FILES – FROM THE BEGINNING

CARDS.

Once upon a time, when the world was young, there were card files. These were stacks of cards, each about 188mm × 83mm, each of which could be punched with holes in any or all of 80 × 12 positions. (That was the most common card size. Other sizes were proposed from time to time, and used in special circumstances, but never really caught on.) The full capacity was used when intrinsically binary information was to be stored, but in the files which were of interest to most people each of the 80 columns was used to store a single character from some approved (usually rather small) character set. Once you had punched your data of whatever sort onto cards, you bundled the cards up into a nice neat rectangular block, and that was your card file. Large files could extend over many card boxes, but they were nice neat rectangular blocks too, so the image was much the same.

As well as being useful objects in their own right, they formed everyone's image of what a file was really like – so everyone knew that a file was an ordered sequence of records, all the same length. (Notice that the "system Genie" isn't just a modern phenomenon.) It made sense to think of the 134th record in the file; sequential and random access had obvious meanings. Also, records usually had some sort of internal structure : they were subdivided into fields, subfields, and so on, all of which were in some sense related – so one would normally wish to read or write a whole record at a time. A few rather weird establishments used paper tape rather than cards, which must have presented an image much closer to the "stream of bytes" file we shall come to shortly, but the cards were overwhelmingly more popular.

The records were usually the logical groups of data required by the programme; all elements of a record were usually related (they could be different properties of some purchased item – name, cost, how many purchased), while different records could well contain similar data pertaining to other items. They were always the physical units of transfer between external world and programme; you couldn't physically read half a card. It was not uncommon for the size of the physical unit to be inconvenient, in which case one had to define logical records which were either smaller or larger than the physical records. The same picture worked for other devices, too; a disc was composed of corresponding physically defined records called *sectors*, which were commonly bigger than card records. How did you record a card file on the disc ? Obviously, you would copy several cards into one sector – so the sector, the unit of transfer between programme and disc, became a *block* of several records which would be transferred together for convenience. And so the images of files evolved.

Line printers fitted this image too. A line printer, which was the common output device at the time, is a printer which prints a whole line in one operation; typically, you had to build up the line character by character, then send it as a whole to the printer. Once again, each line is a record, and all records are the same length. There is no such thing as a short line; if you don't fill up the end of the line with blank characters, you get whatever was in the printer line last time.

TERMINALS.

But then there came the terminal. Well, that wasn't much of a problem : obviously, the terminal keyboard could be seen as a substitute for the card reader, and the screen was just a rather odd sort of printer, so it fitted in rather well.

The only hiccup that we remember was in Pascal. Pascal was originally designed for traditional files, but was defined always to read one record ahead so that it would know if end-of-file were imminent. (In most other languages, you don't know that you've come to the end of the file until you try a ***read*** *that doesn't work, which leads to messy programming; in Pascal, you can ask at any time whether a file is*

*ended, but that mustn't cause a new record to be read, so the **read** operation must be attempted in advance, the result recorded, and any line which was read saved until it's needed.) So long as the records were punched cards, all prepared before the programme was started, that was easy; but with a terminal, where the records are individual characters, the Pascal rules say that you must always have read the next character before you can deal with the current one. To make sure that will work, the system has to read the first record of every input file when the file is first opened – which is to say, it must read the first character from the keyboard before it starts to execute the programme. So how do you write a programme which begins by asking "Please enter your name : ", and then expects a response ? You have to give the response before reading the question. People devised various more or less ingenious compromises to make Pascal work with terminals, but there was a lot of debate at the time.*

It was true that terminals weren't *quite* the same as card readers. For one thing, they were a lot less easy to use with the fixed-field input specifications ("Enter the item number left justified in columns 1 to 7, the number of items purchased right justified with leading zeros in columns 8 to 12, unit price in cents right justified in columns 13 to 20") required by most programmes at the time. With a card punch, someone would transcribe a few hundred such records from the original paper documents, using a format card to set up sensible tabulating positions and so on for the card punch (which actually made the formatting quite easy); with a terminal, it makes more sense to enter the information, as well as other sorts of information, as it becomes available, so it isn't worth taking time to set up anything like a standard format. One response to this was the development of "forms management packages", which displayed something resembling a form to be filled in on the screen, and interpreted tabulation and return characters to make it easy to fill in the form in a sensible way. (The effect is something like that of the Macintosh dialogue boxes, but the forms management displays are much more sophisticated. It isn't unreasonable to think of it as a "forms metaphor" for the interface.) The forms management package converts the terminal into a device for entering records, so resurrecting the card reader, albeit lightly disguised.

The other response was more revolutionary. It became clear that terminals do not fit at all well into the nice cosy world of the card file. Unless you are artificially constrained by something like a forms management package, you use a terminal in a much less disciplined way. You don't want to have to count columns, particularly when the penalty for making a mistake is a factor of 10^n in your accounts. You begin to think that the software should be able to make sense of the perfectly comprehensible line you entered without requiring any additional help from you. People who use terminals a lot develop an image of a file which is less orderly than the classical pattern : a file becomes a sequence of characters, with little structure of its own. Structure might be imposed on the data by certain punctuation marks (such as newline characters), but these are characters within the file like any others, so the structure is determined by the file contents, not predefined as a property of the file itself; the idea of a higher level of organisation is missing. Even the length of a line is not obviously constant; most, though not all, editors will not place the cursor any further to the right than the last explicitly entered character in the line.

This change in view is particularly obvious in environments where programme development is emphasised over programme use – such as, taking an example at random, university computing departments. It is perhaps encouraged by the coincidence that the other sort of file widely used in such work is the code file, which has commonly no obvious fixed record structure either.

Operating systems developed for use in such places naturally provide the sort of file system which their customers find congenial, so typical microcomputer systems and Unix base their file systems on a stream-of-characters model. In consequence, there are now two "standard" patterns for file systems, and they don't mix together too well. It's true, of course, that you can always manage : if you want a stream-of-characters and your file system gives you records and blocks, you store the characters in 80-character records, or something; and proponents of the simpler stream-of-characters model are always eager to urge that, with their file system, you can create any file structure you want, unhindered by system constraints. (Which is true enough, but – as we have said before, and will quite probably say again – a system which knows a lot about the structure of its files is likely to be able to handle them better for you.)

Why is there such a divergence of views ? Is there no standard definition of a file on which operating system designers can agree ? The quick answer is : yes – there *is* no standard definition of a file. The slow answer is : well, maybe, but it's not very adequate as a definition. A file is usually defined in terms such as "a named collection of data which exists outside the programme". (The very slow answer is that no one takes much notice of the definitions anyway, and in practice a file is anything with which you can use a **read** or **write** instruction. We'll come back to this later, but it's interesting that since our students began to learn programming using Macintosh systems even the idea that keyboards and screens can be thought of as files has rather gone out of fashion, perhaps because all transactions with keyboard and screen are executed using Macintosh system procedures.)

WHICH DEFINITION ?

We find ourselves in a position of some confusion. We have found two rather different views of what a file is – either a collection of data stored on some more or less permanent medium, or anything with which a programme can interact with some transfer of data. We have also found two views of file organisation – either an array of records with well defined structure, or a sequence of characters, structureless unless otherwise indicated by items in the sequence itself. We haven't invented these views; they have evolved as people have written programmes which use files of one sort or another.

Their evolution is itself a significant fact : it implies that all the views are useful in some context. It would therefore be idle for us to suggest that some are right and some wrong; all are right sometimes, and perhaps wrong at other times. Nevertheless, the confusion remains, and if we are to proceed with a comparatively reasonable discussion we should seek some less confused classification. We suggest that the reason for the divergence of views is that all are based on experience in using files, but with no attempt to separate the causes of differences in behaviour. Two particularly important groups of factors which are not identified in the traditional descriptions are those concerned with file storage, and those concerned with file use in programmes, and we shall try to justify this distinction in the remainder of this chapter.

Our aim is to develop a more systematic description of files. Consider the "definition" of files which we presented : "a named collection of data which exists outside the programme". This is much the same as defining a variable as "a named set of bits which lives in memory" – which is all right so far as it goes, but it doesn't go very far. Instead, we go to great trouble to consider what we want to do with variables, what values they should be permitted to assume, and what operations can be performed on them; we call the collection of this information the variable's *type*. The variable has to be kept somewhere, but that's really a secondary consideration. Can we define a file's type ? To do so, we must first determine the properties which should constitute the definition of the type, and how the files are used. We first define what we want to do with them. That's easy, because all the views agree : we want to put data in, and take data out. That's the end of the easy bit.

We therefore continue with an attempt to analyse what's behind the first divergence of views – so here's a very sketchy summary of some of the characteristics of files as they appear under the two competing definitions :

Definition :	STORE	DATA EXTERNAL to the programme
Interesting features :	Named, Permanent	Operations : read (or) write,...
Media :	Discs, Tapes, CD (WORM), …	Terminals, printers, pipes ....
Nature :	STATIC DATA	MOVING DATA

What does this tell us about our attempt to define a file type ? The interesting conclusion – and this is what distinguishes files from other data structures – is that we might want the data to outlive the programme, so we require a new property which we might call **persistence**. In the traditional views, not all files need have this new property – which is where the "very slow answer" comes in. (We shall conform to this view for a little longer in the interests of convenience, but we give warning that we shall soon use the property of persistence to distinguish files from structures of a different sort.) Obvious examples of files which aren't persistent are those connected with terminals. We shall say that a file is either *persistent* or *ephemeral*. We can even use the new property to draw together what we might call the internal and external memory of a programme : we can think of all the conventional memory of a programme as ephemeral "files". (Whether or not that is a useful component of our system illusion is determined by whether or not we find it helpful.)

Next we return to the operations which we want to apply to the files. We ask how we want to perform these operations, and we find there are several different answers. Here we'll consider the classical case, where we distinguish only between *random* and *sequential* file access; you can add other types (such as *indexed*, if you want support for a form of associative memory) if you like. For random access, we want to be able to get at the records of the file in any order at any time; we need something very like a conventional array. For sequential access, we only want to read or write the next record in the file, starting at the beginning; the file is just a sequence of items with the single operation *next*.

The difference between randomness and sequence is real, because it distinguishes between different hardware devices – paper tape and magnetic tape are by their very nature sequential media, while magnetic and optical discs are specifically designed to give something close to random access. So far as the files are concerned, though, randomness and sequence are just two different ways of viewing the *orderedness* of files; both derive from the idea, inherent in the card file view, that you can identify a record by its position in the file. Seen in this way, random and sequential file access (and, for that matter, any other modes you care to define) are just different ways of determining the position of the next record.

Those are persistent files. Do the same operations apply to ephemeral files ? You can think of an ephemeral file as a serial file stretched out in time instead of in space. Just as a device must inspect a certain position in space to select an item from a persistent file, so it must inspect a certain position in time to select an item from an ephemeral file. There is one big difference : the only position accessible to us is now. One property of ephemeral files is immediately clear : you can't have a random access file. If the items are fixed in time (as opposed to waiting until you ask for them), there's another interesting property : if you don't look for them at the right time, then they'll be lost, and gone forever. It appears that in an ephemeral file we can only see one of its elements at any moment, and if we keep on looking that element will be replaced by the next one. While the sequence of items is evident, it makes little sense to speak of the position of a record in the ephemeral file.

We believe that this difference is so significant that it is better to regard the ephemeral sequences as something other than files. Henceforth, with the exception of a few cases where accepted terminology is firmly entrenched, we'll call such an ephemeral

file a *stream*. A television set is a device for displaying a stream of pictures; a videotape recorder can convert the stream into a persistent serial file, and it can be made into something like a random access file on a video disc.

This distinction was made in the programming language PL/I, in which one could use both stream and record files. Although a certain amount of exchange between the types was possible, they were seen as essentially distinct – so while the input and output instructions for the record files were ***read*** *and* ***write****, those for the streams were* ***get*** *and* ***put****.*

We can present the results of our ratiocination in a table :

	PERSISTENT	EPHEMERAL
SEQUENTIAL	sequential file	stream
RANDOM	random access file	?

We see that there are at least two sorts of persistent object. What do we do with them ? We must keep them both in the place we designed for just such purposes – the file store. It's therefore the file system's job to look after them – but, as we saw, many file systems only provide one sort of object, and we have to make do for the other.

On closer inspection, that point of view is less simple than it appears, for the "sequentiality" of a file depends not only on the file itself but on the machinery (hard and soft) through which the file is read and written. We've already remarked that if you have the file data available then you can contrive any sort of access you like with appropriate software. Hardware solutions are also possible – if you really wanted random access to a card file, you could build a card reader which would read cards in either direction through a deck. (There's another way – see the QUESTIONS at the end of the chapter.) Digital Equipment Corporation worked the same trick with their DECtape system, which gave slowish but usable random access to magnetic tape. Nevertheless, the ideas are still useful – and there are some unambiguously sequential entities which can't be used randomly, such as input from terminals or mice.

There's more. Consider the statement above : "For random access … we need something very like a conventional array". Why can't we *have* a conventional array ? Then we could forget about random access file instructions, and just use ordinary array indexing to select items from the file. Presumably we'd have to declare the array appropriately (for example, something like "**persistent array x[a : b] of <type>**", with provision for saying what its external name is), but after that there should be no difficulty. One way to achieve this end is to map the file directly into the virtual address space, so it simply becomes another area in the virtual memory. Why not ? Certainly as virtual address spaces become bigger and bigger, there's no problem of space for any but the biggest files.

In fact, some systems have been implemented in just this way[SUP6], but we've no information on their performance. Indeed, the fairly recent development of file caches, where files are completely or partially copied into memory for more rapid access is blurring the distinction between files and arrays in much the way we are suggesting, and it begins to seem rather silly to retain all the cumbersome file access mechanism when all we need is to calculate an address in memory. Some more recent research[SUP15] has been carried out on the implementation of persistence at the operating systems level, and the authors write, "When persistence is included as the basic abstraction of an operating system, many of the inadequacies of existing operating systems are eliminated and the tasks of an application developer are greatly simplified, resulting in major improvements in program development time and execution efficiency". It's good to see the world falling in line with 340 at last.

Does it matter ? We think it does. The operating system is there so that we can handle our computing needs simply and efficiently, using the available hardware to the best advantage. If the system only knows about one sort of file, it is unlikely to administer another sort with different requirements equally efficiently. In addition, people trying to use such an incomplete system will, as is only natural, tend to conform to the facilities which are provided – so they're that much less likely to choose the type of file which best suits their requirements. They will then either try to bend the unsuitable structure to their will within their programmes (expending much time and effort on things which ought to have come with the operating system, and maybe not doing it too well anyway) or adapt themselves to the inappropriate machinery provided (which is bad in principle, and probably inefficient to boot).

REFERENCE.

SUP6 : A. Bartioli, S.J. Mullender, M. van der Valk : "Wide address spaces – exploring the design space", *Operating Systems Review* **27#1**, 11 (January 1993)

SUP15 : J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, S. Norris : "Operating system support for persistent and recoverable computations", *CommACM* **39#9**, 62-69 (September, 1996).

QUESTIONS.

How much of the information storage capacity of a card was lost by restricting it to recording one character in each column ? Assume that there were 48 different characters – that was a common restriction on many early systems, because that was all you could get out of the printers. Can you think of a *practicable* way to use the space more effectively ?

Why do we state that the record for a terminal is an individual character ?

Can you think of something that could reasonably be described as a random ephemeral object ? (If so, please let us know. The closest we can get is an ordinary stream from which you commonly ignore the next *n* items, with *n* calculated in some way, and that doesn't sound very useful.)

How does an indexed file fit into our discussion ? Does it fit in at all ? If not, what changes are necessary to make it fit ? (An indexed file is sometimes described as *content-addressable*; you identify a record by giving part of its contents, not its record number.)

Is a card file persistent or ephemeral ? Is a card file stored on disc by a spool system persistent or ephemeral ? (We know of one system in which random access to spooled card files was permitted, but we don't regard it as a satisfactory precedent.)
