a file of executable code that can be loaded at run time (as opposed to compile time) to provide necessary services.
- Machine code is the native instructions that the processor uses. These are mostly 'simple' instructions such as Fetch the 2 bytes of data starting at the memory address as given by the value currently being held in the HL register interpreted as an address and put it in the CD register. and Jump back 52 bytes (by decrementing the program counter by 52) if the last sum we did using the DE register was zero. [378]
- Pseudo code or byte code is a semi-compiled binary form of almost-object code. This *is not* tied to one make and model of processor, but it is tied to a particular language which runs as a *virtual machine*. Java is an example.
- `make` is a program (of which many variants exist) which uses a configuration file to automate the creation of executable code with the minimum amount of re-compiling and linking.

Resources

A typical Beginner program will start along these lines:

---

[378] That's all you're getting on machine code in this book. Some people study ship building. Some people study timber construction. Some people study wood. Some people study the botanical aspects of wood. All these people are aware of the others, respect their knowledge and hope to be able to learn something useful to themselves from the others. I'm trying to tell you how to make money from ship building - don't ignore the timber you use but don't get distracted either.

0. With pencil and paper
- rest of program continues ...

What this is saying is that you need to collect these resources before you start. Also you'll check your paper is the right size and the pencil is sharp. I expect you've done exams which say "using the graph paper provided ..." or "indicate on the figure..." Just like these which are prepared for you (pre-printed) in the exam so as much as possible is done to prepare the resources a computer program will use. Whimsically we can think of a 'paper module' which 'provides' plain, lined and graph paper and a 'writer' module which 'provides' pencils and pens. In computer programming terms these modules provide functions which either do things or provide access to resources. (Another name for a module is a library, but see the end of the diversion.)

A module might provide access to and functions to control a printer in which case it might have an API¤ which allows you to do the following (in Fudge):

```
printerHandle = CreateNewPrinterInstance(PR_DEFAULT);
SetPrinterColour(printerHandle,PR_BLACK);  // print in black
SetPrinterUnits(printerHandle,PR_MM); // millimetres
DoPrintText(printerHandle,50,50,'Black cat in coal cellar');
DoPageEject(printerHandle);
ReleasePrinter(printerHandle);
```
[379]

Notice that printerHandle is resource (called a handle) and just like you might put a pen down after using it so you have to free handles or else after a while your program stops with an obscure error message because you've used all the available resources without releasing any of them.

Compile and link

The original two-step method[380] of building a complete computer program was

1 To convert text code into 'object code'. Here 'object' is NOT being used in the sense you understand it - In fact I don't know why it is called object code. This was as far as possible all the actual computer instructions that the processor would execute with gaps left for where the variables would go (foo goes here when we know where foo will be located) and question marks for functions that were called but not actually specified in the code. (We are going to call a function called foo with two integers and a real and get a real back)

   Getting this far is called *compiling*. All the syntax has been checked for readability, space allocated for variables, constants set and a list of 'loose ends' created. Also the resulting module has a table of function names in this module and how to call them. Compiled code is pretty much fixed so if we know that the start of the program happens to be loaded into memory at location X then we

---

379 This is what programming using Windows system API looks like. Hmmm. Shame about the lack of objects. The printer handle is a pointer to an object but the function name DoPrintText could easily be used elsewhere leading to confusion. (Something like this happened to me once which caused no end of problems until I twigged.)

380 Originally as for high level languages. There was very significant computer-life before this which is fascinating if you are tickled by the idea of how can you get a quart out of a pint pot and move technology on a step. (In some cases too many steps. See the history of LEO, Lyons Electronic Office and xerography.) There were some very brilliant people about in the days who knew they were pioneering but not where it would lead.

immediately know the ***offsets*** of all of the functions it contains. For example the first function starts at X+0, second at X+123, third at X+456 and so on. This will be really handy when *eventually* we come to run the program and something like

```
A = SIN(B)
```

where SIN() is in another module, becomes

```
500   Push memory location 400 on stack.
501   Jump to subroutine at 890.
890   initialise sine routine
      pop a real variable off the stack
      use value to compute sine
      put real result onto stack
      tidy up sine routine
502   Pop real off stack to memory location to 401
```

[381]

(400 contains B, 401 is allocated to A. The 'jump to 890' is filled in when we join modules together.)

2 One or more modules of object code are ***linked*** together. This does two things:

- Finds out how to connects up all the loose ends. When the ***linker*** comes across a function call in a module that references a module in another one then it can complete the missing pointers to addresses in different modules.
- The bits of object code are then packaged in a standard way that makes it possible for the file to be loaded and run as a program. This result is called an *executable*.

This means that you don't have to recompile all the trigonometric, statistical or whatever functions each time you tweak the working guts of your program.

If people use the same method of representing object code then my trigonometry library and Charlie's map library can be linked with your original work to make a globe. Interestingly my trigonometry could have originally been written in say the C language and the map library might be all binary data.

But

Object code is only suitable for running on one sort of processor.

A compiler is set up to take input in one language (source code - the sort you write) and output it in another (object code) dedicated to one make and model of processor. (There are linker and operating system issues too.) You can get all sorts of permutations of source and target but life is too short to contemplate all possibilities. This is why there are a handful of common computer languages for a handful of processors and operating systems. (If Richard Thompson creates a language called See for Laurence Tebbold's operating system called Loonix running on an Grimtel Bentium processor how many people will be queuing up to pay for it?)

Interpreted language

It is possible, but not very efficient convert source code into executable code line-by-

---

381 500 etc are used to vaguely represent address locations in memory. For illustration only. Take with pinch of salt. I've also used 'the stack' (remember that?) which is practically a universal feature of processors - It's a really handy way of transferring arguments to functions and getting the result back. (It's handy for other related things too.)

line as and when required. Writing an interpreter is quite easy providing the language demands are not too stringent. (The most famous interpreted language is BASIC although it comes in compiled versions as well.) The trouble is, although it isn't too difficult to transport an interpreted program to a new machine[382], each line needs a lot of work each time to do the translation between human readable form and machine readable form. Also a compiler can do wonders of optimisation which might double speed again.

There is an intermediate compromise which seems to work quite well and has come into vogue at least twice. Source code is compiled into a almost-object code variously called 'pseudo-code', 'p-code' or 'byte code'. This has had all the syntax checks done and is designed to be easy to convert into whatever machine code is required on the fly very quickly. Java is the best, but by no means unique, example of this today. This means that you can compile Java source code (to give .class files of byte code). These files are then linked and run together as required by the Java run-time virtual machine[383]. If a computer has a working Java runtime environment then it should be able to run all Java programs subject to version and resource limitations. This makes Java *portable*.

Dynamic linking

In a modern computer there are a dozen things going on at once. The operating system provides the overall infrastructure[384] onto which are attached specialised libraries called *drivers* which interface with hardware and other network devices, all manner of utility and snooping programs[385], really useful background applications, really useful foreground applications and of course your specially written programs.

Many of these programs want to do the same thing. For example, make a rude noise when something needs attention is common to many programs. It makes sense for the operating system to provide this basic level of service by providing a way for any program to call the 'RudeNoise()' function. It would be silly if every program had to duplicate this. Lots of programs want to paint on the screen, they want to use fancy fonts, they want to be able to find out what sort of format to use for the date and 1000 other things. These could be built-into the operating system library of functions but experience has shown that it is big enough as it is and instead it is better to have a colony of add-ons. In fact a lot of the operating system is made of these components. It is impossible to draw a line where the operating system stops and the many utility libraries starts. These 'ready-when-you-want-me' function libraries are often called Dynamic Link Libraries (DLLs) (@@@Pat? Equiv in *nix)

DLLs are great in a couple of respects: They provide functions 'cheaply' and can adapt

---

382 Actually it is a lot of trouble, or used to be when BASIC was common. Even today : Take nothing for granted.

383 Called Java Run Time Environment or JRE.

384 In a software sense.

385 In *nix world called daemons. In Windows called services. As a Real Programmer you should track down every single one and kill those you don't need and find out how they get started in the first place. Most Windows systems have a couple of dozen parasites.

relatively quickly to changing circumstances or bug fixes. The downside is that you are relying on third-party software out of your control. If somehow somebody 'upgrades' a DLL that you use then your program might stop working in an unexplainable way.[386] Some installations will install newer versions of these libraries - in the terms of the new program that's good but it might mean your program that relied on the old library suddenly stops working. Finding out what's wrong will be a nightmare and possibly the only answer is uninstall the new program and live with the fact that they won't work together.

Make

For a lot of people `make` is not history it's a way of life.

In this section we've scratched the surface of compile and link. Compile converts source into machine readable instructions. Link finalises how functions are called across modules and produces a single executable file.

Compiling will typically take a some source files in addition to the main one. These might be for example resources such as pictures or instructions in umpteen languages. In the C language 'header' files are used to tell the compiler not to worry about certain loose ends. So you need to recompile if any of these files has changed. Conversely - this is the important bit - you *don't* have to compile if nothing has changed.

When linking together an application you will have one or more object files as components. If any of these change then you need to re-link.

Some bright spark twigged that if you described these dependencies (in a text file called a '***make file***') by simple rules then you could write a program that read the rules, discovered which files had been altered and do the necessary actions. This meant that once you'd put the ***make dependencies*** into the make file all you had to do was type `make` and the necessary compiling and linking (but no more) would be done. We are talking minutes even for a small change to the source. Nowadays computers are faster but programs are bigger so this sort of efficiency is absolutely vital.

---

386 A case in the news just today. 99% of programmers don't need to worry - there's nothing you can do - and you'll only be using these functions via an intermediate layer.

D. TinyDate object

Dates and times (the two are often joined at the hip) are full of traps and need careful handling. Whatever programming language you use you should have access to some form of date, time and/or combined date and time facilities.

Many date-times are represented as the number of seconds since some moment. WCPGW? 'Some moment' is in the 00:00:00 on the 1st Jan 1970 and you want dates in the 1790s perhaps.[387] Some people find dealing with dates particularly troublesome - it is a frequently raised topic in support newsgroups.

In my experience dates simply based on ticks of a clock are not sufficient to represent the dates we use in the real world. For example on your employment record file there's a space for date you joined the company and date you left... ...And what is the 'date' for the date you left if you still work here? What year did you get married? What's the 'date' for 1982.[388] Dates are not linear, they're just weird.

> Many companies work on 13 four-week periods in a year not 12 months. So does your software allow both options? Some work on a 4-4-5 weeks per 'month' cycle - When analysing production or sales can the raw figures be adjusted in proportion?

Here is the full set of date situations that we need to cater for:

- Not a date
- Beginning of time
- End of time
- Specific year
- Specific month (no year/day)
- Specific day (no month/year)
- Specific Year/month (no day)
- Specific Month/day (no year)
- Year / month / day

And the arithmetic we can do in very interesting:

- How many days between 1875 and 1900? - Not a valid question
- What's the interval between 1875 and 1900 - 24 years (months n/a days n/a)
- Is 14th October 2006 before November 2006 - Yes
- Is 14th October 2006 before October 2006 - No
- Is 14th October 2006 after October 2006 - No

---

387 Many UNIX time stamps count from this time.

388 When I took my cat to the vet they asked how old he was. Sensible question. As a rescue cat I didn't know but guessed 3 years. Now on the vets database the cat has an actual date of birth exactly three years from the day I first took him down there.

With many being not valid

Clearly we are in a different world to simply working in seconds.

My requirements

What I needed was a date with

- this sort of flexibility
- that could be stored and retrieved efficiently in a database
- and be sorted.

My solution

In order to work with these dates I wrote a set of functions which were put into my useful Delphi library. With these I could convert everyday strings to and from tiny dates, convert Delphi's TDateTime type to and from tiny dates, work with database fields of three characters and compare, including "definitely before" etc. The principle was to use 21 bits packed into three bytes to be converted into a three character string. Having the string representation meant immediate access to ready-made sorting and database features.
This library of functions has served me well since 1996.

An alternative using objects

If objects are so wonderful shouldn't these functions be converted into methods of a TinyDate class? At this moment in time I don't see any need for sub-classing and if there is only a single three character string for data why bother? Because at some future time the internal format might change and we need to internalise that mechanism for future-proofing. Also an object can be protected and processed in useful ways within an OO language that lots of functions can't.[389]

> In an OO implementation there would need to be the usual black-box object which has data and methods. That's normal, but in addition there needs to be a pure data representation that is suitable for use in a database. With the non-OO of Delphi there was no 'internal state' of an object beyond the data-only representation.

Exercise

The following is presented as an exercise to give you experience of coding and testing a class in an OO language. You'll want to have finished chapter 14 first. It will probably take between five and ten hours, but will be a very good investment.

Specification

Purpose : To implement the date functionality described above as an object. In particular:

- Encoding non-linear and special date values.
- Translating (with error trapping) between system dates
- Translating (with error trapping) between everyday string representations.

---

389 For example synchronizing between threads.

- Providing a value representation that can be used in a database and will sort into 'order'.
- Providing comparison and arithmetic operators capable of dealing with non-specific dates. As far as possible, possibly with fudge-rules. For example the number of days difference.

Implementation tips

- The three character 'string' used for database and sorting used in my existing library is as follows: This might be a place to start. *You do not have to follow this format or stick to these limitations.*

```
Data format
|  Bytes/Characters expressed left to right
|
|  byte/char 1           byte/char 2          byte/char 3
|  --------------        --------------       --------------
|  0 1 y y y y y y       0 1 y y y m m m      0 1 m d d d d d
|
|  Strange layout is because
|      (1) All high bits = 0
|      (2) No char can ever be all 0 bits
|  Year offset = 1700  which allows dates from 1703 to 2200
|  Year = 0..500     0=No year        1=Not valid date
|                    2=Beg.of.time  511= EndOfTime
|  Month = 1..12     0=No month
|  Day = 1..31       0=No day
```

- You may want to provide constants to represent Beginning of time, End of time and Not a Date.

- You may want to provide a way to say which parts of the date are specified. this could be by some constants used as in

```
if (date.Detail()==TD_MONTHONLY){ ...
```

or you might prefer test methods as in

```
if (date.IsMonthOnly()){ ...
```

or both. Experience might indicate where you want to find out more general (what rather than how) information as in

```
if (date.IsReal()){...
```

- YCPL will have date handling already. To what extend do you want to avoid duplication and what can you re-use.

Prototyping, Proof of concept

This will probably be a big part of this exercise. One of your first jobs will be to split it up into sub-tasks then tackle them in an organised fashion.

Should there be any class methods? For example to convert a tiny date 'database value' into another form without the user having to create an intermediate tiny date object?

Testing

- There will probably be quite a bit of experimental data at the prototyping stage which can get carried across to an exercise program in the same way as chapter 14.

- There will be quite a few combinations of possible dates and the operations on them which will all need to be tried. This is probably best done with a comprehensive automated test program rather than a few examples in a user-friendly 'test' program.
- The specification above should give you some hints about splitting the testing into manageable chunks.
- With many combinations to test you need to think about fully automated testing otherwise you'll never be able to spot erroneous needles in a haystack of results. (This is a tough challenge to deal with 'off-the-cuff' so give it some thought. You might want to ask yourself what you'd do if you were able to share test data with somebody else.)
- What are the database implications? Is the choice of database irrelevant?

Deliverable results

- Finished standard code
- Properly documented API including a well laid out specification of the way tiny dates are represented in a database.
- User guide including 'test program' example use with limitations and restrictions.
- 'Sales brochure' of a few paragraphs[390]

Review

After using tiny dates you'll wonder how you managed before. So this library will become part of your standard toolkit. It will evolve over time so you'll need to look after your development files carefully.

Finally

- I've seen '30 lines of code per day' (whatever that means) at the 'industry average'. Even with all the prototyping, documentation and testing you will do a lot better than that - but it won't happen in a morning.
- It's a funny thing, but it is virtually impossible to discuss or sell something unless it has a name. 'Tiny date' is my name which arose through particular circumstances now lost in the mists of antiquity. Perhaps you can come up with an improvement.

---

390 You might want to write this before head-down coding. Even if you'd been commissioned by a user to create this library it is handy to have a few paragraphs that you can show them to keep them interested and convince them that although it is taking longer than expected it will be worth the wait.

E. Compiling and linking. Libraries

The subject of this appendix used to be (and may still be for all I know) a core aspect of practical computer science. The modern programmer can get away without knowing a single word of this because nowadays things happen by magic.

However, if I should ever come across any person claiming to be a Real Programmer who can't be bothered to look beneath the surface then scathing remarks will be passed - in a very loud and sarcastic manner!

Summary

- A *compiler* normally creates an *object* code module from *source* code. (It may create 'byte code' which is run on a virtual machine.)
- Object code is only suitable for a specific target processor
- Different compilers can take the same source for different targets
- A *linker* creates an executable from object code modules
- A library is a module of object code or source code comprising a number of functions or access to resources (even perhaps objects)
- Module : Handy generic term for a unit of code that can (sort of) stand on its own.
- A dynamic link Library (DLL) is a file of executable code that can be loaded at run time (as opposed to compile time) to provide necessary services.
- Machine code is the native instructions that the processor uses. These are mostly 'simple' instructions such as Fetch the 2 bytes of data starting at the memory address as given by the value currently being held in the HL register interpreted as an address and put it in the CD register. and Jump back 52 bytes (by decrementing the program counter by 52) if the last sum we did using the DE register was zero. [391]
- Pseudo code or byte code is a semi-compiled binary form of almost-object code. This *is not* tied to one make and model of processor, but it is tied to a particular language which runs as a *virtual machine*. Java is an example.
- `make` is a program (of which many variants exist) which uses a configuration file to automate the creation of executable code with the minimum amount of re-compiling and linking.

Resources

A typical Beginner program will start along these lines:

---

391 That's all you're getting on machine code in this book. Some people study ship building. Some people study timber construction. Some people study wood. Some people study the botanical aspects of wood. All these people are aware of the others, respect their knowledge and hope to be able to learn something useful to themselves from the others. I'm trying to tell you how to make money from ship building - don't ignore the timber you use but don't get distracted either.

0. With pencil and paper
- rest of program continues ...

What this is saying is that you need to collect these resources before you start. Also you'll check your paper is the right size and the pencil is sharp. I expect you've done exams which say "using the graph paper provided ..." or "indicate on the figure..." Just like these which are prepared for you (pre-printed) in the exam so as much as possible is done to prepare the resources a computer program will use. Whimsically we can think of a 'paper module' which 'provides' plain, lined and graph paper and a 'writer' module which 'provides' pencils and pens. In computer programming terms these modules provide functions which either do things or provide access to resources. (Another name for a module is a library, but see the end of the diversion.)

A module might provide access to and functions to control a printer in which case it might have an API¤ which allows you to do the following (in Fudge):

```
printerHandle = CreateNewPrinterInstance(PR_DEFAULT);
SetPrinterColour(printerHandle,PR_BLACK);  // print in black
SetPrinterUnits(printerHandle,PR_MM); // millimetres
DoPrintText(printerHandle,50,50,'Black cat in coal cellar');
DoPageEject(printerHandle);
ReleasePrinter(printerHandle);
```
[392]

Notice that printerHandle is resource (called a handle) and just like you might put a pen down after using it so you have to free handles or else after a while your program stops with an obscure error message because you've used all the available resources without releasing any of them.

Compile and link

The original two-step method[393] of building a complete computer program was

1 To convert text code into 'object code'. Here 'object' is NOT being used in the sense you understand it - In fact I don't know why it is called object code. This was as far as possible all the actual computer instructions that the processor would execute with gaps left for where the variables would go (foo goes here when we know where foo will be located) and question marks for functions that were called but not actually specified in the code. (We are going to call a function called foo with two integers and a real and get a real back)

Getting this far is called *compiling*. All the syntax has been checked for readability, space allocated for variables, constants set and a list of 'loose ends' created. Also the resulting module has a table of function names in this module and how to call them. Compiled code is pretty much fixed so if we know that the start of the program happens to be loaded into memory at location X then we

---

392 This is what programming using Windows system API looks like. Hmmm. Shame about the lack of objects. The printer handle is a pointer to an object but the function name DoPrintText could easily be used elsewhere leading to confusion. (Something like this happened to me once which caused no end of problems until I twigged.)

393 Originally as for high level languages. There was very significant computer-life before this which is fascinating if you are tickled by the idea of how can you get a quart out of a pint pot and move technology on a step. (In some cases too many steps. See the history of LEO, Lyons Electronic Office and xerography.) There were some very brilliant people about in the days who knew they were pioneering but not where it would lead.

immediately know the ***offsets*** of all of the functions it contains. For example the first function starts at X+0, second at X+123, third at X+456 and so on. This will be really handy when *eventually* we come to run the program and something like

```
A = SIN(B)
```

where SIN() is in another module, becomes

```
500   Push memory location 400 on stack.
501   Jump to subroutine at 890.
890   initialise sine routine
      pop a real variable off the stack
      use value to compute sine
      put real result onto stack
      tidy up sine routine
502   Pop real off stack to memory location to 401
```

[394]

(400 contains B, 401 is allocated to A. The 'jump to 890' is filled in when we join modules together.)

2 One or more modules of object code are ***linked*** together. This does two things:

- Finds out how to connects up all the loose ends. When the ***linker*** comes across a function call in a module that references a module in another one then it can complete the missing pointers to addresses in different modules.
- The bits of object code are then packaged in a standard way that makes it possible for the file to be loaded and run as a program. This result is called an *executable*.

This means that you don't have to recompile all the trigonometric, statistical or whatever functions each time you tweak the working guts of your program.

If people use the same method of representing object code then my trigonometry library and Charlie's map library can be linked with your original work to make a globe. Interestingly my trigonometry could have originally been written in say the C language and the map library might be all binary data.

But

Object code is only suitable for running on one sort of processor.

A compiler is set up to take input in one language (source code - the sort you write) and output it in another (object code) dedicated to one make and model of processor. (There are linker and operating system issues too.) You can get all sorts of permutations of source and target but life is too short to contemplate all possibilities. This is why there are a handful of common computer languages for a handful of processors and operating systems. (If Richard Thompson creates a language called See for Laurence Tebbold's operating system called Loonix running on an Grimtel Bentium processor how many people will be queuing up to pay for it?)

Interpreted language

It is possible, but not very efficient convert source code into executable code line-by-

---

394 500 etc are used to vaguely represent address locations in memory. For illustration only. Take with pinch of salt. I've also used 'the stack' (remember that?) which is practically a universal feature of processors - It's a really handy way of transferring arguments to functions and getting the result back. (It's handy for other related things too.)

line as and when required. Writing an interpreter is quite easy providing the language demands are not too stringent. (The most famous interpreted language is BASIC although it comes in compiled versions as well.) The trouble is, although it isn't too difficult to transport an interpreted program to a new machine[395], each line needs a lot of work each time to do the translation between human readable form and machine readable form. Also a compiler can do wonders of optimisation which might double speed again.

There is an intermediate compromise which seems to work quite well and has come into vogue at least twice. Source code is compiled into a almost-object code variously called 'pseudo-code', 'p-code' or 'byte code'. This has had all the syntax checks done and is designed to be easy to convert into whatever machine code is required on the fly very quickly. Java is the best, but by no means unique, example of this today. This means that you can compile Java source code (to give .class files of byte code). These files are then linked and run together as required by the Java run-time virtual machine[396]. If a computer has a working Java runtime environment then it should be able to run all Java programs subject to version and resource limitations. This makes Java *portable*.

Dynamic linking

In a modern computer there are a dozen things going on at once. The operating system provides the overall infrastructure[397] onto which are attached specialised libraries called *drivers* which interface with hardware and other network devices, all manner of utility and snooping programs[398], really useful background applications, really useful foreground applications and of course your specially written programs.

Many of these programs want to do the same thing. For example, make a rude noise when something needs attention is common to many programs. It makes sense for the operating system to provide this basic level of service by providing a way for any program to call the 'RudeNoise()' function. It would be silly if every program had to duplicate this. Lots of programs want to paint on the screen, they want to use fancy fonts, they want to be able to find out what sort of format to use for the date and 1000 other things. These could be built-into the operating system library of functions but experience has shown that it is big enough as it is and instead it is better to have a colony of add-ons. In fact a lot of the operating system is made of these components. It is impossible to draw a line where the operating system stops and the many utility libraries starts. These 'ready-when-you-want-me' function libraries are often called Dynamic Link Libraries (DLLs) (@@@Pat? Equiv in *nix)

DLLs are great in a couple of respects: They provide functions 'cheaply' and can adapt

---

395 Actually it is a lot of trouble, or used to be when BASIC was common. Even today : Take nothing for granted.

396 Called Java Run Time Environment or JRE.

397 In a software sense.

398 In *nix world called daemons. In Windows called services. As a Real Programmer you should track down every single one and kill those you don't need and find out how they get started in the first place. Most Windows systems have a couple of dozen parasites.

relatively quickly to changing circumstances or bug fixes. The downside is that you are relying on third-party software out of your control. If somehow somebody 'upgrades' a DLL that you use then your program might stop working in an unexplainable way.[399] Some installations will install newer versions of these libraries - in the terms of the new program that's good but it might mean your program that relied on the old library suddenly stops working. Finding out what's wrong will be a nightmare and possibly the only answer is uninstall the new program and live with the fact that they won't work together.

Make

For a lot of people `make` is not history it's a way of life.

In this section we've scratched the surface of compile and link. Compile converts source into machine readable instructions. Link finalises how functions are called across modules and produces a single executable file.

Compiling will typically take a some source files in addition to the main one. These might be for example resources such as pictures or instructions in umpteen languages. In the C language 'header' files are used to tell the compiler not to worry about certain loose ends. So you need to recompile if any of these files has changed. Conversely - this is the important bit - you *don't* have to compile if nothing has changed.

When linking together an application you will have one or more object files as components. If any of these change then you need to re-link.

Some bright spark twigged that if you described these dependencies (in a text file called a '***make file***') by simple rules then you could write a program that read the rules, discovered which files had been altered and do the necessary actions. This meant that once you'd put the ***make dependencies*** into the make file all you had to do was type `make` and the necessary compiling and linking (but no more) would be done. We are talking minutes even for a small change to the source. Nowadays computers are faster but programs are bigger so this sort of efficiency is absolutely vital.

---

399 A case in the news just today. 99% of programmers don't need to worry - there's nothing you can do - and you'll only be using these functions via an intermediate layer.

F. Filing system

Help yourself

Say goodbye to "my documents".
Say hello to PACT.

Say goodbye to "it's here somewhere - this looks like the latest copy".
Say hello to "I have it in my hand"

Say goodbye to "what did we agree?"
Say hello to "I have a note here"

Say goodbye to "will you spend time looking for me?"
Say hello to "look in the filing system under Foo"

In this appendix I will show you how with a little bit of organisation you can astonish your friends, colleagues, and especially confound, your enemies by having important information ready to hand. It is not rocket science but it is vital. Just because you're a wacky free-wheeling programmer doesn't mean you have to live in a shambolic world of missed appointments, lost code and guesswork. You can still be a creative genius by knowing exactly where your tools are - and your tool is information. In fact you *will* amaze people by how well organised you are - they soon know not to try to bamboozle you like they do others.

The parts

Mental filing system

This important system is rather error prone, but is good at recalling things in a general way when prompted. As you get older so you may find your memory and capacity to juggle with a number of things at the same time starts to decay. A lot of programming consists of diversions and loose ends - get into the habit of making to-follow-up notes either in the code or on a jotter. Many programmers type in a code that can be searched for such as @@@ that is unlikely to appear otherwise perhaps with a shorthand for what needs revisiting. It's a great aid to rapid progress not to have to stop your flow to investigate, validate, document or explore minor points.

An experienced chef can be chopping, cleaning, mixing, grilling, simmering, roasting and finishing all at the same time using all of the facilities of a kitchen. Your task is similar except that you have to do it all in your head. You should be training your brain (see chapter 18) to deal with six things-to-do at the same time and being able to switch instantly between them.[400]

---

400 Also practice being able to break a task down into levels and segments so that each task is a manageable size in a logical structure.

Jottings

Some people find that writing something down and referring to it once or twice is a powerful memory aid. If you are one of these people, be creative with layout and coloured pens.

Sketch, explain, prepare. Multi-coloured masses of ideas and shopping lists. Recall that your mental filing system will be fully occupied with programming so you must get into the habit of unloading bits to paper even if only for half an hour. I know this sounds too basic for any intelligent person but yes, you need paper and a scribblything¤; and you need it ready to hand. Just see how quickly a desk jotter fills up with odds and sods. Just see how ideas develop if you sketch them out on a larger piece of paper. Just see how many things you forget to buy when you don't make a shopping list. Jottings have a short life and not much context. However they are flexible and nicely fill the gap between your mental filing system and more permanent records.

Notes

Tip. Use tick boxes - 9 -against things to do or follow up. You can scan a page and quickly see the ones you haven't managed to cross out yet.

Every programmer carries a hard-back notebook. Everything of any importance that isn't otherwise filed goes in it. Records of meetings, system configurations, phone numbers, wiring diagrams and even, ahem, passwords for your client's systems.[401] Your notebook is a precious part of your working toolkit. Look after it, put your name and address on it. 95% of the stuff in there will never be needed - but the other 5% is gold-dust. You made a note of what happened at the last meeting and now you ask why the foo that was going to be provided at this one hasn't been. The specification you are working on as discussed has somehow changed when it gets circulated? What's going on? "I can tell you the exact version of foo we are using at the moment and also the gotchas¤ to go with updating". Because most people are not used to working with large amounts of abstract information, technical data, what-ifs, having to marry the big vision with minute precision and picking the reality out of what people say, they are not geared up to being organised like you and will be amazed. (And probably cross when you open your book and the sunlight shines on dark areas.)

Other paperwork

You *need* a filing cabinet. If only to put copies of invoices and time sheets in. I'm not a Clean Desk Nazi but being able to put a project to bed for the time being knowing that it isn't too far away if the phone rings is really handy. If other people are likely to share this filing system then have a rule that either the whole package is extracted and replaced (so you can shout "who's got the Foo stuff" and caring for it is that person's responsibility) or a place marker is used to indicate temporary theft. It will help if you

---

401 I'd like a pound for every time I've referred to user's passwords when they have lost them. Fortunately the security risk is low because I know what they are but a casual reader wouldn't. Not ideal but very practical.

clearly and systematically label the dividers so that people don't start looking in all sorts of places.

The well read programmer

You need some books around if only just to show off your interests in cats, cacti and castrati to nosy parkers[402]. Seriously, books can be relaxing - If you're a good programmer you (a) need to rest - having a routine schedule if possible[403], and (b) can enjoy ostentatiously 'doing nothing' while relaxing with a book. The rest of office-life and lesser programmers value the appearance of activity; and yet if you look, they have random periods of interfering gossip, coffee making and rearranging their desk-junk. *You* on the other hand are always in control, always confident and normally relaxed - until you switch into code-mode when your silent, intense concentration is a sign of a great mind at work.

The more important or highly charged an issue is the more difficult it is for people to approach the subject. It's a lot easier if they can start by talking to you about a 'harmless', non-technical matter before tackling the ticking bomb. So you need to be 'talkable to' - (you know, just like a real human being with hobbies and interests) - in order to get these confidences, opinions and observations that show you a view 'behind the scenes'.[404]

Review

We haven't got onto electronic filing yet, but I think you can see the value of being organised and the necessity of using the right tools for the job. Try the alternatives if you don't believe me.

Because you work with such a slippery commodity, information - most of it abstract, in such a wide variety of forms you have to organise it to be good at your job. Other people might work with information, but so what if the graphs are pretty but fact free, or factual but not pretty? So what if the press release is vague? What sort of abstract concepts are the bean counters working with? Why are good programmers rare and treated with respect? Because they have the skills to tame information and are not afraid to use them.

---

402 Magazines are strictly out! They are entertainment for people who like looking at the pictures.

403 If you work '9-to-5' then always have a minimum of half an hour's fresh air away from a screen at lunch time. Try the alternative and see how drained you get by the end of the day. Experiment with different breaks and work patterns to see what works best for you in your environment. In my first job we could earn overtime which was a nice little bonus - I found that a switch of project for the overtime session gave a fresh start.

404 Warning - A lot of 'take it from me' confidences are not as reliable as they might appear. Furthermore your correspondent might be on a fishing expedition for devious reasons. Nevertheless it's good to have an ear to the ground and people who want you 'to be on their side'.

And so to electronic filing

If the last section was about the joys of being organised, this one is a rule book for safe storage. Naturally nearly all of your work is computerised. There is no perfect system for electronic filing - there comes a point where trade-offs have to be made. Let's look at the various aspects common to computing then those extra issues specific to programming.

Physical storage

You can probably store all of you code, documentation, development tools, servers and test data on a hard disc. So WCPGW physically? Of course you need good, reliable backups. By the way, that's not any old 'of course you do' but a "STOP AND BACKUP NOW" 'of course'. I told you this was the rule book section.

Almost the only thing I can say about your backup strategy is that it depends on what you're trying to protect against what.[405] Oh, I see you've got a Raid file server... no problem there then ... for values of no = any eggs left in basket. Oh, I see you backup the whole disc to tape every night... no problem there then ... for values of no = restoring works as advertised I expect? Oh, I see all your data is copied to a laptop daily and taken off site. No problem there then... for values of no = "now can anyone think what was confidential on that stolen laptop?" Oh, I see you keep daily backups of your code base. No problem there then... For values of no = "can we go back to last December's version".

Unfortunately it is easier to point out what might possibly go wrong than for me to tell you how to do it right. However one way to make more backups, on the principle that at least some of them will cover the target is to make backing-up simple and quick. Programs to do this are probably best written as batch/shell programs that can be invoked with a standard command.[406] As we'll see in a moment, different types of files need different strategies.

PACT

A handy acronym for four logical division of hard disc data. You need to understand this for the highest level of file structuring, and also *so your programs are well designed for the convenience of users.*

Programs - Archive - Current - Temporary

Program files are sourced from elsewhere and can be re-installed. This includes operating system, utilities and applications.

- In an ideal world configuration files are not stored with the applications
- Program files should be read-only

---

405 I can also say with confidence that it won't be perfect and Sod's law applies.

406 I have safety.bat which zips all the files in the current directory to a file called safety.zip and renames any previous safety.zip to safety.001 and any safety.001 to safety.002 etc. Then I have a one line batch file called grab.bat which calls the grab.bat file on my removable discs - Each removable disc 'knows', because of the custom batch file on it which files it should be backing up. safety.bat is used before working on a project, grab.bat can be invoked at the end of a long day without thinking when time is of the essence to get down the pub before closing time.

- Data files associated with applications shouldn't be in the program area
- You need to backup the installer as well as a snapshot.[407]
- Program files will only change rarely and monitoring when this happens is probably a good idea.

Note that many program installers want to install itself in the root or mix data files and programs. Always follow the advanced installation options to try to avoid this.

Archive files are still on hard disc, but are generally write once. This is where you plant the root of your reference library, driver installers, customer-specific references and any other reference resources.

- The archive will be continuously growing but files will seldom change.

Temporary files are not going to be backed up. All downloads go here to begin with as do tests and log results and quick safety backup copies. The purpose of this area is to avoid cluttering other areas with files which are not worth keeping or perhaps you have just in case you need them later.

Current is where user data goes. In the case of a programmer much (but not the IDE and utilities and 3rd party libraries) are in here. Frequent backing up is required (even though typically only a fraction will change between backups).

Deploying your programs

Now you might be wondering why so many programs insist on running in first level directories (ie straight off the root) or mix programs with user data and temporary files. The answer is ignorance, arrogance and laziness plus a bit of 'the users are too thick to work with multiple directories'.[408] Nowadays, with sophisticated installation programs easily available there is no excuse for not putting the components of your software in the right places.

- Configuration files that are fixed at installation time should stay with the programs and associated files in the program area. (Although *in theory* documentation belongs in the archive, because it is so closely associated with a program it is better being considered as 'program'.)
- Configurations and user preferences that change should go in the current area. (Possibly on a user-by-user basis - you need to understand the target operating system.)
- Data directories are of course in the current area.
- Log files will probably go in the temporary area, although audit trails might be candidates for the archive area.

You need to tell the users and/or administrators how files are distributed, tell them why, and give instructions for backing up, weeding and restoring.

If you can incorporate a one-click backup utility then that would be excellent. It is a good programming exercise and will be an investment. Usability and varied target systems make this task tricky.

---

407 A good reason for avoiding magic on-line updates. You can't recreate you system.

408 There is some truth in this last bit, but your program should be doing the work seamlessly for them.

Your own development system

You need a consistent, navigable and precisely controlled electronic filing system. In my experience this has to adapt to deal with the restrictions imposed by various development environments.

Having understood the PACT basics, and taken into account any security issues that may affect your system you now have to figure out how to farm-out a lot of different (they're relatively easy) and 'the same' (versions are horrible to deal with) types of file.

The following are suggestions to help you work out for yourself what's going to be practical in your environment.

- Keep directory names short. eg "Dvnt" not "Development", "Ref" not "Reference"[409]
- All sections will divide and divide and divide.
- Set up a *reference* section (divided by subject of course) where your general reference materials go.
- Set up a *store* section for
  - Master copies of application installers, drivers, libraries. (You might keep some of this off-line)
  - Handy resources such as images.
  - Any other resources that are not to be thrown away
- Set up a *temporary* section for almost anything: Backups, mirrors, log files, experimental results, intermediate files, caches, trial installations.... (For what it's worth all my downloads go to \temp\dload which lets me pick the bones out later at my leisure.)
- Set up a programming tools section for editors, compilers, IDEs etc. You'll need to manage this section carefully because most of it will never change but there may be configurations and other resources that are tightly linked with the tools that will change and can be a big pain to recreate from scratch.
- As far as possible separate from the programming tools section, set up a programming resources section for third party components and libraries. This is a tricky area as there are often conflicts where some items are to be shared while others need to be in a separate space.
- You might want a separate section for servers and shared resources such as databases which have their own sub-filing system, security considerations and backing up requirements.
- Set up a private section for non-project items. For example invoices, timesheets, blog and other non-programming documents.
- ...and finally one section for projects.

The proj section will develop organically. Don't be afraid to create new directories but always make sure that when creating a directory it is specific to it's parent. After reading chapter 14 you will probably be thinking of a handful of directories per project. This might be overkill in the case of a relatively small item as we looked at in chapter 14, where the various files could be distinguished by a clear naming convention. However a larger project will have correspondence, specifications, drafts, previous

---

409 **Watch the case in *nix. Decide NOW if you're going to use leading capitals or not and stick to it.**

versions, test data and results and perhaps a couple of dozen program files. At some stage, preferably at the beginning when you've got a feel for the depth of the task, you can add data, docs, instln sub-directories etc.

G. Quality in a nutshell

I've collected a few concepts that should clear the fog surrounding 'quality'.

- *Adopting them yourself* will put you far in advance of the field and very efficient.
- *Applying them to user's systems* will be a revelation in more ways than one.

It is extremely satisfying to see how to improve things and give people the tools to do so... ...But be prepared for deeply entrenched resistance. Unless you can find a champion¤ you will only achieve your goals by subterfuge and fait accompli.

Quality system - QS

This is the complete machinery of delivering good results. Competence and diligence of individuals is only a part of this - Good management is the essential ingredient. Typically a QS has ways to know what it is trying to deliver, monitors how well things are going, acts to correct problems and is simple enough for somebody to guide the system and exploit 'new-improved' possibilities.

Quality assurance - QA

This is the confidence that everybody has in the QS to deliver what it is supposed to. For example an annual review of safety (if carried out properly and acted upon) contributes to our knowledge of what could go wrong and where the risky bits are and where we must take more care. QA is not about wrapping everything up in a layers of protective paperwork but about knowing you're taking quality seriously, able to spot risks in advance in order to deal with them appropriately and carrying out enough checks to catch situations before they get out of hand.

In practical terms we're trying to make it easier to get things right, or simple to spot mistakes when they do happen. For example important numbers have a check-digit which will spot common transcription errors.

Costs

If is often difficult to see a tangible result for all the effort that goes into quality. Making an approximate estimate of the cost of the mistakes is difficult. Many mistakes go unreported. We're not just talking about mistakes but things that fall short of expectations. For example a user may be unhappy because their program isn't a wonder cure they had hoped for - and so give up completely on getting the best out of what is possible.

Just because it is difficult to link cost of quality to benefits doesn't mean there isn't one. But the reverse of the coin is that just because you spend a fortune on quality doesn't mean you'll get your money's worth.

So lets deal with two things straight away: Defining what we're trying to do and using this knowledge to put the right quality systems in place in order to target our efforts for the maximum result.

Quality goals

Ask somebody what their Quality Goals are and they will probably just look at you blankly.

Quality goals for code

In plain coding such as we did in the second half of chapter 14 the mechanical methods used to identify WCPGW and tell the user how to use the program achieved some quality goals that were taken for granted:

- The program must work and be useful
- The program must not crash horribly
- The user must be able to grasp how to use it
- The code must be maintainable

Often this set of goals is pretty much standard. However there may be additional risks and costs to avoid and bonuses to be won. Perhaps the program is safety-critical and needs to pass specified tests. Perhaps the code will be adapted for multiple languages. Perhaps a particular sort of user will have special needs. Perhaps a 'working version' needs to be ready by a certain date to enable another part of the project to continue. •

You can think of quality goals in this context as a combination of trapping all WCPGWs and 'why are we getting paid'.

Achieving user's quality goals

The user's quality goals should be identified at the systems analysis and design stages. Remember that your program should be making it easier for people to do their job correctly - What is that job and what is 'correctly'? It is vital that you understand this.

For example why do the stores keep running out of parts when the computer says you have plenty in stock? Perhaps it is because booking out is so tedious that it gets 'left until later'. To achieve the quality goal of 'not running out of parts' you may need to implement a faster or easier booking-out system.

Creating quality goals

I wrote quality goals for a NHS client. There were twelve headings the first of which was "Detecting disease" with 13 goals one of which was "Correct diagnosis leads to appropriate treatment." Motherhood and apple pie you cry - Anyone could write out a list like that. Correct! Go ahead. It really is quite easy. (A one bottle of wine job - ie. about 3 people and 25 minutes.) It is all very well to trumpet slogans such as "Correct diagnosis leads to appropriate treatment." but how will we turn that into something practical?

- By using our knowledge to investigate how we do things at the moment.
- By asking how we can measure how well we're doing.
- By considering the methods used elsewhere that we might transplant.
- By asking what do we mean by 'correct diagnosis' and how, in this case, we connect to the 'appropriate treatment'.
- By asking 'what can go wrong'.
- By asking 'who should do this'.

These then lead to suggestions for improving methods, a decision to computerise the new methods rather than rehash the current hotchpotch, and making sure that the new systems were designed from the ground up to deliver these goals. That's how to build-in quality.

Quality goals for decision making

Another way to use the quality goals is as a set of prompts for asking questions about suggested changes. "Everyone's using the Ruby On Rails framework - so should we"... Hmmm. So how will that help to deliver our quality goals? Perhaps one of your quality goals is "Keep abreast of technical innovations in programming" and another "Provide opportunities for team members to develop their research, analysis, presentation, and self-project-management skills" so there's a handy match between suggestion and quality-driven reasons if it becomes "Give Karen the job of evaluating RoR and coming up with a report and presentation".

Observation Decision Action - Transparent decision making

How obvious is it that O,D and A are separate things, requiring different tools, skills and diligence. Although a skilled expert (and 'intelligent' instrument) is probably capable of performing all three in one seamless activity, it would be possible to ask them (audit) each part on its own.

Let's look at an example: Is this program ready for release?

- Observation : What's on our checklist? Do we take somebody's word for it that the documentation is complete or review it for ourselves? Do we have tools to measure conformance to standard - if so are they good tools well operated?
- Decision : How do we use the data collected to decide what to do? Are we operating a plain pass-fail or something more sophisticated?
- Action : Implementing the changes, re-work, re-test and release notes.

That's so straightforward you might be wondering what the big deal is...

- There could be different actors involved at each stage. For example it might be the programmer's job to complete a checklist and attaching test results... to be submitted to the team leader for a decision on what happens next... which becomes added to various people's to-do lists.
- Each stage requires different tools, skills and resources.
- Each stage can be audited separately. For example when something is released that shouldn't have been is it because the actions weren't carried out as specified, or because the wrong actions were decided upon, or because the observation process was faulty? Aha! When the source of the problem can be identified like this it can be fixed for the future.

ODA applies to the *What* and *How* of your programs:

- Where your program is supporting decision making it should endeavour to keep O,D and A separate. Perhaps they go on separate pages of a tabsheet wizard or perhaps you make it easy for one person to do the O and another the D with the A's being logged communicated and chase-up watchdogs¤ set.
- Where your program is interpreting data to make decisions it needs to be clear what is O,D and A. This should be clear in the design documentation but in small cases it might be sufficient to put it in the code.

```
// O1: No login for 2 months (test weekly)
// O2: Have we sent a Ping! email and did we get a reply?
// D:  If O1 but not sent ping then
//       A1a : Send ping email.
//       A1b  Bump watchdog by 3 days.
//     If O1 and no reply to ping then
```

```
//       A2a : Suspend account
//       A2b : Alert administrator[410]
```

If your program is doing complex decision making but giving a dumbed-down traffic-light answer you should still be able to expose your workings in test-mode and you might want to log it's suggested or actual actions against data inputs for auditing or process improvement.

Basic quality implementation principles

With one important exception the only way to operate a quality system is to:

1 determine what you're trying to achieve
2 monitor your performance
3 take a view on what sort of corrective actions are required
4 implement change at the right place
5 repeat.

This is the principle of feedback.

It means you have to have some way of monitoring what's going on which is why there has been so little of it - Somebody has to keep records and be able to take management decisions based on an analysis of the data. Too difficult! - For all sorts of reasons that boil down to lack of organisation and leadership.

The traditional method, which isn't efficient, effective or economical is to use an ever bigger scatter-gun approach of polishing up professional skills and finding more things to tick in more detail. Does anybody think drivers would be better if they had a test every week or had to learn the road traffic acts off by heart? This approach just hopes that something useful will rub off. Things that are difficult to measure are left off the ticklists and going through the motions replaces getting to grips with identified issues.

The important exception to the above is safety where you don't want to wait until you get an accident before doing something about it. Even so the principle of near-miss reporting is a feedback process and for obvious reasons needs to be developed.

Programs to help

Your programs will be involved with all sorts of matters that affect quality. To take a simple example your on-line ordering system is supposed, amongst other things, to *support* shipping the right goods to the right place at the right time. Sometimes there will be problems. Can your system be used to quantify the problems and track back to causes? What can you do to your system to make identified problems less likely - perhaps a double-check, or a bar-code or a total item count for each shipment and so on.

> Thinking like this will revolutionise your approach to program design. You'll find all sorts of little checks going and exception reporting going in. (But see the warning at the top of this appendix.)

Risk

---

410 The bug in this logic should be obvious - That's the point of laying it out clearly.

I briefly touched upon the subject of risk earlier. Risk assessment can be an excuse for extended protective paperwork or sensible appreciation of where risks lie and what to do about those that can't be designed-out of the system.

Probably the most useful thing people can do is increase their risk *awareness*. Consequences may happen far removed from the original mistake, or events may be so rare (or rarely discovered - not quite the same thing) as to be hidden from the people who need to be on their guard. Therefore an *event reporting* system is required with enough hints added for people to recognise their exposure to making the same error.

Finally on this topic there are many activities with low risk (or where errors are easily spotted) which should *not* get the same attention as high risk ones. This seems obvious when written down here but bureaucracies don't work like that. You have been warned!

- Risk is an *estimated statistic*. Estimates may be anything from received wisdom to well specified, up-to date, locally collected and analysed data.
- The *consequences* of risk tell us the importance of getting it right.
- The *rarity* indicates different approaches. An important but rare risk may need awareness to be refreshed from time to time.
- The *immediacy* of risk tells how quickly we need to respond. Practice exercises may be required.
- The *visibility* of risk tells how easy it is to spot a situation.
- The *graduation* of risk indicates how we might get away with a small mistake or a 'this works most of the time' and not lose the whole game. Dealing with a graduated risk like this requires a way to catch the small proportion that we didn't get with our first attempt.
- The *diligence requirement* (of the way we handle) a risk is an extremely important indicator of the sort of training, calibre of staff and checking necessary. High-diligence and high-risk together are to be avoided. Good procedures can reduce the diligence requirement.
- The *cost of prevention* is always something of concern and something that we would like to reduce if possible. Technology, techniques, imagination and alternative methods are things used here. Note: There is little kudos to be obtained from making a problem go away when compared to a heroic battle. It is also very difficult to join-up funds and data between prevention and cure.
- The *counter-risk* is the risk associated with dealing with the original risk. Treatment may have side effects.

Bad - Good - Best

You can't lump all matters relating to let's say 'professional standards' together. Dealing with alcoholics and Masters degrees in the same way is plainly barmy. Hence the Bad-Good-Best model:

- Bad practice is that which we won't countenance and is basically a list of Don't!'s.
- Good practice is the standard we expect from most people all of the time or all people most of the time.
- Best practice is what we would like everyone to aspire to but realistically expect a

few highly motivated people to acquire specialist competencies.

Activators

Bad Complaints and catastrophes
Good Initial training, keeping up with developments, auditing
BestPersonal motivation, group leadership and support

Moderators

Bad Rule book (Contract of employment)
Good Standards documentation, informative articles, team spirit
BestProfessional development programme, research

The important aspects of BGB related to programming are:

- People in the Best category are usually streets ahead in all respects, typically many times more productive and accurate. Also they're capable of dealing with unusual situations and dealing sensibly with risks. Often they have good awareness of their limitations - though what to them is a limitation might be exceptional to an average person.
- Some of the Best people have annoying personal habits and personality defects[411] which need careful handling.
- Software might be used to trap bad practice. For example to alert when somebody hasn't responded to a customer query within a set time. A lot of this comes down to enforcing procedures from simple correct form completion to spotting a mismatch between requests and responses.

Real quality

The expensive way is to bolt-on numerous onerous tasks to everyday procedures. It is doubtful whether this has any positive effect at all.

The real way is to

- Establish quality goals
- Get people to buy-in to the goals or get-out
- Show them how to achieve these gaols and give them the necessary tools
- Accept the cost of doing the job well is inherent (but (with the exception of safety) more effectively spent than on scatter-gun preventive measures.)
- Monitor and manage

Review

There are two ways to implement quality:

- Waffle, paperwork, lip-service
- Establishing quality goals, giving them to people to implement in their daily jobs then monitoring how well they perform.

From a programmer's perspective there are two realms of application

- The quality of their own work

---

411 As do ordinary people, but there's a tendency to 'let them get away with it' which can be exploited to develop to the prima donna stage which can upset middle-of-the-road types.

- Assisting users to make good quality come easily

Likewise BGB is applicable to programming and the environment programs will be run in.

Don't forget that unless Observation, Decision and Action are separated it won't be possible to 'debug' decision making. This applies within a program and the application environment.

H. Two quick management tools

These two back-of-the-envelope tools are very briefly described to give you a framework for collecting and analysing management information. They should help to clarify decision making at the start of projects and give you some pointers to the project monitoring required.

From a real programmer's point of view the important thing is having the concepts ready to hand in your mind so you can use then even without an envelope. You may want to use them formally when trying to get your strategy across or pointing out where there's a lack of coherence in management thinking.

Object-Methods

What are you trying to do? When was the last time you sat down and listed your objectives? By putting these in writing don't you think that will

- clarify your thoughts
- help to prioritise your ambitions
- be an agenda for discussion?

How are you going to do it? If you listed the alternative methods against each objective would that:

- form the basis for discussion
- highlight areas of weak strategy
- eventually become an agreed plan of action?

If you can see the benefits of doing this then divide a page down the middle and write objectives in the left hand column and corresponding methods on the right.

Ref no	Why	Objective	Method	Tactics	Respon-sibility Remarks Restrictions
Optional reference	Optional motivation for wanting to achieve the objective	What is to be achieved	How it might be achieved. - Options - Alternatives	Detailed implementation issues. Optional	Optional notes

There's nothing difficult or startling about this tool. The surprising thing is that it is so rarely used. At this point it is best to have a go on a subject of your choice.

Although in simple O-M back-of-the-envelope form it is a strong tool there's a lot more detail to be exploited by using it in a structured management environment with the bits on the side.

Factors

What makes a project successful?

We all know of projects that fizzled out, were wrongly conceived, suffered too many 'unforseen' circumstances were vaguely managed, and limped home. "If only..." is the cry of hindsight. But managers are paid for their foresight and ability to grasp a situation. If you don't see a problem or a potential ahead of time and don't keep a sufficient eye on its development and fail to make contingency plans and get distracted by less important issues then what do you expect?

FACTORS is a simple back-of-the-envelope method of listing those factors that contribute to the positive or negative outcome of a project. This is ideal personal preparation for project planning meetings, and with a slight refinement, highlights the questions to be asked at progress meetings.

Factors is simply a table of issues, management involvement and preventive or corrective action. This method seeks to stimulate avenues of possibilities and is not a quantitive or in-depth project management tool. Having assisted in focussing foresight and assessing potential impact of factors, it then serves as a reminder of those issues that require surveillance.

Method outline

Take any project.

- What factors are directly down to management to control? For example getting the right people for the job, avoiding the holiday period, ensuring the specification is final before starting. These are called *success factors*. This focusses on the way you drive the project.
- Which factors or events are not under your direct control. For example a supplier gets behind schedule making you late. These *failure factors* are quite probable events that are 'not in the plan' over which you may have some influence.
- The third category of factors are *luck*. You just have to put up with these if your project depends on them, but at the planning stage you may choose a less risky strategy.

Failure and Luck factors show where contingency plans are desirable.

You may have picked up from the above that Success factors are positive, while Failure and Luck factors are negative. *This is not the case*. A positive luck factor is where a fortunate circumstance, say a competitor's blunder, puts you in a position where you can take an advantage - *IF* you can recognise the opportunity and *IF* you happen to be in a position to exploit it. An example of a negative success factor might be where you know there will be repercussions unless you take avoiding action. (It turns out that it is far more important to identify as many factors as possible rather than strictly categorise them - often there are shades of grey between the three types of factor and the sense of whether a factor is positive or negative may be opposite sides of the same coin.)

However it is a useful discipline to decide if each factor is central or peripheral. There are two reasons for doing this. It helps keep discussions focussed on core issues. Secondly, a matter that will appeal to those who are not content with doing a job but must do it well, highlighting where marginal effort can result in significant improvement in overall performance and profit. For example a result that is good enough to win an award will naturally be more attractive to potential purchasers. The effort to get that award may be, for sake of argument, reading the rules of the competition carefully and employing an industrial designer to turn 'just another' into 'the one to beat'. This makes the Factors method suitable for both keeping management aware of essentials and inspire them to get beyond the mediocre.

At a project planning meeting a table of Factors could be brainstormed from scratch, or presented in the form of a skeleton strategy for refinement by the project proposer. It can be used as personal preparation or as the basis for group discussions. Having described the factor and categorised it two more columns in the list are provided. Firstly a 'concern' column. Is there serious risk of this getting out of control, or will a periodic review be sufficient, or is there nothing to worry about. Three simple options. Focussing concern is an important management function, as a manager can hardly deal with everything at once will often cause more confusion and unnecessary work than one who leaves things in capable hands while addressing the few issues. Associated with this is an 'Action' column. What management action is to be taken? For example, suppose as part of the initial planning we'd established that supplier X simply had to deliver their component on time. A prudent manager would recognise this as a trigger to keep a watch on this supplier's progress and probably see this as a prompt to set up a management reporting system. (The 'central' categorisation of the factor also tells the manager that the quality of this information needs to be assured.)

The reason that this tool is so simple is that it concentrates on management strategy and tactics rather than catch-all project management. It is a framework for organising thoughts, an agenda for finalising strategy, a check list for being prepared for mischance and opportunity, and a tool for directing management efforts.

Factors table

As many rows as you like with the following columns

- Ref

  In formal situations it helps to have a reference number
- Factor

  One of: Success, Failure or Luck (Or a graphic)
- Impact

  One of: Major or Peripheral (Or a graphic)

  You can make a rough assessment of the impact of a factor without needing to undertake a formal risk analysis. You may wish to investigate knotty problems in more details as a result of concluding you have a major potential problem.

  Note. From a profitability point of view peripheral factors are quite important.
- Description of influence

  Achievement, event, or issue described briefly. This is not intended to be a complete description, simply a summing-up in a nutshell. The best descriptions contain a definition of something that may happen and how it may be influenced by an event. These tend to fall into two categories:

1 Event - Situation eg. "If the software arrives late our schedule will be delayed"

2 Situation - Action eg. "If we can't use our existing staff we will need to hire more."

Both modes might be combined. For example: "If the software is late, the rest of our schedule will need to be speeded up avoid late delivery penalty clauses."

- Concern

  One of Immediate and urgent, Keep under scrutiny, No worries (Or graphic)
  This shows the degree of concern that this issue is being handled correctly.
  As a manager reviewing a proposal or monitoring progress you may be looking at your Factors listing as preparation for a meeting or to assure yourself that everything is on course. This column is for your reference as a guide to the importance of taking action or preventing a mistake.

- Action

  Management action. Text describing, in a nutshell, what management action is to be taken.