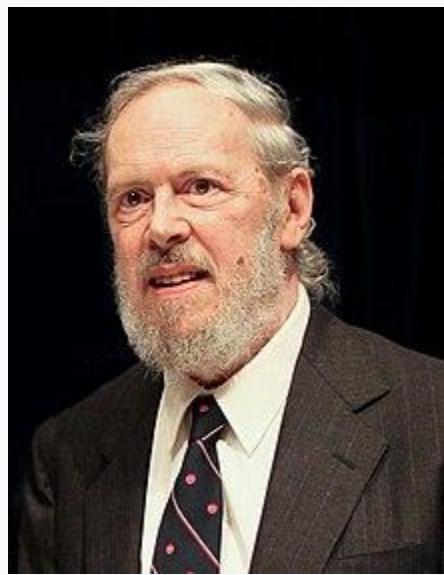


Collected Papers | Dennis Ritchie



Dennis MacAlistair Ritchie (September 9, 1941 – October 12, 2011) was an American computer scientist born to Alistair Ritchie, a switching systems engineer for Bell Laboratories, and Jean McGee Ritchie, a homemaker. He created the C programming language and, with long-time colleague Ken Thompson and is best known for his work on computer languages and operating systems ALTRAN, B, BCPL, C, Multics, and especially Unix.

Dennis Ritchie



Dennis Ritchie at the Japan Prize Foundation in
May 2011

Born September 9, 1941

Bronxville, New York, U.S.

Died c. October 12, 2011 (aged 70)

Berkeley Heights, New Jersey,
U.S.

Nationality American

Alma mater Harvard University (Ph.D., 1968)

Known for ALTRAN

B

BCPL

C

Multics

Unix

Awards [Turing Award](#) (1983)
[National Medal of Technology](#) (1998)
[IEEE Richard W. Hamming Medal](#) (1990)
[Computer Pioneer Award](#) (1994)
[Computer History Museum Fellow](#) (1997)
[Harold Pender Award](#) (2003)
[Japan Prize](#) (2011)

Scientific career

Fields [Computer science](#)

Institutions [Lucent Technologies](#)
[Bell Labs](#)



Ken Thompson (left) and Dennis Ritchie (right)

- [The Development of the C Language](#)
- [The M4 Macro Processor](#)
- [The complexity of loop programs](#)
- [Reflections on Software Research](#)
- [The Evolution of the Unix Time-sharing System](#)
- [A Stream Input Output System](#)
- [The UNIX Time-Sharing System](#)
- [On the Security of UNIX](#)
- [C Reference Manual](#)
- [The Inferno™ Operating System](#)
- [Portability of C Programs and the UNIX System](#)
- [The UNIX Timesharing System - A Retrospective](#)
- [The C Programming Language](#)
- [Unix Programmer's Manual](#)

The Development of the C Language†

Dennis M. Ritchie
Bell Labs/Lucent Technologies
Murray Hill, NJ 07974 USA

dmr@bell-labs.com

ABSTRACT

The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today. This paper studies its evolution.

Introduction

This paper is about the development of the C programming language, the influences on it, and the conditions under which it was created. For the sake of brevity, I omit full descriptions of C itself, its parent B [Johnson 73] and its grandparent BCPL [Richards 79], and instead concentrate on characteristic elements of each language and how they evolved.

C came into being in the years 1969-1973, in parallel with the early development of the Unix operating system; the most creative period occurred during 1972. Another spate of changes peaked between 1977 and 1979, when portability of the Unix system was being demonstrated. In the middle of this second period, the first widely available description of the language appeared: *The C Programming Language*, often called the ‘white book’ or ‘K&R’ [Kernighan 78]. Finally, in the middle 1980s, the language was officially standardized by the ANSI X3J11 committee, which made further changes. Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with Unix; more recently, its use has spread much more widely, and today it is among the languages most commonly used throughout the computer industry.

History: the setting

The late 1960s were a turbulent era for computer systems research at Bell Telephone Laboratories [Ritchie 78] [Ritchie 84]. The company was pulling out of the Multics project [Organick 75], which had started as a joint venture of MIT, General Electric, and Bell Labs; by 1969, Bell Labs management, and even the researchers, came to believe that the promises of Multics could be fulfilled only too late and too expensively. Even before the GE-645 Multics machine was removed from the premises, an informal group, led primarily by Ken Thompson, had begun investigating alternatives.

Thompson wanted to create a comfortable computing environment constructed according to his own design, using whatever means were available. His plans, it is evident in retrospect,

†Copyright 1993 Association for Computing Machinery, Inc. This electronic reprint made available by the author as a courtesy. For further publication rights contact ACM or the author. This article was presented at Second History of Programming Languages conference, Cambridge, Mass., April, 1993.

incorporated many of the innovative aspects of Multics, including an explicit notion of a process as a locus of control, a tree-structured file system, a command interpreter as user-level program, simple representation of text files, and generalized access to devices. They excluded others, such as unified access to memory and to files. At the start, moreover, he and the rest of us deferred another pioneering (though not original) element of Multics, namely writing almost exclusively in a higher-level language. PL/I, the implementation language of Multics, was not much to our tastes, but we were also using other languages, including BCPL, and we regretted losing the advantages of writing programs in a language above the level of assembler, such as ease of writing and clarity of understanding. At the time we did not put much weight on portability; interest in this arose later.

Thompson was faced with a hardware environment cramped and spartan even for the time: the DEC PDP-7 on which he started in 1968 was a machine with 8K 18-bit words of memory and no software useful to him. While wanting to use a higher-level language, he wrote the original Unix system in PDP-7 assembler. At the start, he did not even program on the PDP-7 itself, but instead used a set of macros for the GEMAP assembler on a GE-635 machine. A postprocessor generated a paper tape readable by the PDP-7.

These tapes were carried from the GE machine to the PDP-7 for testing until a primitive Unix kernel, an editor, an assembler, a simple shell (command interpreter), and a few utilities (like the Unix *rm*, *cat*, *cp* commands) were completed. After this point, the operating system was self-supporting: programs could be written and tested without resort to paper tape, and development continued on the PDP-7 itself.

Thompson's PDP-7 assembler outdid even DEC's in simplicity; it evaluated expressions and emitted the corresponding bits. There were no libraries, no loader or link editor: the entire source of a program was presented to the assembler, and the output file—with a fixed name—that emerged was directly executable. (This name, *a.out*, explains a bit of Unix etymology; it is the output of the assembler. Even after the system gained a linker and a means of specifying another name explicitly, it was retained as the default executable result of a compilation.)

Not long after Unix first ran on the PDP-7, in 1969, Doug McIlroy created the new system's first higher-level language: an implementation of McClure's TMG [McClure 65]. TMG is a language for writing compilers (more generally, TransMoGrifiers) in a top-down, recursive-descent style that combines context-free syntax notation with procedural elements. McIlroy and Bob Morris had used TMG to write the early PL/I compiler for Multics.

Challenged by McIlroy's feat in reproducing TMG, Thompson decided that Unix—possibly it had not even been named yet—needed a system programming language. After a rapidly scuttled attempt at Fortran, he created instead a language of his own, which he called B. B can be thought of as C without types; more accurately, it is BCPL squeezed into 8K bytes of memory and filtered through Thompson's brain. Its name most probably represents a contraction of BCPL, though an alternate theory holds that it derives from Bon [Thompson 69], an unrelated language created by Thompson during the Multics days. Bon in turn was named either after his wife Bonnie, or (according to an encyclopedia quotation in its manual), after a religion whose rituals involve the murmuring of magic formulas.

Origins: the languages

BCPL was designed by Martin Richards in the mid-1960s while he was visiting MIT, and was used during the early 1970s for several interesting projects, among them the OS6 operating system at Oxford [Stoy 72], and parts of the seminal Alto work at Xerox PARC [Thacker 79]. We became familiar with it because the MIT CTSS system [Corbato 62] on which Richards worked was used for Multics development. The original BCPL compiler was transported both to Multics and to the GE-635 GECOS system by Rudd Canaday and others at Bell Labs [Canaday 69]; during the final throes of Multics's life at Bell Labs and immediately after, it was the language of choice among the group of people who would later become involved with Unix.

BCPL, B, and C all fit firmly in the traditional procedural family typified by Fortran and

Algol 60. They are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are ‘close to the machine’ in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system. With less success, they also use library procedures to specify interesting control constructs such as coroutines and procedure closures. At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved.

BCPL, B and C differ syntactically in many details, but broadly they are similar. Programs consist of a sequence of global declarations and function (procedure) declarations. Procedures can be nested in BCPL, but may not refer to non-static objects defined in containing procedures. B and C avoid this restriction by imposing a more severe one: no nested procedures at all. Each of the languages (except for earliest versions of B) recognizes separate compilation, and provides a means for including text from named files.

Several syntactic and lexical mechanisms of BCPL are more elegant and regular than those of B and C. For example, BCPL’s procedure and data declarations have a more uniform structure, and it supplies a more complete set of looping constructs. Although BCPL programs are notionally supplied from an undelimited stream of characters, clever rules allow most semicolons to be elided after statements that end on a line boundary. B and C omit this convenience, and end most statements with semicolons. In spite of the differences, most of the statements and operators of BCPL map directly into corresponding B and C.

Some of the structural differences between BCPL and B stemmed from limitations on intermediate memory. For example, BCPL declarations may take the form

```
let P1 be command
and P2 be command
and P3 be command
...

```

where the program text represented by the commands contains whole procedures. The subdeclarations connected by *and* occur simultaneously, so the name P3 is known inside procedure P1. Similarly, BCPL can package a group of declarations and statements into an expression that yields a value, for example

```
E1 := valof $( declarations ; commands ; resultis E2 $) + 1
```

The BCPL compiler readily handled such constructs by storing and analyzing a parsed representation of the entire program in memory before producing output. Storage limitations on the B compiler demanded a one-pass technique in which output was generated as soon as possible, and the syntactic redesign that made this possible was carried forward into C.

Certain less pleasant aspects of BCPL owed to its own technological problems and were consciously avoided in the design of B. For example, BCPL uses a ‘global vector’ mechanism for communicating between separately compiled programs. In this scheme, the programmer explicitly associates the name of each externally visible procedure and data object with a numeric offset in the global vector; the linkage is accomplished in the compiled code by using these numeric offsets. B evaded this inconvenience initially by insisting that the entire program be presented all at once to the compiler. Later implementations of B, and all those of C, use a conventional linker to resolve external names occurring in files compiled separately, instead of placing the burden of assigning offsets on the programmer.

Other fiddles in the transition from BCPL to B were introduced as a matter of taste, and some remain controversial, for example the decision to use the single character = for assignment instead of :=. Similarly, B uses /* */ to enclose comments, where BCPL uses //, to ignore text up to the end of the line. The legacy of PL/I is evident here. (C++ has resurrected the BCPL comment convention.) Fortran influenced the syntax of declarations: B declarations begin with a specifier like auto or static, followed by a list of names, and C not only followed this style

but ornamented it by placing its type keywords at the start of declarations.

Not every difference between the BCPL language documented in Richards's book [Richards 79] and B was deliberate; we started from an earlier version of BCPL [Richards 67]. For example, the `endcase` that escapes from a BCPL `switchon` statement was not present in the language when we learned it in the 1960s, and so the overloading of the `break` keyword to escape from the B and C `switch` statement owes to divergent evolution rather than conscious change.

In contrast to the pervasive syntax variation that occurred during the creation of B, the core semantic content of BCPL—its type structure and expression evaluation rules—remained intact. Both languages are typeless, or rather have a single data type, the ‘word,’ or ‘cell,’ a fixed-length bit pattern. Memory in these languages consists of a linear array of such cells, and the meaning of the contents of a cell depends on the operation applied. The `+` operator, for example, simply adds its operands using the machine’s integer add instruction, and the other arithmetic operations are equally unconscious of the actual meaning of their operands. Because memory is a linear array, it is possible to interpret the value in a cell as an index in this array, and BCPL supplies an operator for this purpose. In the original language it was spelled `rv`, and later `!`, while B uses the unary `*`. Thus, if `p` is a cell containing the index of (or address of, or pointer to) another cell, `*p` refers to the contents of the pointed-to cell, either as a value in an expression or as the target of an assignment.

Because pointers in BCPL and B are merely integer indices in the memory array, arithmetic on them is meaningful: if `p` is the address of a cell, then `p+1` is the address of the next cell. This convention is the basis for the semantics of arrays in both languages. When in BCPL one writes

```
let V = vec 10
```

or in B,

```
auto V[10];
```

the effect is the same: a cell named `V` is allocated, then another group of 10 contiguous cells is set aside, and the memory index of the first of these is placed into `V`. By a general rule, in B the expression

```
*(V+i)
```

adds `V` and `i`, and refers to the `i`-th location after `V`. Both BCPL and B each add special notation to sweeten such array accesses; in B an equivalent expression is

```
V[i]
```

and in BCPL

```
V!i
```

This approach to arrays was unusual even at the time; C would later assimilate it in an even less conventional way.

None of BCPL, B, or C supports character data strongly in the language; each treats strings much like vectors of integers and supplements general rules by a few conventions. In both BCPL and B a string literal denotes the address of a static area initialized with the characters of the string, packed into cells. In BCPL, the first packed byte contains the number of characters in the string; in B, there is no count and strings are terminated by a special character, which B spelled ‘`*e`’. This change was made partially to avoid the limitation on the length of a string caused by holding the count in an 8- or 9-bit slot, and partly because maintaining the count seemed, in our experience, less convenient than using a terminator.

Individual characters in a BCPL string were usually manipulated by spreading the string out into another array, one character per cell, and then repacking it later; B provided corresponding routines, but people more often used other library functions that accessed or replaced individual characters in a string.

More History

After the TMG version of B was working, Thompson rewrote B in itself (a bootstrapping step). During development, he continually struggled against memory limitations: each language addition inflated the compiler so it could barely fit, but each rewrite taking advantage of the feature reduced its size. For example, B introduced generalized assignment operators, using $x=+y$ to add y to x . The notation came from Algol 68 [Wijngaarden 75] via McIlroy, who had incorporated it into his version of TMG. (In B and early C, the operator was spelled $=+$ instead of $+=$; this mistake, repaired in 1976, was induced by a seductively easy way of handling the first form in B's lexical analyzer.)

Thompson went a step further by inventing the $++$ and $--$ operators, which increment or decrement; their prefix or postfix position determines whether the alteration occurs before or after noting the value of the operand. They were not in the earliest versions of B, but appeared along the way. People often guess that they were created to use the auto-increment and auto-decrement address modes provided by the DEC PDP-11 on which C and Unix first became popular. This is historically impossible, since there was no PDP-11 when B was developed. The PDP-7, however, did have a few ‘auto-increment’ memory cells, with the property that an indirect memory reference through them incremented the cell. This feature probably suggested such operators to Thompson; the generalization to make them both prefix and postfix was his own. Indeed, the auto-increment cells were not used directly in implementation of the operators, and a stronger motivation for the innovation was probably his observation that the translation of $++x$ was smaller than that of $x=x+1$.

The B compiler on the PDP-7 did not generate machine instructions, but instead ‘threaded code’ [Bell 72], an interpretive scheme in which the compiler’s output consists of a sequence of addresses of code fragments that perform the elementary operations. The operations typically—in particular for B—act on a simple stack machine.

On the PDP-7 Unix system, only a few things were written in B except B itself, because the machine was too small and too slow to do more than experiment; rewriting the operating system and the utilities wholly into B was too expensive a step to seem feasible. At some point Thompson relieved the address-space crunch by offering a ‘virtual B’ compiler that allowed the interpreted program to occupy more than 8K bytes by paging the code and data within the interpreter, but it was too slow to be practical for the common utilities. Still, some utilities written in B appeared, including an early version of the variable-precision calculator *dc* familiar to Unix users [McIlroy 79]. The most ambitious enterprise I undertook was a genuine cross-compiler that translated B to GE-635 machine instructions, not threaded code. It was a small *tour de force*: a full B compiler, written in its own language and generating code for a 36-bit mainframe, that ran on an 18-bit machine with 4K words of user address space. This project was possible only because of the simplicity of the B language and its run-time system.

Although we entertained occasional thoughts about implementing one of the major languages of the time like Fortran, PL/I, or Algol 68, such a project seemed hopelessly large for our resources: much simpler and smaller tools were called for. All these languages influenced our work, but it was more fun to do things on our own.

By 1970, the Unix project had shown enough promise that we were able to acquire the new DEC PDP-11. The processor was among the first of its line delivered by DEC, and three months passed before its disk arrived. Making B programs run on it using the threaded technique required only writing the code fragments for the operators, and a simple assembler which I coded in B; soon, *dc* became the first interesting program to be tested, before any operating system, on our PDP-11. Almost as rapidly, still waiting for the disk, Thompson recoded the Unix kernel and some basic commands in PDP-11 assembly language. Of the 24K bytes of memory on the machine, the earliest PDP-11 Unix system used 12K bytes for the operating system, a tiny space for user programs, and the remainder as a RAM disk. This version was only for testing, not for real work; the machine marked time by enumerating closed knight’s tours on chess boards of various sizes. Once its disk appeared, we quickly migrated to it after transliterating assembly-language commands to the PDP-11 dialect, and porting those already in B.

By 1971, our miniature computer center was beginning to have users. We all wanted to create interesting software more easily. Using assembler was dreary enough that B, despite its performance problems, had been supplemented by a small library of useful service routines and was being used for more and more new programs. Among the more notable results of this period was Steve Johnson's first version of the *yacc* parser-generator [Johnson 79a].

The Problems of B

The machines on which we first used BCPL and then B were word-addressed, and these languages' single data type, the 'cell,' comfortably equated with the hardware machine word. The advent of the PDP-11 exposed several inadequacies of B's semantic model. First, its character-handling mechanisms, inherited with few changes from BCPL, were clumsy: using library procedures to spread packed strings into individual cells and then repack, or to access and replace individual characters, began to feel awkward, even silly, on a byte-oriented machine.

Second, although the original PDP-11 did not provide for floating-point arithmetic, the manufacturer promised that it would soon be available. Floating-point operations had been added to BCPL in our Multics and GCOS compilers by defining special operators, but the mechanism was possible only because on the relevant machines, a single word was large enough to contain a floating-point number; this was not true on the 16-bit PDP-11.

Finally, the B and BCPL model implied overhead in dealing with pointers: the language rules, by defining a pointer as an index in an array of words, forced pointers to be represented as word indices. Each pointer reference generated a run-time scale conversion from the pointer to the byte address expected by the hardware.

For all these reasons, it seemed that a typing scheme was necessary to cope with characters and byte addressing, and to prepare for the coming floating-point hardware. Other issues, particularly type safety and interface checking, did not seem as important then as they became later.

Aside from the problems with the language itself, the B compiler's threaded-code technique yielded programs so much slower than their assembly-language counterparts that we discounted the possibility of recoding the operating system or its central utilities in B.

In 1971 I began to extend the B language by adding a character type and also rewrote its compiler to generate PDP-11 machine instructions instead of threaded code. Thus the transition from B to C was contemporaneous with the creation of a compiler capable of producing programs fast and small enough to compete with assembly language. I called the slightly-extended language NB, for 'new B.'

Embryonic C

NB existed so briefly that no full description of it was written. It supplied the types `int` and `char`, arrays of them, and pointers to them, declared in a style typified by

```
int i, j;
char c, d;
int iarray[10];
int ipointer[];
char carray[10];
char cpointer[];
```

The semantics of arrays remained exactly as in B and BCPL: the declarations of `iarray` and `carray` create cells dynamically initialized with a value pointing to the first of a sequence of 10 integers and characters respectively. The declarations for `ipointer` and `cpointer` omit the size, to assert that no storage should be allocated automatically. Within procedures, the language's interpretation of the pointers was identical to that of the array variables: a pointer declaration created a cell differing from an array declaration only in that the programmer was expected to assign a referent, instead of letting the compiler allocate the space and initialize the cell.

Values stored in the cells bound to array and pointer names were the machine addresses,

measured in bytes, of the corresponding storage area. Therefore, indirection through a pointer implied no run-time overhead to scale the pointer from word to byte offset. On the other hand, the machine code for array subscripting and pointer arithmetic now depended on the type of the array or the pointer: to compute `iarray[i]` or `ipointer+i` implied scaling the addend `i` by the size of the object referred to.

These semantics represented an easy transition from B, and I experimented with them for some months. Problems became evident when I tried to extend the type notation, especially to add structured (record) types. Structures, it seemed, should map in an intuitive way onto memory in the machine, but in a structure containing an array, there was no good place to stash the pointer containing the base of the array, nor any convenient way to arrange that it be initialized. For example, the directory entries of early Unix systems might be described in C as

```
struct {
    int      inumber;
    char     name[14];
};
```

I wanted the structure not merely to characterize an abstract object but also to describe a collection of bits that might be read from a directory. Where could the compiler hide the pointer to name that the semantics demanded? Even if structures were thought of more abstractly, and the space for pointers could be hidden somehow, how could I handle the technical problem of properly initializing these pointers when allocating a complicated object, perhaps one that specified structures containing arrays containing structures to arbitrary depth?

The solution constituted the crucial jump in the evolutionary chain between typeless BCPL and typed C. It eliminated the materialization of the pointer in storage, and instead caused the creation of the pointer when the array name is mentioned in an expression. The rule, which survives in today's C, is that values of array type are converted, when they appear in expressions, into pointers to the first of the objects making up the array.

This invention enabled most existing B code to continue to work, despite the underlying shift in the language's semantics. The few programs that assigned new values to an array name to adjust its origin—possible in B and BCPL, meaningless in C—were easily repaired. More important, the new language retained a coherent and workable (if unusual) explanation of the semantics of arrays, while opening the way to a more comprehensive type structure.

The second innovation that most clearly distinguishes C from its predecessors is this fuller type structure and especially its expression in the syntax of declarations. NB offered the basic types `int` and `char`, together with arrays of them, and pointers to them, but no further ways of composition. Generalization was required: given an object of any type, it should be possible to describe a new object that gathers several into an array, yields it from a function, or is a pointer to it.

For each object of such a composed type, there was already a way to mention the underlying object: index the array, call the function, use the indirection operator on the pointer. Analogical reasoning led to a declaration syntax for names mirroring that of the expression syntax in which the names typically appear. Thus,

```
int i, *pi, **ppi;
```

declare an integer, a pointer to an integer, a pointer to a pointer to an integer. The syntax of these declarations reflects the observation that `i`, `*pi`, and `**ppi` all yield an `int` type when used in an expression. Similarly,

```
int f(), *f(), (*f)();
```

declare a function returning an integer, a function returning a pointer to an integer, a pointer to a function returning an integer;

```
int *api[10], (*pai)[10];
```

declare an array of pointers to integers, and a pointer to an array of integers. In all these cases the

declaration of a variable resembles its usage in an expression whose type is the one named at the head of the declaration.

The scheme of type composition adopted by C owes considerable debt to Algol 68, although it did not, perhaps, emerge in a form that Algol's adherents would approve of. The central notion I captured from Algol was a type structure based on atomic types (including structures), composed into arrays, pointers (references), and functions (procedures). Algol 68's concept of unions and casts also had an influence that appeared later.

After creating the type system, the associated syntax, and the compiler for the new language, I felt that it deserved a new name; NB seemed insufficiently distinctive. I decided to follow the single-letter style and called it C, leaving open the question whether the name represented a progression through the alphabet or through the letters in BCPL.

Neonatal C

Rapid changes continued after the language had been named, for example the introduction of the `&&` and `||` operators. In BCPL and B, the evaluation of expressions depends on context: within `if` and other conditional statements that compare an expression's value with zero, these languages place a special interpretation on the `and` (`&`) and `or` (`|`) operators. In ordinary contexts, they operate bitwise, but in the `B` statement

```
if (e1 & e2) ...
```

the compiler must evaluate `e1` and if it is non-zero, evaluate `e2`, and if it too is non-zero, elaborate the statement dependent on the `if`. The requirement descends recursively on `&` and `|` operators within `e1` and `e2`. The short-circuit semantics of the Boolean operators in such 'truth-value' context seemed desirable, but the overloading of the operators was difficult to explain and use. At the suggestion of Alan Snyder, I introduced the `&&` and `||` operators to make the mechanism more explicit.

Their tardy introduction explains an infelicity of C's precedence rules. In B one writes

```
if (a==b & c) ...
```

to check whether `a` equals `b` and `c` is non-zero; in such a conditional expression it is better that `&` have lower precedence than `==`. In converting from B to C, one wants to replace `&` by `&&` in such a statement; to make the conversion less painful, we decided to keep the precedence of the `&` operator the same relative to `==`, and merely split the precedence of `&&` slightly from `&`. Today, it seems that it would have been preferable to move the relative precedences of `&` and `==`, and thereby simplify a common C idiom: to test a masked value against another value, one must write

```
if ((a&mask) == b) ...
```

where the inner parentheses are required but easily forgotten.

Many other changes occurred around 1972-3, but the most important was the introduction of the preprocessor, partly at the urging of Alan Snyder [Snyder 74], but also in recognition of the utility of the file-inclusion mechanisms available in BCPL and PL/I. Its original version was exceedingly simple, and provided only included files and simple string replacements: `#include` and `#define` of parameterless macros. Soon thereafter, it was extended, mostly by Mike Lesk and then by John Reiser, to incorporate macros with arguments and conditional compilation. The preprocessor was originally considered an optional adjunct to the language itself. Indeed, for some years, it was not even invoked unless the source program contained a special signal at its beginning. This attitude persisted, and explains both the incomplete integration of the syntax of the preprocessor with the rest of the language and the imprecision of its description in early reference manuals.

Portability

By early 1973, the essentials of modern C were complete. The language and compiler were strong enough to permit us to rewrite the Unix kernel for the PDP-11 in C during the summer of that year. (Thompson had made a brief attempt to produce a system coded in an early version of C—before structures—in 1972, but gave up the effort.) Also during this period, the compiler was retargeted to other nearby machines, particularly the Honeywell 635 and IBM 360/370; because the language could not live in isolation, the prototypes for the modern libraries were developed. In particular, Lesk wrote a ‘portable I/O package’ [Lesk 72] that was later reworked to become the C ‘standard I/O’ routines. In 1978 Brian Kernighan and I published *The C Programming Language* [Kernighan 78]. Although it did not describe some additions that soon became common, this book served as the language reference until a formal standard was adopted more than ten years later. Although we worked closely together on this book, there was a clear division of labor: Kernighan wrote almost all the expository material, while I was responsible for the appendix containing the reference manual and the chapter on interfacing with the Unix system.

During 1973-1980, the language grew a bit: the type structure gained unsigned, long, union, and enumeration types, and structures became nearly first-class objects (lacking only a notation for literals). Equally important developments appeared in its environment and the accompanying technology. Writing the Unix kernel in C had given us enough confidence in the language’s usefulness and efficiency that we began to recode the system’s utilities and tools as well, and then to move the most interesting among them to the other platforms. As described in [Johnson 78a], we discovered that the hardest problems in propagating Unix tools lay not in the interaction of the C language with new hardware, but in adapting to the existing software of other operating systems. Thus Steve Johnson began to work on *pcc*, a C compiler intended to be easy to retarget to new machines [Johnson 78b], while he, Thompson, and I began to move the Unix system itself to the Interdata 8/32 computer.

The language changes during this period, especially around 1977, were largely focused on considerations of portability and type safety, in an effort to cope with the problems we foresaw and observed in moving a considerable body of code to the new Interdata platform. C at that time still manifested strong signs of its typeless origins. Pointers, for example, were barely distinguished from integral memory indices in early language manuals or extant code; the similarity of the arithmetic properties of character pointers and unsigned integers made it hard to resist the temptation to identify them. The *unsigned* types were added to make unsigned arithmetic available without confusing it with pointer manipulation. Similarly, the early language condoned assignments between integers and pointers, but this practice began to be discouraged; a notation for type conversions (called ‘casts’ from the example of Algol 68) was invented to specify type conversions more explicitly. Beguiled by the example of PL/I, early C did not tie structure pointers firmly to the structures they pointed to, and permitted programmers to write `pointer->member` almost without regard to the type of `pointer`; such an expression was taken uncritically as a reference to a region of memory designated by the pointer, while the member name specified only an offset and a type.

Although the first edition of K&R described most of the rules that brought C’s type structure to its present form, many programs written in the older, more relaxed style persisted, and so did compilers that tolerated it. To encourage people to pay more attention to the official language rules, to detect legal but suspicious constructions, and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his *pcc* compiler to produce *lint* [Johnson 79b], which scanned a set of files and remarked on dubious constructions.

Growth in Usage

The success of our portability experiment on the Interdata 8/32 soon led to another by Tom London and John Reiser on the DEC VAX 11/780. This machine became much more popular than the Interdata, and Unix and the C language began to spread rapidly, both within AT&T and outside. Although by the middle 1970s Unix was in use by a variety of projects within the Bell

System as well as a small group of research-oriented industrial, academic, and government organizations outside our company, its real growth began only after portability had been achieved. Of particular note were the System III and System V versions of the system from the emerging Computer Systems division of AT&T, based on work by the company's development and research groups, and the BSD series of releases by the University of California at Berkeley that derived from research organizations in Bell Laboratories.

During the 1980s the use of the C language spread widely, and compilers became available on nearly every machine architecture and operating system; in particular it became popular as a programming tool for personal computers, both for manufacturers of commercial software for these machines, and for end-users interesting in programming. At the start of the decade, nearly every compiler was based on Johnson's *pcc*; by 1985 there were many independently-produced compiler products.

Standardization

By 1982 it was clear that C needed formal standardization. The best approximation to a standard, the first edition of K&R, no longer described the language in actual use; in particular, it mentioned neither the `void` or `enum` types. While it foreshadowed the newer approach to structures, only after it was published did the language support assigning them, passing them to and from functions, and associating the names of members firmly with the structure or union containing them. Although compilers distributed by AT&T incorporated these changes, and most of the purveyors of compilers not based on *pcc* quickly picked up them up, there remained no complete, authoritative description of the language.

The first edition of K&R was also insufficiently precise on many details of the language, and it became increasingly impractical to regard *pcc* as a 'reference compiler,' it did not perfectly embody even the language described by K&R, let alone subsequent extensions. Finally, the incipient use of C in projects subject to commercial and government contract meant that the imprimatur of an official standard was important. Thus (at the urging of M. D. McIlroy), ANSI established the X3J11 committee under the direction of CBEMA in the summer of 1983, with the goal of producing a C standard. X3J11 produced its report [ANSI 89] at the end of 1989, and subsequently this standard was accepted by ISO as ISO/IEC 9899-1990.

From the beginning, the X3J11 committee took a cautious, conservative view of language extensions. Much to my satisfaction, they took seriously their goal: 'to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments.' [ANSI 89] The committee realized that mere promulgation of a standard does not make the world change.

X3J11 introduced only one genuinely important change to the language itself: it incorporated the types of formal arguments in the type signature of a function, using syntax borrowed from C++ [Stroustrup 86]. In the old style, external functions were declared like this:

```
double sin();
```

which says only that `sin` is a function returning a `double` (that is, double-precision floating-point) value. In the new style, this better rendered

```
double sin(double);
```

to make the argument type explicit and thus encourage better type checking and appropriate conversion. Even this addition, though it produced a noticeably better language, caused difficulties. The committee justifiably felt that simply outlawing 'old-style' function definitions and declarations was not feasible, yet also agreed that the new forms were better. The inevitable compromise was as good as it could have been, though the language definition is complicated by permitting both forms, and writers of portable software must contend with compilers not yet brought up to standard.

X3J11 also introduced a host of smaller additions and adjustments, for example, the type

qualifiers `const` and `volatile`, and slightly different type promotion rules. Nevertheless, the standardization process did not change the character of the language. In particular, the C standard did not attempt to specify formally the language semantics, and so there can be dispute over fine points; nevertheless, it successfully accounted for changes in usage since the original description, and is sufficiently precise to base implementations on it.

Thus the core C language escaped nearly unscathed from the standardization process, and the Standard emerged more as a better, careful codification than a new invention. More important changes took place in the language's surroundings: the preprocessor and the library. The preprocessor performs macro substitution, using conventions distinct from the rest of the language. Its interaction with the compiler had never been well-described, and X3J11 attempted to remedy the situation. The result is noticeably better than the explanation in the first edition of K&R; besides being more comprehensive, it provides operations, like token concatenation, previously available only by accidents of implementation.

X3J11 correctly believed that a full and careful description of a standard C library was as important as its work on the language itself. The C language itself does not provide for input-output or any other interaction with the outside world, and thus depends on a set of standard procedures. At the time of publication of K&R, C was thought of mainly as the system programming language of Unix; although we provided examples of library routines intended to be readily transportable to other operating systems, underlying support from Unix was implicitly understood. Thus, the X3J11 committee spent much of its time designing and documenting a set of library routines required to be available in all conforming implementations.

By the rules of the standards process, the current activity of the X3J11 committee is confined to issuing interpretations on the existing standard. However, an informal group originally convened by Rex Jaeschke as NCEG (Numerical C Extensions Group) has been officially accepted as subgroup X3J11.1, and they continue to consider extensions to C. As the name implies, many of these possible extensions are intended to make the language more suitable for numerical use: for example, multi-dimensional arrays whose bounds are dynamically determined, incorporation of facilities for dealing with IEEE arithmetic, and making the language more effective on machines with vector or other advanced architectural features. Not all the possible extensions are specifically numerical; they include a notation for structure literals.

Successors

C and even B have several direct descendants, though they do not rival Pascal in generating progeny. One side branch developed early. When Steve Johnson visited the University of Waterloo on sabbatical in 1972, he brought B with him. It became popular on the Honeywell machines there, and later spawned Eh and Zed (the Canadian answers to ‘what follows B?’). When Johnson returned to Bell Labs in 1973, he was disconcerted to find that the language whose seeds he brought to Canada had evolved back home; even his own *yacc* program had been rewritten in C, by Alan Snyder.

More recent descendants of C proper include Concurrent C [Gehani 89], Objective C [Cox 86], C* [Thinking 90], and especially C++ [Stroustrup 86]. The language is also widely used as an intermediate representation (essentially, as a portable assembly language) for a wide variety of compilers, both for direct descendants like C++, and independent languages like Modula 3 [Nelson 91] and Eiffel [Meyer 88].

Critique

Two ideas are most characteristic of C among languages of its class: the relationship between arrays and pointers, and the way in which declaration syntax mimics expression syntax. They are also among its most frequently criticized features, and often serve as stumbling blocks to the beginner. In both cases, historical accidents or mistakes have exacerbated their difficulty. The most important of these has been the tolerance of C compilers to errors in type. As should be clear from the history above, C evolved from typeless languages. It did not suddenly appear to its earliest users and developers as an entirely new language with its own rules; instead we

continually had to adapt existing programs as the language developed, and make allowance for an existing body of code. (Later, the ANSI X3J11 committee standardizing C would face the same problem.)

Compilers in 1977, and even well after, did not complain about usages such as assigning between integers and pointers or using objects of the wrong type to refer to structure members. Although the language definition presented in the first edition of K&R was reasonably (though not completely) coherent in its treatment of type rules, that book admitted that existing compilers didn't enforce them. Moreover, some rules designed to ease early transitions contributed to later confusion. For example, the empty square brackets in the function declaration

```
int f(a) int a[]; { ... }
```

are a living fossil, a remnant of NB's way of declaring a pointer; *a* is, in this special case only, interpreted in C as a pointer. The notation survived in part for the sake of compatibility, in part under the rationalization that it would allow programmers to communicate to their readers an intent to pass *f* a pointer generated from an array, rather than a reference to a single integer. Unfortunately, it serves as much to confuse the learner as to alert the reader.

In K&R C, supplying arguments of the proper type to a function call was the responsibility of the programmer, and the extant compilers did not check for type agreement. The failure of the original language to include argument types in the type signature of a function was a significant weakness, indeed the one that required the X3J11 committee's boldest and most painful innovation to repair. The early design is explained (if not justified) by my avoidance of technological problems, especially cross-checking between separately-compiled source files, and my incomplete assimilation of the implications of moving between an untyped to a typed language. The *lint* program, mentioned above, tried to alleviate the problem: among its other functions, *lint* checks the consistency and coherency of a whole program by scanning a set of source files, comparing the types of function arguments used in calls with those in their definitions.

An accident of syntax contributed to the perceived complexity of the language. The indirection operator, spelled *** in C, is syntactically a unary prefix operator, just as in BCPL and B. This works well in simple expressions, but in more complex cases, parentheses are required to direct the parsing. For example, to distinguish indirection through the value returned by a function from calling a function designated by a pointer, one writes **fp()* and *(*pf)()* respectively. The style used in expressions carries through to declarations, so the names might be declared

```
int *fp();
int (*pf)();
```

In more ornate but still realistic cases, things become worse:

```
int *(*pfp)();
```

is a pointer to a function returning a pointer to an integer. There are two effects occurring. Most important, C has a relatively rich set of ways of describing types (compared, say, with Pascal). Declarations in languages as expressive as C—Algol 68, for example—describe objects equally hard to understand, simply because the objects themselves are complex. A second effect owes to details of the syntax. Declarations in C must be read in an ‘inside-out’ style that many find difficult to grasp [Anderson 80]. Sethi [Sethi 81] observed that many of the nested declarations and expressions would become simpler if the indirection operator had been taken as a postfix operator instead of prefix, but by then it was too late to change.

In spite of its difficulties, I believe that the C's approach to declarations remains plausible, and am comfortable with it; it is a useful unifying principle.

The other characteristic feature of C, its treatment of arrays, is more suspect on practical grounds, though it also has real virtues. Although the relationship between pointers and arrays is unusual, it can be learned. Moreover, the language shows considerable power to describe important concepts, for example, vectors whose length varies at run time, with only a few basic rules and conventions. In particular, character strings are handled by the same mechanisms as any

other array, plus the convention that a null character terminates a string. It is interesting to compare C's approach with that of two nearly contemporaneous languages, Algol 68 and Pascal [Jensen 74]. Arrays in Algol 68 either have fixed bounds, or are 'flexible.' considerable mechanism is required both in the language definition, and in compilers, to accommodate flexible arrays (and not all compilers fully implement them.) Original Pascal had only fixed-sized arrays and strings, and this proved confining [Kernighan 81]. Later, this was partially fixed, though the resulting language is not yet universally available.

C treats strings as arrays of characters conventionally terminated by a marker. Aside from one special rule about initialization by string literals, the semantics of strings are fully subsumed by more general rules governing all arrays, and as a result the language is simpler to describe and to translate than one incorporating the string as a unique data type. Some costs accrue from its approach: certain string operations are more expensive than in other designs because application code or a library routine must occasionally search for the end of a string, because few built-in operations are available, and because the burden of storage management for strings falls more heavily on the user. Nevertheless, C's approach to strings works well.

On the other hand, C's treatment of arrays in general (not just strings) has unfortunate implications both for optimization and for future extensions. The prevalence of pointers in C programs, whether those declared explicitly or arising from arrays, means that optimizers must be cautious, and must use careful dataflow techniques to achieve good results. Sophisticated compilers can understand what most pointers can possibly change, but some important usages remain difficult to analyze. For example, functions with pointer arguments derived from arrays are hard to compile into efficient code on vector machines, because it is seldom possible to determine that one argument pointer does not overlap data also referred to by another argument, or accessible externally. More fundamentally, the definition of C so specifically describes the semantics of arrays that changes or extensions treating arrays as more primitive objects, and permitting operations on them as wholes, become hard to fit into the existing language. Even extensions to permit the declaration and use of multidimensional arrays whose size is determined dynamically are not entirely straightforward [MacDonald 89] [Ritchie 90], although they would make it much easier to write numerical libraries in C. Thus, C covers the most important uses of strings and arrays arising in practice by a uniform and simple mechanism, but leaves problems for highly efficient implementations and for extensions.

Many smaller infelicities exist in the language and its description besides those discussed above, of course. There are also general criticisms to be lodged that transcend detailed points. Chief among these is that the language and its generally-expected environment provide little help for writing very large systems. The naming structure provides only two main levels, 'external' (visible everywhere) and 'internal' (within a single procedure). An intermediate level of visibility (within a single file of data and procedures) is weakly tied to the language definition. Thus, there is little direct support for modularization, and project designers are forced to create their own conventions.

Similarly, C itself provides two durations of storage: 'automatic' objects that exist while control resides in or below a procedure, and 'static,' existing throughout execution of a program. Off-stack, dynamically-allocated storage is provided only by a library routine and the burden of managing it is placed on the programmer: C is hostile to automatic garbage collection.

Whence Success?

C has become successful to an extent far surpassing any early expectations. What qualities contributed to its widespread use?

Doubtless the success of Unix itself was the most important factor; it made the language available to hundreds of thousands of people. Conversely, of course, Unix's use of C and its consequent portability to a wide variety of machines was important in the system's success. But the language's invasion of other environments suggests more fundamental merits.

Despite some aspects mysterious to the beginner and occasionally even to the adept, C

remains a simple and small language, translatable with simple and small compilers. Its types and operations are well-grounded in those provided by real machines, and for people used to how computers work, learning the idioms for generating time- and space-efficient programs is not difficult. At the same time the language is sufficiently abstracted from machine details that program portability can be achieved.

Equally important, C and its central library support always remained in touch with a real environment. It was not designed in isolation to prove a point, or to serve as an example, but as a tool to write programs that did useful things; it was always meant to interact with a larger operating system, and was regarded as a tool to build larger tools. A parsimonious, pragmatic approach influenced the things that went into C: it covers the essential needs of many programmers, but does not try to supply too much.

Finally, despite the changes that it has undergone since its first published description, which was admittedly informal and incomplete, the actual C language as seen by millions of users using many different compilers has remained remarkably stable and unified compared to those of similarly widespread currency, for example Pascal and Fortran. There are differing dialects of C—most noticeably, those described by the older K&R and the newer Standard C—but on the whole, C has remained freer of proprietary extensions than other languages. Perhaps the most significant extensions are the ‘far’ and ‘near’ pointer qualifications intended to deal with peculiarities of some Intel processors. Although C was not originally designed with portability as a prime goal, it succeeded in expressing programs, even including operating systems, on machines ranging from the smallest personal computers through the mightiest supercomputers.

C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.

Acknowledgments

It is worth summarizing compactly the roles of the direct contributors to today’s C language. Ken Thompson created the B language in 1969-70; it was derived directly from Martin Richards’s BCPL. Dennis Ritchie turned B into C during 1971-73, keeping most of B’s syntax while adding types and many other changes, and writing the first compiler. Ritchie, Alan Snyder, Steven C. Johnson, Michael Lesk, and Thompson contributed language ideas during 1972-1977, and Johnson’s portable compiler remains widely used. During this period, the collection of library routines grew considerably, thanks to these people and many others at Bell Laboratories. In 1978, Brian Kernighan and Ritchie wrote the book that became the language definition for several years. Beginning in 1983, the ANSI X3J11 committee standardized the language. Especially notable in keeping its efforts on track were its officers Jim Brodie, Tom Plum, and P. J. Plauger, and the successive draft redactors, Larry Rosler and Dave Prosser.

I thank Brian Kernighan, Doug McIlroy, Dave Prosser, Peter Nelson, Rob Pike, Ken Thompson, and HOPL’s referees for advice in the preparation of this paper.

References

- [ANSI 89] American National Standards Institute, *American National Standard for Information Systems—Programming Language C*, X3.159-1989.
- [Anderson 80] B. Anderson, ‘Type syntax in the language C: an object lesson in syntactic innovation,’ SIGPLAN Notices **15** (3), March, 1980, pp. 21-27.
- [Bell 72] J. R. Bell, ‘Threaded Code,’ C. ACM **16** (6), pp. 370-372.
- [Canaday 69] R. H. Canaday and D. M. Ritchie, ‘Bell Laboratories BCPL,’ AT&T Bell Laboratories internal memorandum, May, 1969.
- [Corbato 62] F. J. Corbato, M. Merwin-Dagget, R. C. Daley, ‘An Experimental Time-sharing System,’ AFIPS Conf. Proc. SJCC, 1962, pp. 335-344.
- [Cox 86] B. J. Cox and A. J. Novobilski, *Object-Oriented Programming: An Evolutionary*

- [Gehani 89] *Approach*, Addison-Wesley: Reading, Mass., 1986. Second edition, 1991.
- [Jensen 74] N. H. Gehani and W. D. Roome, *Concurrent C*, Silicon Press: Summit, NJ, 1989.
- [Johnson 73] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag: New York, Heidelberg, Berlin. Second Edition, 1974.
- [Johnson 73] S. C. Johnson and B. W. Kernighan, ‘The Programming Language B,’ Comp. Sci. Tech. Report #8, AT&T Bell Laboratories (January 1973).
- [Johnson 78a] S. C. Johnson and D. M. Ritchie, ‘Portability of C Programs and the UNIX System,’ Bell Sys. Tech. J. **57** (6) (part 2), July-Aug, 1978.
- [Johnson 78b] S. C. Johnson, ‘A Portable Compiler: Theory and Practice,’ Proc. 5th ACM POPL Symposium (January 1978).
- [Johnson 79a] S. C. Johnson, ‘Yet another compiler-compiler,’ in *Unix Programmer’s Manual*, Seventh Edition, Vol. 2A, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.
- [Johnson 79b] S. C. Johnson, ‘Lint, a Program Checker,’ in *Unix Programmer’s Manual*, Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.
- [Kernighan 78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988.
- [Kernighan 81] B. W. Kernighan, ‘Why Pascal is not my favorite programming language,’ Comp. Sci. Tech. Rep. #100, AT&T Bell Laboratories, 1981.
- [Lesk 73] M. E. Lesk, ‘A Portable I/O Package,’ AT&T Bell Laboratories internal memorandum ca. 1973.
- [MacDonald 89] T. MacDonald, ‘Arrays of variable length,’ J. C Lang. Trans **1** (3), Dec. 1989, pp. 215-233.
- [McClure 65] R. M. McClure, ‘TMG—A Syntax Directed Compiler,’ Proc. 20th ACM National Conf. (1965), pp. 262-274.
- [McIlroy 60] M. D. McIlroy, ‘Macro Instruction Extensions of Compiler Languages,’ C. ACM **3** (4), pp. 214-220.
- [McIlroy 79] M. D. McIlroy and B. W. Kernighan, eds, *Unix Programmer’s Manual*, Seventh Edition, Vol. I, AT&T Bell Laboratories: Murray Hill, NJ, 1979.
- [Meyer 88] B. Meyer, *Object-oriented Software Construction*, Prentice-Hall: Englewood Cliffs, NJ, 1988.
- [Nelson 91] G. Nelson, *Systems Programming with Modula-3*, Prentice-Hall: Englewood Cliffs, NJ, 1991.
- [Organick 75] E. I. Organick, *The Multics System: An Examination of its Structure*, MIT Press: Cambridge, Mass., 1975.
- [Richards 67] M. Richards, ‘The BCPL Reference Manual,’ MIT Project MAC Memorandum M-352, July 1967.
- [Richards 79] M. Richards and C. Whitbey-Strevens, *BCPL: The Language and its Compiler*, Cambridge Univ. Press: Cambridge, 1979.
- [Ritchie 78] D. M. Ritchie, ‘UNIX: A Retrospective,’ Bell Sys. Tech. J. **57** (6) (part 2), July-Aug, 1978.
- [Ritchie 84] D. M. Ritchie, ‘The Evolution of the UNIX Time-sharing System,’ AT&T Bell Labs. Tech. J. **63** (8) (part 2), Oct. 1984.
- [Ritchie 90] D. M. Ritchie, ‘Variable-size arrays in C,’ J. C Lang. Trans. **2** (2), Sept. 1990, pp. 81-86.
- [Sethi 81] R. Sethi, ‘Uniform syntax for type expressions and declarators,’ Softw. Prac. and Exp. **11** (6), June 1981, pp. 623-628.
- [Snyder 74] A. Snyder, *A Portable Compiler for the Language C*, MIT: Cambridge, Mass., 1974.
- [Stoy 72] J. E. Stoy and C. Strachey, ‘OS6—An experimental operating system for a small computer. Part I: General principles and structure,’ Comp J. **15**, (Aug. 1972), pp.

- 117-124.
- [Stroustrup 86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley: Reading, Mass., 1986. Second edition, 1991.
- [Thacker 79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, D. R. Boggs, ‘Alto: A Personal Computer,’ in *Computer Structures: Principles and Examples*, D. Siewiorek, C. G. Bell, A. Newell, McGraw-Hill: New York, 1982.
- [Thinking 90] *C* Programming Guide*, Thinking Machines Corp.: Cambridge Mass., 1990.
- [Thompson 69] K. Thompson, ‘Bon—an Interactive Language,’ undated AT&T Bell Laboratories internal memorandum (ca. 1969).
- [Wijngaarden 75] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. Koster, M. Sintzoff, C. Lindsey, L. G. Meertens, R. G. Fisker, ‘Revised report on the algorithmic language Algol 68,’ *Acta Informatica* 5, pp. 1-236.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

M4 is a macro processor available on UNIX† and GCOS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

July 1, 1977

†UNIX is a Trademark of Bell Laboratories.

The M4 Macro Processor

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

Usage

On UNIX, use

m4 [files]

Each argument file is processed in order; if there are no arguments, or if an argument is `-', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

m4 [files] >outputfile

On GCOS, usage is identical, but the program is called **./m4**.

Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

define(name, stuff)

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

define(N, 100)

...

if (i > N)

defines **N** to be 100, and uses this ``symbolic constant'' in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by `(', it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by

non-alphanumerics. For example, in

```
define(N, 100)
...
if (NNN > 100)
```

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains a lot of **N**'s.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if **N** is redefined? Or, to say it another way, is **M** defined as **N** or as 100? In M4, the latter is true — **M** is 100, so even if **N** subsequently changes, **M** does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now **M** is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the **M** will be replaced by **N** which will be replaced by 100).

Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ' is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N')
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of

macros. If you want the word **define** to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the **N** in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine **N**, you must delay the evaluation by quoting:

```
define(N, 100)
...
define(`N', 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ' are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote( [, ] )
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N')
```

removes the definition of **N**. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. In particular, M4 has pre-defined the names **unix** and **geos** on the corresponding systems, so you can tell which one you're using:

```
ifdef(`unix', `define(wordsize,16)')
```

ifdef(`gcos', `define(wordsize,36)')

makes a definition appropriate for the particular machine. Don't forget the quotes!
ifdef actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

ifdef(`unix', on UNIX, not on UNIX)

Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **\$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

define(bump, \$1 = \$1 + 1)

generates code to increment its argument by 1:

bump(x)

is

x = x + 1

A macro can have as many arguments as you want, but only the first nine are accessible, through **\$1** to **\$9**. (The macro name itself is **\$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

define(cat, \$1\$2\$3\$4\$5\$6\$7\$8\$9)

Thus

cat(x, y, z)

is equivalent to

xyz

\$4 through **\$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

define(a, b c)

defines **a** to be **b c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma ``protected'' by parentheses does not terminate an argument. That is, in

define(a, (b,c))

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as ``one more than N'', write

**define(N, 100)
define(N1, `incr(N)')**

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

unary + and -
** or ^ (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
! (not)
& or && (logical and)
| or || (logical or)

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like $1>0$) is 1, and false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be 2^{**N+1} . Then

**define(N, 3)
define(M, `eval(2**N+1)')**

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

include(filename)

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include**

(that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (`silent include') says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

divert(n)

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

undivert

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

syscmd(date)

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp*: a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

ifelse(a, b, c, d)

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns ``yes'' or ``no'' if they are the same or different.

define(compare, `ifelse(\$1, \$2, yes, no)')

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

ifelse(a, b, c, d, e, f, g)

if the string **a** matches the string **b**, the result is **c**. Otherwise, if **d** is the same as **e**, the result is **f**. Otherwise the result is **g**. If the final argument is omitted, the result is null, so

ifelse(a, b, c)

is **c** if **a** matches **b**, and null otherwise.

String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

len(abcdef)

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

substr('now is the time', 1)

is

ow is the time

If **i** or **n** are out of range, various sensible things happen.

index(s1, s2) returns the index (position) in **s1** where the string **s2** occurs, or -1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

translit(s, f, t)

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

translit(s, aeiou, 12345)

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have an entry in **t** are deleted; as a limiting case, if **t** is not present at all, characters from **f** are deleted from **s**. So

translit(s, aeiou)

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(-1)
define(...)
...
divert
```

Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

errprint(`fatal error`)

dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

Summary of Built-ins

Each entry is preceded by the page number where it is described.

```
3 changequote(L, R)
1 define(name, replacement)
4 divert(number)
4 divnum
5 dnl
5 dumpdef(`name`, `name`, ...)
5 errprint(s, s, ...)
```

```
4 eval(numeric expression)
3 ifdef(`name', this if true, this if false)
5 ifelse(a, b, c, d)
4 include(file)
3 incr(number)
5 index(s1, s2)
5 len(string)
4 maketemp(...XXXXX...)
4 sinclude(file)
5 substr(string, position, number)
4 syscmd(s)
5 translit(str, from, to)
3 undefine(`name')
4 undivert(number,number,...)
```

Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

References

[1]B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Inc., 1976.

The complexity of loop programs*

by ALBERT R. MEYER

*IBM Watson Research Center
Yorktown Heights, New York*

and

DENNIS M. RITCHIE

*Harvard University
Cambridge, Massachusetts*

INTRODUCTION

Anyone familiar with the theory of computability will be aware that practical conclusions from the theory must be drawn with caution. If a problem can theoretically be solved by computation, this does not mean that it is practical to do so. Conversely, if a problem is formally undecidable, this does not mean that the subcases of primary interest are impervious to solution by algorithmic methods.

For example, consider the problem of improving assembly code. Compilers for languages like FORTRAN and MAD typically check the code of an assembled program for obvious inefficiencies—say, two “clear and add” instructions in a row—and then produce edited programs which are shorter and faster than the original. From the theory of computability one can conclude quite properly that no code improving algorithm can work all the time. There is always a program which can be improved in ways that no particular improvement algorithm can detect, so no such algorithm can be perfect. But the non-existence of a perfect algorithm is not much of an obstacle in the practical problem of finding an algorithm to improve large classes of common programs.

The question of detecting improvable programs will appear again later in this paper, but our main concern will be with a related question: can one look at a program and determine an upper bound on its running time? Again, a fundamental theorem in the theory of

computability implies that this cannot be done.* The theorem does *not* imply that one cannot bound the running time of broad categories of interesting programs, including programs capable of computing all the arithmetic functions one is likely to encounter outside the theory of computability itself.

In the next section we describe such a class of programs, called “Loop programs.” Each Loop program consists only of assignment statements and iteration (loop) statements, the latter resembling the DO statement of FORTRAN, and special cases of the FOR and THROUGH statements of ALGOL and MAD. The bound on the running time of a Loop program is determined essentially by the length of the program and the depth of nesting of its loops.

Although Loop programs cannot compute all the computable functions, they can compute all the primitive recursive functions. The functions computable by Loop programs are, in fact, precisely the primitive recursive functions. Several of our results can be regarded as an attempt to make precise the notion that the complexity of a primitive recursive function is apparent from its definition or program. This property is one of the reasons that the primitive recursive functions are used throughout the theory of computability, for, as we remarked in the opening paragraph, knowing that a function is computable is

*Roughly speaking, the undecidability of the halting problem for Turing machines implies that if a programming language is powerful enough to describe arbitrarily complex computations, then the language must inevitably be powerful enough to describe infinite computations. Furthermore, descriptions of finite and infinite computations are generally indistinguishable, so there is certainly no way to choose for each program a function which bounds its running time.

*This research was supported in part by NSF GP-2880 under Professor P. C. Fischer, by the Division of Engineering and Applied Physics, Harvard University, and by the IBM Watson Research Center.

not very useful unless one can tell how difficult the function is to compute. A bound on the running time of a Loop program provides a rough estimate of the degree of difficulty of the computation defined by the program.

Loop programs are so powerful that our bounds on running time cannot be of practical value—for functions computable by Loop programs are almost wholly beyond the computational capacity of any real device. Nevertheless they provide a good illustration of the theoretical issues involved in estimating the running time of programs, and we believe that readers with a practical orientation may find some of the results provocative.

Loop programs

A Loop program is a finite sequence of instructions for changing non-negative integers stored in registers. There is no limit to the size of an integer which may be stored in a register, nor any limit to the number of registers to which a program may refer, although any given program will refer to only a fixed number of registers.

Instructions are of five types: (1) $X = Y$, (2) $X = X + 1$, (3) $X = 0$, (4) LOOP X , (5) END, where "X" and "Y" may be replaced by any names for registers.

The first three types of instructions have the same interpretation as in several common languages for programming digital computers. " $X = Y$ " means that the integer contained in Y is to be copied into X ; previous contents of X disappear, but the contents of Y remain unchanged. " $X = X + 1$ " means that the integer in X is to be incremented by one. " $X = 0$ " means that the contents of X are to be set to zero. These are the only instructions which affect the registers.

A sequence of instructions is a Loop program providing that type (4) and type (5) instructions are matched like left and right parentheses. The instructions in a Loop program are normally executed sequentially in the order in which they occur in the program. Type (4) and (5) instructions affect the normal order by indicating that a block of instructions is to be repeated. Specifically if P is a Loop program, and the integer in X is x, then "LOOP X, P, END" means that P is to be performed x times in succession before the next instruction, if any, after the END is executed; changes in the contents of X while P is being repeated do not alter the number of times P is to be repeated. The final clause is needed to ensure that executions of Loop programs always terminate. For example, the program

LOOP X
X=X+1
END

is a program for doubling the contents of X, rather than an infinite loop. Note that when X initially contains zero, the second instruction is not executed.

Since LOOP's and END's appear in pairs like left and right parentheses, the block of instructions affected by a LOOP instruction is itself a Loop program and is uniquely determined by the matching END instruction. L_n is defined as the class of programs with LOOP-END pairs nested to a depth of at most n ; depth zero means the program has no LOOP's.

For example, (2.2) is an L₂ program in which type (4) and (5) instructions are paired as indicated by the indentations.

```

LOOP Y          (2.2)
  A=0
  LOOP X
    X=A
    A=A+1
  END
END

```

If X and Y initially contain x and y, execution of (2.2) would leave $x \div y$ in X, where $x \div y$ equals $x - y$ if $x \geq y$, and is zero otherwise.

We say that a Loop program computes a function as soon as some of the registers are designated for input and output. If f is a function (from non-negative integers into non-negative integers) of m -variables, then a Loop program P with input registers X_1, \dots, X_m and output register F computes f providing that, when registers X_1, \dots, X_m initially contain integers x_1, \dots, x_m and all other registers initially contain zero, then the integer left in F after P has been executed is $f(x_1, \dots, x_m)$. Thus, program (2.1) with X serving as both input and output register computes the function $2x$. For each $n \geq 0$, ℓ_n is defined as the set of functions computable by a program in L_n .

Primitive recursive functions

Loop programs are extremely powerful despite their simple definition. Addition, multiplication, exponentiation, the x^{th} digit in the decimal expansion of $\sin(\frac{1}{2})$, the x^{th} prime number, etc., are all functions computable by Loop programs. In fact, a fairly careful analysis of the definition of Loop programs is required in order to discover a function which they cannot compute.

These properties are well-known for the class of computable functions known as the *primitive recursive functions*. They apply to Loop programs as well by Theorem 1.

Theorem 1. Every primitive recursive function is computed by some Loop program.

Thorough treatments of the primitive recursive functions can be found in many elementary texts on logic and computability.^{1,2} For the reader's convenience we provide a definition of the primitive recursive functions. It is fairly easy to translate a definition by primitive recursions into a Loop program, and thereby prove Theorem 1.

Definition 1. The primitive recursive functions are the smallest class of functions \mathcal{P} satisfying

- (1) the functions $s(x) = x + 1$, $p_{12}(x,y) =$ are in \mathcal{P}
- (2) \mathcal{P} is closed under the operations of *substitution*: substituting constants, permuting variables (obtaining the function h from f where $h(x,y) = f(y,x)$), identifying variables (obtaining $h(x) = f(x,x)$ from $f(y,z)$), and composing functions.
- (3) \mathcal{P} is closed under the scheme of primitive recursion: if $g,h \in \mathcal{P}$, then $f \in \mathcal{P}$ where f is defined as $f(x_1, \dots, x_m, 0) = g(x_1, \dots, x_m)$

$$f(x_1, \dots, x_m, y+1) = h(x_1, \dots, x_m, y, f(x_1, \dots, x_m, y)).$$

Bounding the running time

Given a Loop program and the integers initially in its registers, the running time of a program is defined as the number of individual instruction executions required to execute the entire program. Thus each Loop program, P , has an associated running time, T_p , which is a function of as many variables as there are registers in P .

Hopefully Section 2 makes it obvious how to count the number of individual executions of type (1), (2), and (3) instructions in any particular computation. The number of executions of LOOP and END instructions can be counted in several ways, but the simplest course is to ignore them. For example, the running time function of an L_0 program (a program with no loops) is a constant function equal to the length of the program; the running time function of program (2.1) is the identity function $f(x) = x$.

This definition of running time has the advantage that it leads to a trivial proof of

Theorem 2. If P is a Loop program in L_n , then the running time function T_p is in L_n .

The proof consists of the observation that if P' is the program obtained by inserting the instruction " $T = T + 1$ " after each type (1), (2) and (3) instruction on P , then P' , computes T_p when all registers of P' except for T are designated as input registers, and T is designated as output register. We assume that " T " itself does not already occur in P . Clearly if P is in L_n , then so is P' , and therefore $T_p \in L_n$.

Now this argument depends heavily on our conventions for measuring running time, and these con-

ventions may seem arbitrary. Actually, Theorem 2 remains true under a wide variety of definitions of running time, as does the following

Corollary. If a Loop program computes a function f , then f is totally defined and is effectively computable.

This amounts to saying that computations defined by Loop programs are always finite no matter what the initial contents of the registers may be, i.e., Loop programs always halt.

As a result of this theorem and corollary, the claim that the running time of Loop programs can be bounded *a priori* by inspection of the program becomes trivial. After all, T_p certainly bounds the running time since by definition it is the running time, T_p is totally defined and effectively computable, and moreover given P one can effectively describe how to compute T_p . Therefore, one can bound the running time of P by T_p .

Of course, bounding P by T_p gives absolutely no new information. It amounts to "predicting" that the program P will run as long as it runs, which is not much of an answer to the question, "How long does my program run?"

A proper answer should be in terms of familiar bounding functions whose properties are simple and understandable. For example, the running time of a typical program computing $f(x) =$ "the x^{th} digit in the decimal expansion of $\sqrt{2}$ " is bounded by x^2 ; for any context-free language there is a recognition algorithm whose running time is bounded by a constant times the cube of the length of an input word; for context-sensitive languages the bound is an exponential of an exponential of the length of an input word, etc.

The functions we use to bound the running time of Loop programs are given in

Definition 2. For a function g of one variable, let $g^{(y)}(z) = g(g(\dots g(z) \dots))$, the composition being taken y times. By convention, $g^{(0)}(z) = z$. For $n \geq 0$, the functions f_n are defined by:

$$f_0(x) = \begin{cases} x + 1 & \text{if } x = 0, 1 \\ x + 2 & \text{if } x \geq 2, \end{cases}$$

$$f_{n+1}(x) = f_n^{(x)}(1).$$

We will say that a function f of one variable *bounds* a function g of several variables if $g(x_1, \dots, x_m) \leq f(\max\{x_1, \dots, x_m\})$ for all integers x_1, \dots, x_m ; $\max\{x_1, \dots, x_m\}$ is the largest member of $\{x_1, \dots, x_m\}$.

Theorem 3. Bounding Theorem. If P is in L_n , then T_p is bounded by $f_n^{(k)}$ where k is the length (number of instructions) of P .

Theorem 3 at least provides a particular class of bounding functions which are reasonably simple and

well-behaved. The first few functions f_n are familiar, viz., $f_1(x) = 2x$ for $x > 0$, and $f_2(x) = 2^x$. The function f_3 is also easy to describe:

$$f_3(x) = 2^2 \cdot \dots \left\{ \begin{array}{l} \text{height } x. \end{array} \right.$$

Furthermore, $f_{n^{(k)}}(x)$ is strictly increasing in n , k , and x whenever $x \geq 2$, and in fact f_{n+1} grows more rapidly than $f_n^{(k)}$ for any fixed value of k . The latter property implies that f_{n+1} majorizes $f_n^{(k)}$ for any fixed values of n and k , i.e., $f_{n+1}(x) > f_n^{(k)}(x)$ for all values of x larger than some bound which depends on n and k .

The definition of f_{n+1} from f_n is by a special case of primitive recursion. Translating this definition into a Loop program, it is easy to show that for $n \geq 1$, f_n is in ℓ_n . The majorizing property and the fact that the running time of a program for any function $g(x)$ is at least $g(x) - x$ (it requires this many steps just to leave the answer in the output register), may be combined to prove

Lemma. For all $n \geq 0$, $f_{n+1} \in \ell_{n+1} - \ell_n$.

Theorem 4. For all $n, k \geq 0$, there are functions in ℓ_{n+1} , which cannot be computed by any Loop program whose running time is bounded by $f_n^{(k)}$.

One can still question whether a bound on running time of f_7 , for example, is any better than no bound at all. In practice it probably is not, since the underlying practical question is always whether a program runs in a reasonable amount of time on a reasonable finite domain of inputs. If a program's running time could not be bounded by a function smaller than f_7 , the program might just as well contain an infinite loop. In both cases, computation for inputs larger than two would not terminate during the lifetime of the Milky Way galaxy. Of course similar objections can be raised against a bound of x^7 , or even $10^7 \cdot x$.

Complexity classes

The sequence $\ell_0, \ell_1, \ell_2, \dots$ is, by Theorem 4, a strictly increasing sequence of sets which provide an infinite number of categories for classifying functions. Although the sets ℓ_n were defined syntactically, by depth of loops, the Bounding Theorem implies that classification by position in the sequence relates to classification by running time or computational complexity. If we define complexity classes C_0, C_1, C_2, \dots by letting C_n be precisely the functions computable by programs with running time bounded by $f_n^{(k)}$ for any fixed $k \geq 0$, then the results of Section 4 imply that $C_n \supset \ell_n$, and $f_{n+1} \in \ell_{n+1} - C_n$.

The fact that $C_n = \ell_n$ is still something of a surprise. *Theorem 5.* For $n \geq 2$, a function is in ℓ_n if and only if it can be computed by a Loop program whose running time is bounded by $f_n^{(k)}$ for some k .

The "only if" part of the theorem is simply the Bounding Theorem. The reverse implication can be proved by showing that if the running time of a program P is bounded by $f_n^{(k)}$, then regardless of the actual depth of loops, P can be rewritten as a program with loops nested to depth n (providing $n \geq 2$).

The rewriting of P proceeds roughly as follows: if P is a sequence of instructions I_1, I_2, \dots, I_k , then a sequence of L_n program I_1, I_2, \dots, I_k is constructed. Each program I_i has a flag which it tests. If its flag is off, I_i has no effect. If its flag is on, I_i has the same effect on the registers of P as execution of the instruction I_i , and I_i also sets a flag for the next instruction which would be executed in a computation of P . Let M be the program "LOOP T, I_1, I_2, \dots, I_k , END" where T is a new register name. Given the integers x_1, \dots, x_m initially in the registers of P , if an integer larger than $T_p(x_1, \dots, x_m)$ is initially in register T , then M will simulate the computation of P .

Since T_p is bounded by $f_n^{(k)}$, one need only construct an L_n program M_n which leaves $f_n^{(k)}$ in register T . Then M_n followed by the L_2 program M together form a program in L_n which computes the same functions as P . This composite program is the "rewritten" version of P .

Theorem 5 would be trivial if every program not in L_n took more time than f_n , for in that case rewriting would never be possible. So far we have shown that some programs in L_n do run as long as f_n , but an obvious flaw in our Bounding Theorem remains: there are programs in L_n with running times which can be bounded by functions smaller than f_n . For example, a program of the form

T=0
LOOP T
P
END (5.1)

has running time equal to one for all inputs and all possible programs P . Hence, our procedure for bounding running time can certainly be improved somewhat. This naturally suggests the question: can one tell if a program runs more rapidly than its loop structure indicates? In general, the answer is no. *Definition 3.* The complexity problem for L_n is: given a program P in L_n determine whether T_p is bounded by $f_{n-1}^{(k)}$ for any integer k .

Theorem 6. For each $n \geq 3$, the complexity problem for L_n is effectively undecidable.

One of the implications of Theorem 6 is that any procedure for bounding the running time of Loop programs can be supplemented to cover special cases of the sort illustrated by (5.1). There cannot be a perfect bounding procedure. Yet no matter how many special cases are treated, there remain an infinite num-

ber of cases for which any procedure must reduce to essentially the one given by the Bounding Theorem. It is in this, admittedly weak, sense that we claim the Bounding Theorem is best possible.

If an "improvement" of the code of a program is defined as a reduction in depth of loops or number of instructions (without an increase in running time), then the proof of Theorem 6 also reveals that there can be no perfect code improvement algorithm. Thus the code improvement problem, which we noted in the introduction was undecidable for programs in general, is still undecidable for the more restricted class of Loop programs.

Primitive recursive classes.

Loop programs were devised specifically as a programming language for primitive recursive functions, so of course we have: *Theorem 7.* The functions computable by Loop programs are precisely the primitive recursive functions.

Half of Theorem 7 already follows from Theorem 1, and the other half can be proved in much the same way as Theorem 1. Parallel to the definition of L_n , one can define* \mathcal{P}_n as the set of functions definable using primitive recursions nested to depth at most n . The computation defined by a LOOP-END pair is slightly more general than that defined by a single primitive recursion, so that $L_1 \supsetneq \mathcal{P}_1$ and $L_2 \supsetneq \mathcal{P}_2$ ($\mathcal{P}_0 = L_0$ by definition), but for $n \geq 4$, $\mathcal{P}_n = L_n$. The simplest proof of this fact, however, seems to depend on Theorem 5 rather than the translation of Loop programs into primitive recursive definitions used in Theorem 7.

Another sequence $\xi^0, \xi^1, \xi^2, \dots$ of sets of primitive recursive functions has been defined by Grzegorczyk⁴ using closure properties rather than programs. For example, ξ^3 is the class of *elementary functions*

defined as the closure of the function $x \dot{-} y$ under substitution and functional operations which transform $f(x,y)$ into $\sum_{i=0}^y f(x,i)$ or into $\prod_{i=0}^y f(x,i)$. Our class L_2 equals the elementary functions; in fact for $n \geq 2, \xi^{n+1} = L_n$.

The proofs of most of the theorem in this paper appear in [5]. The Axt and Grzegorczyk classes are treated in detail in [6].

REFERENCES

- 1 M DAVIS
Computability and unsolvability
McGraw-Hill Book Company Inc New York 1958
- 2 S C KLEENE
Introduction to metamathematics
Van Nostrand New York 1952
- 3 P AXT
Iteration of primitive recursion
Abstract 597-182 Notices Amer Math Soc Jan. 1963
- 4 A GRZEGORCZYK
Some classes of recursive functions
Rozprawy Matematyczne, Warsaw, 1953
- 5 A R MEYER and D M RITCHIE
Computational Complexity and Program Structure
IBM Research Research Report, RC-1817.
- 6 A R MEYER and D M RITCHIE
Hierarchies of primitive recursive functions, in preparation
- 7 A COBHAM
The intrinsic computational difficulty of functions
Proc of the 1964 Cong for Logic Meth and Phil of Science
North-Holland, Amsterdam 1964
- 8 R W RITCHIE
Classes of predictably computable functions
Trans Amer Math Soc Vol 106 Jan 1963 pp 139-173.

*The definition is due to Axt³.

Reflections on Software Research

Can the circumstances that existed in Bell Labs that nurtured the UNIX project be produced again?

DENNIS M. RITCHIE

The UNIX¹ operating system has suddenly become news, but it is not new. It began in 1969 when Ken Thompson discovered a little-used PDP-7 computer and set out to fashion a computing environment that he liked. His work soon attracted me; I joined in the enterprise, though most of the ideas, and most of the work for that matter, were his. Before long, others from our group in the research area of AT&T Bell Laboratories were using the system; Joe Ossanna, Doug McIlroy, and Bob Morris were especially enthusiastic critics and contributors. In 1971, we acquired a PDP-11, and by the end of that year we were supporting our first real users: three typists entering patent applications. In 1973, the system was rewritten in the C language, and in that year, too, it was first described publicly at the Operating Systems Principles conference; the resulting paper [8] appeared in *Communications of the ACM* the next year.

Thereafter, its use grew steadily, both inside and outside of Bell Laboratories. A development group was established to support projects inside the company, and several research versions were licensed for outside use.

The last research distribution was the seventh edition system, which appeared in 1979; more recently, AT&T began to market System III, and now offers System V, both products of the development group. All research versions were "as is," unsupported software;

System V is a supported product on several different hardware lines, most recently including the 3B systems designed and built by AT&T.

UNIX is in wide use, and is now even spoken of as a possible industry standard. How did it come to succeed?

There are, of course, its technical merits. Because the system and its history have been discussed at some length in the literature [6, 7, 11], I will not talk about these qualities except for one: despite its frequent surface inconsistency, so colorfully annotated by Don Norman in his *Datamation* article [4] and despite its richness, UNIX is a simple, coherent system that pushes a few good ideas and models to the limit. It is this aspect of the system, above all, that endears it to its adherents.

Beyond technical considerations, there were sociological forces that contributed to its success. First, it appeared at a time when alternatives to large, centrally administered computation centers were becoming possible; the 1970s were the decade of the minicomputer. Small groups could set up their own computation facilities. Because they were starting afresh, and because manufacturers' software was, at best, unimaginative and often horrible, some adventuresome people were willing to take a chance on a new and intriguing, even though unsupported, operating system.

Second, UNIX was first available on the PDP-11, one of the most successful of the new minicomputers that appeared in the 1970s, and soon its portability brought

¹UNIX is a trademark of AT&T Bell Laboratories.

© 1984 ACM 0001-0782/84/0800-0758 75¢

it to many new machines as they appeared. At the time that UNIX was created, we were pushing hard for a machine, either a DEC PDP-10 or SDS (later Xerox) Sigma 7. It is certain, in retrospect, that if we had succeeded in acquiring such a machine, UNIX might have been written but would have withered away. Similarly, UNIX owes much to Multics [5], as I have described [6, 7] it eclipsed its parent as much because it does not demand unusual hardware support as because of any other qualities.

Finally, UNIX enjoyed an unusually long gestation period. During much of this time (say 1969–1979), the system was effectively under the control of its designers and being used by them. It took time to develop all of the ideas and software, but even though the system was still being developed people were using it, both inside Bell Labs, and outside under license. Thus, we managed to keep the central ideas in hand, while accumulating a base of enthusiastic, technically competent users who contributed ideas and programs in a calm, communicative, and noncompetitive environment. Some outside contributions were substantial, for example those from the University of California at Berkeley. Our users were widely, though thinly, distributed within the company, at universities, and at some commercial and government organizations. The system became important in the intellectual, if not yet commercial, marketplace because of this network of early users.

What does industrial computer science research consist of? Some people have the impression that the original UNIX work was a bootleg project, a “skunk works.” This is not so. Research workers are supposed to discover or invent new things, and although in the early days we subsisted on meager hardware, we always had management encouragement. At the same time, it was certainly nothing like a development project. Our intent was to create a pleasant computing environment for ourselves, and our hope was that others liked it. The Computing Science Research Center at Bell Laboratories to which Thompson and I belong studies three broad areas: theory; numerical analysis; and systems, languages, and software. Although work for its own sake resulting, for example, in a paper in a learned journal, is not only tolerated but welcomed, there is strong though wonderfully subtle pressure to think about problems somehow relevant to our corporation. This has been so since I joined Bell Labs around 15 years ago, and it should not be surprising; the old Bell System may have seemed a sheltered monopoly, but research has always had to pay its way. Indeed, researchers love to find problems to work on; one of the advantages of doing research in a large company is the enormous range of the puzzles that turn up. For example, theorists may contribute to compiler design, or to LSI algorithms; numerical analysts study charge and current distribution in semiconductors; and, of course, software types like to design systems and write programs that people use. Thus, computer research at Bell Labs has always had a considerable commitment to the world, and does not fear edicts commanding us to be practical.

For some of us, in fact, a principal frustration has been the inability to convince others that our research products can indeed be useful. Someone may invent a new application, write an illustrative program, and put it to use in our own lab. Many such demonstrations require further development and continuing support in order for the company to make best use of them. In the past, this use would have been exclusively inside the Bell System; more recently, there is the possibility of developing a product for direct sale.

For example, some years ago Mike Lesk developed an automated directory-assistance system [3]. The program had an online Bell Labs phone book, and was connected to a voice synthesizer on a telephone line with a tone decoder. One dialed the system, and keyed in a name and location code on the telephone’s key pad; it spoke back the person’s telephone number and office address (It didn’t attempt to pronounce the name). In spite of the hashing through twelve buttons (which, for example, squashed “A,” “B,” and “C” together), it was acceptably accurate: it had to give up on around 5 percent of the tries. The program was a local hit and well-used. Unfortunately, we couldn’t find anyone to take it over, even as a supported service within the company, let alone a public offering, and it was an excessive drain on our resources, so it was finally scrapped. (I chose this example not only because it is old enough not to exacerbate any current squabbles, but also because it is timely: The organization that publishes the company telephone directory recently asked us whether the system could be revived.)

Of course not every idea is worth developing or supporting. In any event, the world is changing: Our ideas and advice are being sought much more avidly than before. This increase in influence has been going on for several years, partly because of the success of UNIX, but, more recently, because of the dramatic alteration of the structure of our company.

AT&T divested its telephone operating companies at the beginning of 1984. There has been considerable public speculation about what this will mean for fundamental research at Bell Laboratories; one report in *Science* [2] is typical. One fear sometimes expressed is that basic research, in general, may languish because it yields insufficient short-term gains to the new, smaller AT&T. The public position of the company is reassuring; moreover, research management at Bell Labs seems to believe deeply, and argues persuasively, that the commitment to support of basic research is deep and will continue [1].

Fundamental research at Bell Labs in physics and chemistry and mathematics may, indeed, not be threatened; nevertheless, the danger it might face, and the case against which it must be prepared to argue, is that of irrelevance to the goals of the company. Computer science research is different from these more traditional disciplines. Philosophically it differs from the physical sciences because it seeks not to discover, explain, or exploit the natural world, but instead to study the properties of machines of human creation. In this it is analogous to mathematics, and indeed the “science” part of computer science is, for the most part, mathe-

matical in spirit. But an inevitable aspect of computer science is the creation of computer programs: objects that, though intangible, are subject to commercial exchange.

More than anything else, the greatest danger to good computer science research today may be *excessive relevance*. Evidence for the worldwide fascination with computers is everywhere, from the articles on the financial, and even the front pages of the newspapers, to the difficulties that even the most prestigious universities experience in finding and keeping faculty in computer science. The best professors, instead of teaching bright students, join start-up companies, and often discover that their brightest students have preceded them. Computer science is in the limelight, especially those aspects, such as systems, languages, and machine architecture, that may have immediate commercial applications. The attention is flattering, but it can work to the detriment of good research.

As the intensity of research in a particular area increases, so does the impulse to keep its results secret. This is true even in the university (Watson's account [12] of the discovery of the structure of DNA provides a well-known example), although in academia there is a strong counterpressure: Unless one publishes, one never becomes known at all. In industry, a natural impulse of the establishment is to guard proprietary information. Researchers understand reasonable restrictions on what and when they publish, but many will become irritated and flee elsewhere, or start working in less delicate areas, if prevented from communicating their discoveries and inventions in suitable fashion. Research management at Bell Labs has traditionally been sensitive to maintaining a careful balance between company interests and the industrial equivalent of academic freedom. The entrance of AT&T into the computer industry will test, and perhaps strain, this balance.

Another danger is that commercial pressures of one sort or another will divert the attention of the best thinkers from real innovation to exploitation of the current fad, from prospecting to mining a known lode. These pressures manifest themselves not only in the disappearance of faculty into industry, but also in the conservatism that overtakes those with well-paying investments—intellectual or financial—in a given idea. Perhaps this effect explains why so few interesting software systems have come from the large computer companies; they are locked into the existing world. Even IBM, which supports a well-regarded and productive research establishment, has in recent years produced little to cause even a minor revolution in the way people think about computers. The working examples of important new systems seem to have come either from entrepreneurial efforts (Visicalc is a good example) or from large companies, like Bell Labs and most especially Xerox, that were much involved with computers and could afford research into them, but did not regard them as their primary business.

On the other hand, in smaller companies, even the most vigorous research support is highly dependent on market conditions. *The New York Times*, in an article

describing Alan Kay's passage from Atari to Apple, notes the problem: "Mr. Kay . . . said that Atari's laboratories had lost some of the atmosphere of innovation that once attracted some of the finest talent in the industry. "When I left last month it was clear that they would be putting their efforts in the short term," he said . . . "I guess the tree of research must from time to time be refreshed with the blood of bean counters." [9]

Partly because they are new and still immature, and partly because they are a creation of the intellect, the arts and sciences of software abridge the chain, usual in physics and engineering, between fundamental discoveries, advanced development, and application. The inventors of ideas about how software should work usually find it necessary to build demonstration systems. For large systems, and for revolutionary ideas, much time is required: It can be said that UNIX was written in the 70s to distill the best systems ideas of the 60s, and became the commonplace of the 80s. The work at Xerox PARC on personal computers, bitmap graphics, and programming environments [10] shows a similar progression, starting, and coming to fruition a few years later. Time, and a commitment to the long-term value of the research, are needed on the part of both the researchers and their management.

Bell Labs has provided this commitment and more: a rare and uniquely stimulating research environment for my colleagues and me. As it enters what company publications call "the new competitive era," its managers and workers will do well to keep in mind how, and under what conditions, the UNIX system succeeded. If we can keep alive enough openness to new ideas, enough freedom of communication, enough patience to allow the novel to prosper, it will remain possible for a future Ken Thompson to find a little-used CRAY/I computer and fashion a system as creative, and as influential, as UNIX.

REFERENCES

1. Bell Labs: New order augurs well. *Nature* 305, 5933 (Sept. 29, 1983).
2. Bell Labs on the brink. *Science* 221 (Sept. 23, 1983).
3. Lesk, M. E. User-activated BTL directory assistance. Bell Laboratories internal memorandum (1972).
4. Norman, D. A. The truth about UNIX. *Datamation* 27, 12 (1981).
5. Organick, E. I. *The Multics System*. MIT Press, Cambridge, MA, 1972.
6. Ritchie, D. M. UNIX time-sharing system: A retrospective. *Bell Syst. Tech. J.* 57, 6 (1978), 1947–1969.
7. Ritchie, D. M. The evolution of the UNIX time-sharing system. In *Language Design and Programming Methodology*, Jeffrey M. Tobias, ed., Springer-Verlag, New York, (1980).
8. Ritchie, D. M. and Thompson, K. The UNIX time-sharing system. *Commun. ACM* 17, 7 (July 1974), 365–375.
9. Sanger, D. E. Key Atari scientist switches to Apple. *The New York Times* 133, 46, 033 (May 3, 1984).
10. Thacker, C. P. et al. Alto, a personal computer. Xerox PARC Technical Report CSL-79-11.
11. Thompson, K. UNIX time-sharing system: UNIX implementation. *Bell Syst. Tech. J.* 57, 6 (1978), 1931–1946.
12. Watson, J. D. *The Double Helix: A Personal Account of the Discovery of the Structure of DNA*. Atheneum Publishers, New York (1968).

Author's Present Address: Dennis M. Ritchie, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

The Evolution of the Unix Time-sharing System*

Dennis M. Ritchie
Bell Laboratories, Murray Hill, NJ, 07974

ABSTRACT

This paper presents a brief history of the early development of the Unix operating system. It concentrates on the evolution of the file system, the process-control mechanism, and the idea of pipelined commands. Some attention is paid to social conditions during the development of the system.

NOTE: **This paper was first presented at the Language Design and Programming Methodology conference at Sydney, Australia, September 1979. The conference proceedings were published as Lecture Notes in Computer Science #79: Language Design and Programming Methodology, Springer-Verlag, 1980. This rendition is based on a reprinted version appearing in AT&T Bell Laboratories Technical Journal 63 No. 6 Part 2, October 1984, pp. 1577-93.*

Introduction

During the past few years, the Unix operating system has come into wide use, so wide that its very name has become a trademark of Bell Laboratories. Its important characteristics have become known to many people. It has suffered much rewriting and tinkering since the first publication describing it in 1974 [1], but few fundamental changes. However, Unix was born in 1969 not 1974, and the account of its development makes a little-known and perhaps instructive story. This paper presents a technical and social history of the evolution of the system.

Origins

For computer science at Bell Laboratories, the period 1968-1969 was somewhat unsettled. The main reason for this was the slow, though clearly inevitable, withdrawal of the Labs from the Multics project. To the Labs computing community as a whole, the problem was the increasing obviousness of the failure of Multics to deliver promptly any sort of usable system, let alone the panacea envisioned earlier. For much of this time, the Murray Hill Computer Center was also running a costly GE 645 machine that inadequately simulated the GE 635. Another shake-up that occurred during this period was the organizational separation of computing services and computing research.

From the point of view of the group that was to be most involved in the beginnings of Unix (K. Thompson, Ritchie, M. D. McIlroy, J. F. Ossanna), the decline and fall of Multics had a directly felt effect. We were among the last Bell Laboratories holdouts actually working on Multics, so we still felt some sort of stake in its success. More important, the convenient interactive computing service that Multics had promised to the entire community was in fact available to our limited group, at first under the CTSS system used to develop Multics, and later under Multics itself. Even though Multics could not then support many users, it could support us, albeit at exorbitant cost. We didn't want to lose the

pleasant niche we occupied, because no similar ones were available; even the time-sharing service that would later be offered under GE's operating system did not exist. What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication.

Thus, during 1969, we began trying to find an alternative to Multics. The search took several forms. Throughout 1969 we (mainly Ossanna, Thompson, Ritchie) lobbied intensively for the purchase of a medium-scale machine for which we promised to write an operating system; the machines we suggested were the DEC PDP-10 and the SDS (later Xerox) Sigma 7. The effort was frustrating, because our proposals were never clearly and finally turned down, but yet were certainly never accepted. Several times it seemed we were very near success. The final blow to this effort came when we presented an exquisitely complicated proposal, designed to minimize financial outlay, that involved some outright purchase, some third-party lease, and a plan to turn in a DEC KA-10 processor on the soon-to-be-announced and more capable KI-10. The proposal was rejected, and rumor soon had it that W. O. Baker (then vice-president of Research) had reacted to it with the comment 'Bell Laboratories just doesn't do business this way!'

Actually, it is perfectly obvious in retrospect (and should have been at the time) that we were asking the Labs to spend too much money on too few people with too vague a plan. Moreover, I am quite sure that at that time operating systems were not, for our management, an attractive area in which to support work. They were in the process of extricating themselves not only from an operating system development effort that had failed, but from running the local Computation Center. Thus it may have seemed that buying a machine such as we suggested might lead on the one hand to yet another Multics, or on the other, if we produced something useful, to yet another Comp Center for them to be responsible for.

Besides the financial agitations that took place in 1969, there was technical work also. Thompson, R. H. Canaday, and Ritchie developed, on blackboards and scribbled notes, the basic design of a file system that was later to become the heart of Unix. Most of the design was Thompson's, as was the impulse to think about file systems at all, but I believe I contributed the idea of device files. Thompson's itch for creation of an operating system took several forms during this period; he also wrote (on Multics) a fairly detailed simulation of the performance of the proposed file system design and of paging behavior of programs. In addition, he started work on a new operating system for the GE-645, going as far as writing an assembler for the machine and a rudimentary operating system kernel whose greatest achievement, so far as I remember, was to type a greeting message. The complexity of the machine was such that a mere message was already a fairly notable accomplishment, but when it became clear that the lifetime of the 645 at the Labs was measured in months, the work was dropped.

Also during 1969, Thompson developed the game of 'Space Travel.' First written on Multics, then transliterated into Fortran for GECOS (the operating system for the GE, later Honeywell, 635), it was nothing less than a simulation of the movement of the major bodies of the Solar System, with the player guiding a ship here and there, observing the scenery, and attempting to land on the various planets and moons. The GECOS version was unsatisfactory in two important respects: first, the display of the state of the game was jerky and hard to control because one had to type commands at it, and second, a game cost about \$75 for CPU time on the big computer. It did not take long, therefore, for Thompson to find a little-used PDP-7 computer with an excellent display processor; the whole system was used as a

Graphic-II terminal. He and I rewrote Space Travel to run on this machine. The undertaking was more ambitious than it might seem; because we disdained all existing software, we had to write a floating-point arithmetic package, the pointwise specification of the graphic characters for the display, and a debugging subsystem that continuously displayed the contents of typed-in locations in a corner of the screen. All this was written in assembly language for a cross-assembler that ran under GECOS and produced paper tapes to be carried to the PDP-7.

Space Travel, though it made a very attractive game, served mainly as an introduction to the clumsy technology of preparing programs for the PDP-7. Soon Thompson began implementing the paper file system (perhaps ‘chalk file system’ would be more accurate) that had been designed earlier. A file system without a way to exercise it is a sterile proposition, so he proceeded to flesh it out with the other requirements for a working operating system, in particular the notion of processes. Then came a small set of user-level utilities: the means to copy, print, delete, and edit files, and of course a simple command interpreter (shell). Up to this time all the programs were written using GECOS and files were transferred to the PDP-7 on paper tape; but once an assembler was completed the system was able to support itself. Although it was not until well into 1970 that Brian Kernighan suggested the name ‘Unix,’ in a somewhat treacherous pun on ‘Multics,’ the operating system we know today was born.

The PDP-7 Unix file system

Structurally, the file system of PDP-7 Unix was nearly identical to today’s. It had

- 1) An i-list: a linear array of *i-nodes* each describing a file. An i-node contained less than it does now, but the essential information was the same: the protection mode of the file, its type and size, and the list of physical blocks holding the contents.
- 2) Directories: a special kind of file containing a sequence of names and the associated i-number.
- 3) Special files describing devices. The device specification was not contained explicitly in the i-node, but was instead encoded in the number: specific i-numbers corresponded to specific files.

The important file system calls were also present from the start. Read, write, open, creat (sic), close: with one very important exception, discussed below, they were similar to what one finds now. A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters. Another minor, occasionally annoying difference was the lack of erase and kill processing for terminals. Terminals, in effect, were always in raw mode. Only a few programs (notably the shell and the editor) bothered to implement erase-kill processing.

In spite of its considerable similarity to the current file system, the PDP-7 file system was in one way remarkably different: there were no path names, and each file-name argument to the system was a simple name (without '/') taken relative to the current directory. Links, in the usual Unix sense, did exist. Together with an elaborate set of conventions, they were the principal means by which the lack of path names became acceptable.

The *link* call took the form

```
link(dir, file, newname)
```

where *dir* was a directory file in the current directory, *file* an existing entry in that directory, and

newname the name of the link, which was added to the current directory. Because *dir* needed to be in the current directory, it is evident that today's prohibition against links to directories was not enforced; the PDP-7 Unix file system had the shape of a general directed graph.

So that every user did not need to maintain a link to all directories of interest, there existed a directory called *dd* that contained entries for the directory of each user. Thus, to make a link to file *x* in directory *ken*, I might do

```
ln dd ken ken  
ln ken x x  
rm ken
```

This scheme rendered subdirectories sufficiently hard to use as to make them unused in practice. Another important barrier was that there was no way to create a directory while the system was running; all were made during recreation of the file system from paper tape, so that directories were in effect a nonrenewable resource.

The *dd* convention made the *chdir* command relatively convenient. It took multiple arguments, and switched the current directory to each named directory in turn. Thus

```
chdir dd ken
```

would move to directory *ken*. (Incidentally, *chdir* was spelled *ch*; why this was expanded when we went to the PDP-11 I don't remember.)

The most serious inconvenience of the implementation of the file system, aside from the lack of path names, was the difficulty of changing its configuration; as mentioned, directories and special files were both made only when the disk was recreated. Installation of a new device was very painful, because the code for devices was spread widely throughout the system; for example there were several loops that visited each device in turn. Not surprisingly, there was no notion of mounting a removable disk pack, because the machine had only a single fixed-head disk.

The operating system code that implemented this file system was a drastically simplified version of the present scheme. One important simplification followed from the fact that the system was not multi-programmed; only one program was in memory at a time, and control was passed between processes only when an explicit swap took place. So, for example, there was an *iget* routine that made a named i-node available, but it left the i-node in a constant, static location rather than returning a pointer into a large table of active i-nodes. A precursor of the current buffering mechanism was present (with about 4 buffers) but there was essentially no overlap of disk I/O with computation. This was avoided not merely for simplicity. The disk attached to the PDP-7 was fast for its time; it transferred one 18-bit word every 2 microseconds. On the other hand, the PDP-7 itself had a memory cycle time of 1 microsecond, and most instructions took 2 cycles (one for the instruction itself, one for the operand). However, indirectly addressed instructions required 3 cycles, and indirection was quite common, because the machine had no index registers. Finally, the DMA controller was unable to access memory during an instruction. The upshot was that the disk would incur overrun errors if any indirectly-addressed instructions were executed while it was transferring. Thus control could not be returned to the user, nor in fact could general system code be executed, with the disk running. The interrupt routines for the clock and terminals, which needed to be runnable at all times, had to be coded in very strange fashion to avoid indirection.

Process control

By ‘process control,’ I mean the mechanisms by which processes are created and used; today the system calls *fork*, *exec*, *wait*, and *exit* implement these mechanisms. Unlike the file system, which existed in nearly its present form from the earliest days, the process control scheme underwent considerable mutation after PDP-7 Unix was already in use. (The introduction of path names in the PDP-11 system was certainly a considerable notational advance, but not a change in fundamental structure.)

Today, the way in which commands are executed by the shell can be summarized as follows:

- 1) The shell reads a command line from the terminal.
- 2) It creates a child process by *fork*.
- 3) The child process uses *exec* to call in the command from a file.
- 4) Meanwhile, the parent shell uses *wait* to wait for the child (command) process to terminate by calling *exit*.
- 5) The parent shell goes back to step 1).

Processes (independently executing entities) existed very early in PDP-7 Unix. There were in fact precisely two of them, one for each of the two terminals attached to the machine. There was no *fork*, *wait*, or *exec*. There was an *exit*, but its meaning was rather different, as will be seen. The main loop of the shell went as follows.

- 1) The shell closed all its open files, then opened the terminal special file for standard input and output (file descriptors 0 and 1).
- 2) It read a command line from the terminal.
- 3) It linked to the file specifying the command, opened the file, and removed the link. Then it copied a small bootstrap program to the top of memory and jumped to it; this bootstrap program read in the file over the shell code, then jumped to the first location of the command (in effect an *exec*).
- 4) The command did its work, then terminated by calling *exit*. The *exit* call caused the system to read in a fresh copy of the shell over the terminated command, then to jump to its start (and thus in effect to go to step 1).

The most interesting thing about this primitive implementation is the degree to which it anticipated themes developed more fully later. True, it could support neither background processes nor shell command files (let alone pipes and filters); but IO redirection (via ‘<’ and ‘>’) was soon there; it is discussed below. The implementation of redirection was quite straightforward; in step 3) above the shell just replaced its standard input or output with the appropriate file. Crucial to subsequent development was the implementation of the shell as a user-level program stored in a file, rather than a part of the operating system.

The structure of this process control scheme, with one process per terminal, is similar to that of many interactive systems, for example CTSS, Multics, Honeywell TSS, and IBM TSS and TSO. In general such systems require special mechanisms to implement useful facilities such as detached computations and command files; Unix at that stage didn’t bother to supply the special mechanisms. It also exhibited some irritating, idiosyncratic problems. For example, a newly recreated shell had to close all its open files both to get rid of any open files left by the command just executed and to rescind previous IO redirection. Then it had to reopen the special file corresponding to its terminal, in order to read a new command line. There was no */dev* directory (because no path names); moreover, the shell could retain no

memory across commands, because it was reexecuted afresh after each command. Thus a further file system convention was required: each directory had to contain an entry *tty* for a special file that referred to the terminal of the process that opened it. If by accident one changed into some directory that lacked this entry, the shell would loop hopelessly; about the only remedy was to reboot. (Sometimes the missing link could be made from the other terminal.)

Process control in its modern form was designed and implemented within a couple of days. It is astonishing how easily it fitted into the existing system; at the same time it is easy to see how some of the slightly unusual features of the design are present precisely because they represented small, easily-coded changes to what existed. A good example is the separation of the *fork* and *exec* functions. The most common model for the creation of new processes involves specifying a program for the process to execute; in Unix, a forked process continues to run the same program as its parent until it performs an explicit *exec*. The separation of the functions is certainly not unique to Unix, and in fact it was present in the Berkeley time-sharing system [2], which was well-known to Thompson. Still, it seems reasonable to suppose that it exists in Unix mainly because of the ease with which *fork* could be implemented without changing much else. The system already handled multiple (i.e. two) processes; there was a process table, and the processes were swapped between main memory and the disk. The initial implementation of *fork* required only

- 1) Expansion of the process table
- 2) Addition of a *fork* call that copied the current process to the disk swap area, using the already existing swap IO primitives, and made some adjustments to the process table.

In fact, the PDP-7's *fork* call required precisely 27 lines of assembly code. Of course, other changes in the operating system and user programs were required, and some of them were rather interesting and unexpected. But a combined *fork-exec* would have been considerably more complicated, if only because *exec* as such did not exist; its function was already performed, using explicit IO, by the shell.

The *exit* system call, which previously read in a new copy of the shell (actually a sort of automatic *exec* but without arguments), simplified considerably; in the new version a process only had to clean out its process table entry, and give up control.

Curiously, the primitives that became *wait* were considerably more general than the present scheme. A pair of primitives sent one-word messages between named processes:

```
smes(pid, message)
(pid, message) = rmes()
```

The target process of *smes* did not need to have any ancestral relationship with the receiver, although the system provided no explicit mechanism for communicating process IDs except that *fork* returned to each of the parent and child the ID of its relative. Messages were not queued; a sender delayed until the receiver read the message.

The message facility was used as follows: the parent shell, after creating a process to execute a command, sent a message to the new process by *smes*; when the command terminated (assuming it did not try to read any messages) the shell's blocked *smes* call returned an error indication that the target process did not exist. Thus the shell's *smes* became, in effect, the equivalent of *wait*.

A different protocol, which took advantage of more of the generality offered by messages, was used

between the initialization program and the shells for each terminal. The initialization process, whose ID was understood to be 1, created a shell for each of the terminals, and then issued *rmes*; each shell, when it read the end of its input file, used *smes* to send a conventional ‘I am terminating’ message to the initialization process, which recreated a new shell process for that terminal.

I can recall no other use of messages. This explains why the facility was replaced by the *wait* call of the present system, which is less general, but more directly applicable to the desired purpose. Possibly relevant also is the evident bug in the mechanism: if a command process attempted to use messages to communicate with other processes, it would disrupt the shell’s synchronization. The shell depended on sending a message that was never received; if a command executed *rmes*, it would receive the shell’s phony message, and cause the shell to read another input line just as if the command had terminated. If a need for general messages had manifested itself, the bug would have been repaired.

At any rate, the new process control scheme instantly rendered some very valuable features trivial to implement; for example detached processes (with ‘&’) and recursive use of the shell as a command. Most systems have to supply some sort of special ‘batch job submission’ facility and a special command interpreter for files distinct from the one used interactively.

Although the multiple-process idea slipped in very easily indeed, there were some aftereffects that weren’t anticipated. The most memorable of these became evident soon after the new system came up and apparently worked. In the midst of our jubilation, it was discovered that the *chdir* (change current directory) command had stopped working. There was much reading of code and anxious introspection about how the addition of *fork* could have broken the *chdir* call. Finally the truth dawned: in the old system *chdir* was an ordinary command; it adjusted the current directory of the (unique) process attached to the terminal. Under the new system, the *chdir* command correctly changed the current directory of the process created to execute it, but this process promptly terminated and had no effect whatsoever on its parent shell! It was necessary to make *chdir* a special command, executed internally within the shell. It turns out that several command-like functions have the same property, for example *login*.

Another mismatch between the system as it had been and the new process control scheme took longer to become evident. Originally, the read/write pointer associated with each open file was stored within the process that opened the file. (This pointer indicates where in the file the next read or write will take place.) The problem with this organization became evident only when we tried to use command files. Suppose a simple command file contains

```
ls  
who
```

and it is executed as follows:

```
sh comfile >output
```

The sequence of events was

- 1) The main shell creates a new process, which opens *outfile* to receive the standard output and executes the shell recursively.
- 2) The new shell creates another process to execute *ls*, which correctly writes on file *output* and then terminates.

- 3) Another process is created to execute the next command. However, the IO pointer for the output is copied from that of the shell, and it is still 0, because the shell has never written on its output, and IO pointers are associated with processes. The effect is that the output of *who* overwrites and destroys the output of the preceding *ls* command.

Solution of this problem required creation of a new system table to contain the IO pointers of open files independently of the process in which they were opened.

IO Redirection

The very convenient notation for IO redirection, using the ‘>’ and ‘<’ characters, was not present from the very beginning of the PDP-7 Unix system, but it did appear quite early. Like much else in Unix, it was inspired by an idea from Multics. Multics has a rather general IO redirection mechanism [3] embodying named IO streams that can be dynamically redirected to various devices, files, and even through special stream-processing modules. Even in the version of Multics we were familiar with a decade ago, there existed a command that switched subsequent output normally destined for the terminal to a file, and another command to reattach output to the terminal. Where under Unix one might say

```
ls >xx
```

to get a listing of the names of one’s files in *xx*, on Multics the notation was

```
iocall attach user_output file xx  
list  
iocall attach user_output syn user_i/o
```

Even though this very clumsy sequence was used often during the Multics days, and would have been utterly straightforward to integrate into the Multics shell, the idea did not occur to us or anyone else at the time. I speculate that the reason it did not was the sheer size of the Multics project: the implementors of the IO system were at Bell Labs in Murray Hill, while the shell was done at MIT. We didn’t consider making changes to the shell (it was *their* program); correspondingly, the keepers of the shell may not even have known of the usefulness, albeit clumsiness, of *iocall*. (The 1969 Multics manual [4] lists *iocall* as an ‘author-maintained,’ that is non-standard, command.) Because both the Unix IO system and its shell were under the exclusive control of Thompson, when the right idea finally surfaced, it was a matter of an hour or so to implement it.

The advent of the PDP-11

By the beginning of 1970, PDP-7 Unix was a going concern. Primitive by today’s standards, it was still capable of providing a more congenial programming environment than its alternatives. Nevertheless, it was clear that the PDP-7, a machine we didn’t even own, was already obsolete, and its successors in the same line offered little of interest. In early 1970 we proposed acquisition of a PDP-11, which had just been introduced by Digital. In some sense, this proposal was merely the latest in the series of attempts that had been made throughout the preceding year. It differed in two important ways. First, the amount of money (about \$65,000) was an order of magnitude less than what we had previously asked; second, the charter sought was not merely to write some (unspecified) operating system, but instead to create a system specifically designed for editing and formatting text, what might today be called a ‘word-processing system.’ The impetus for the proposal came mainly from J. F. Ossanna, who was then and until the end of his life interested in text processing. If our early proposals were too vague, this one

was perhaps too specific; at first it too met with disfavor. Before long, however, funds were obtained through the efforts of L. E. McMahon and an order for a PDP-11 was placed in May.

The processor arrived at the end of the summer, but the PDP-11 was so new a product that no disk was available until December. In the meantime, a rudimentary, core-only version of Unix was written using a cross-assembler on the PDP-7. Most of the time, the machine sat in a corner, enumerating all the closed Knight's tours on a 6×8 chess board-a three-month job.

The first PDP-11 system

Once the disk arrived, the system was quickly completed. In internal structure, the first version of Unix for the PDP-11 represented a relatively minor advance over the PDP-7 system; writing it was largely a matter of transliteration. For example, there was no multi-programming; only one user program was present in core at any moment. On the other hand, there were important changes in the interface to the user: the present directory structure, with full path names, was in place, along with the modern form of *exec* and *wait*, and conveniences like character-erase and line-kill processing for terminals. Perhaps the most interesting thing about the enterprise was its small size: there were 24K bytes of core memory (16K for the system, 8K for user programs), and a disk with 1K blocks (512K bytes). Files were limited to 64K bytes.

At the time of the placement of the order for the PDP-11, it had seemed natural, or perhaps expedient, to promise a system dedicated to word processing. During the protracted arrival of the hardware, the increasing usefulness of PDP-7 Unix made it appropriate to justify creating PDP-11 Unix as a development tool, to be used in writing the more special-purpose system. By the spring of 1971, it was generally agreed that no one had the slightest interest in scrapping Unix. Therefore, we transliterated the *roff* text formatter into PDP-11 assembler language, starting from the PDP-7 version that had been transliterated from McIlroy's BCPL version on Multics, which had in turn been inspired by J. Saltzer's *runoff* program on CTSS. In early summer, editor and formatter in hand, we felt prepared to fulfill our charter by offering to supply a text-processing service to the Patent department for preparing patent applications. At the time, they were evaluating a commercial system for this purpose; the main advantages we offered (besides the dubious one of taking part in an in-house experiment) were two in number: first, we supported Teletype's model 37 terminals, which, with an extended type-box, could print most of the math symbols they required; second, we quickly endowed *roff* with the ability to produce line-numbered pages, which the Patent Office required and which the other system could not handle.

During the last half of 1971, we supported three typists from the Patent department, who spent the day busily typing, editing, and formatting patent applications, and meanwhile tried to carry on our own work. Unix has a reputation for supplying interesting services on modest hardware, and this period may mark a high point in the benefit/equipment ratio; on a machine with no memory protection and a single .5 MB disk, every test of a new program required care and boldness, because it could easily crash the system, and every few hours' work by the typists meant pushing out more information onto DECTape, because of the very small disk.

The experiment was trying but successful. Not only did the Patent department adopt Unix, and thus become the first of many groups at the Laboratories to ratify our work, but we achieved sufficient credibility to convince our own management to acquire one of the first PDP 11/45 systems made. We have accumulated much hardware since then, and labored continuously on the software, but because

most of the interesting work has already been published, (e.g. on the system itself [1, 5, 6, 7, 8, 9]) it seems unnecessary to repeat it here.

Pipes

One of the most widely admired contributions of Unix to the culture of operating systems and command languages is the *pipe*, as used in a pipeline of commands. Of course, the fundamental idea was by no means new; the pipeline is merely a specific form of coroutine. Even the implementation was not unprecedented, although we didn't know it at the time; the 'communication files' of the Dartmouth Time-Sharing System [10] did very nearly what Unix pipes do, though they seem not to have been exploited so fully.

Pipes appeared in Unix in 1972, well after the PDP-11 version of the system was in operation, at the suggestion (or perhaps insistence) of M. D. McIlroy, a long-time advocate of the non-hierarchical control flow that characterizes coroutines. Some years before pipes were implemented, he suggested that commands should be thought of as binary operators, whose left and right operand specified the input and output files. Thus a 'copy' utility would be commanded by

```
inputfile copy outfile
```

To make a pipeline, command operators could be stacked up. Thus, to sort *input*, paginate it neatly, and print the result off-line, one would write

```
input sort paginate offprint
```

In today's system, this would correspond to

```
sort input | pr | opr
```

The idea, explained one afternoon on a blackboard, intrigued us but failed to ignite any immediate action. There were several objections to the idea as put: the infix notation seemed too radical (we were too accustomed to typing 'cp x y' to copy *x* to *y*); and we were unable to see how to distinguish command parameters from the input or output files. Also, the one-input one-output model of command execution seemed too confining. What a failure of imagination!

Some time later, thanks to McIlroy's persistence, pipes were finally installed in the operating system (a relatively simple job), and a new notation was introduced. It used the same characters as for I/O redirection. For example, the pipeline above might have been written

```
sort input >pr>opr>
```

The idea is that following a '>' may be either a file, to specify redirection of output to that file, or a command into which the output of the preceding command is directed as input. The trailing '>' was needed in the example to specify that the (nonexistent) output of *opr* should be directed to the console; otherwise the command *opr* would not have been executed at all; instead a file *opr* would have been created.

The new facility was enthusiastically received, and the term 'filter' was soon coined. Many commands were changed to make them usable in pipelines. For example, no one had imagined that anyone would want the *sort* or *pr* utility to sort or print its standard input if given no explicit arguments.

Soon some problems with the notation became evident. Most annoying was a silly lexical problem: the string after ‘>’ was delimited by blanks, so, to give a parameter to *pr* in the example, one had to quote:

```
sort input >"pr -2">opr>
```

Second, in attempt to give generality, the pipe notation accepted ‘<’ as an input redirection in a way corresponding to ‘>’; this meant that the notation was not unique. One could also write, for example,

```
opr <pr<"sort input"<
```

or even

```
pr <"sort input"< >opr>
```

The pipe notation using ‘<’ and ‘>’ survived only a couple of months; it was replaced by the present one that uses a unique operator to separate components of a pipeline. Although the old notation had a certain charm and inner consistency, the new one is certainly superior. Of course, it too has limitations. It is unabashedly linear, though there are situations in which multiple redirected inputs and outputs are called for. For example, what is the best way to compare the outputs of two programs? What is the appropriate notation for invoking a program with two parallel output streams?

I mentioned above in the section on IO redirection that Multics provided a mechanism by which IO streams could be directed through processing modules on the way to (or from) the device or file serving as source or sink. Thus it might seem that stream-splicing in Multics was the direct precursor of Unix pipes, as Multics IO redirection certainly was for its Unix version. In fact I do not think this is true, or is true only in a weak sense. Not only were coroutines well-known already, but their embodiment as Multics spliceable IO modules required that the modules be specially coded in such a way that they could be used for no other purpose. The genius of the Unix pipeline is precisely that it is constructed from the very same commands used constantly in simplex fashion. The mental leap needed to see this possibility and to invent the notation is large indeed.

High-level languages

Every program for the original PDP-7 Unix system was written in assembly language, and bare assembly language it was—for example, there were no macros. Moreover, there was no loader or link-editor, so every program had to be complete in itself. The first interesting language to appear was a version of McClure’s TMG [11] that was implemented by McIlroy. Soon after TMG became available, Thompson decided that we could not pretend to offer a real computing service without Fortran, so he sat down to write a Fortran in TMG. As I recall, the intent to handle Fortran lasted about a week. What he produced instead was a definition of and a compiler for the new language B [12]. B was much influenced by the BCPL language [13]; other influences were Thompson’s taste for spartan syntax, and the very small space into which the compiler had to fit. The compiler produced simple interpretive code; although it and the programs it produced were rather slow, it made life much more pleasant. Once interfaces to the regular system calls were made available, we began once again to enjoy the benefits of using a reasonable language to write what are usually called ‘systems programs:’ compilers, assemblers, and the like. (Although some might consider the PL/I we used under Multics unreasonable, it was much better than assembly language.) Among other programs, the PDP-7 B cross-compiler for the PDP-11 was written in B, and in the course of time, the B compiler for the PDP-7 itself was transliterated from TMG into B.

When the PDP-11 arrived, B was moved to it almost immediately. In fact, a version of the multi-precision ‘desk calculator’ program *dc* was one of the earliest programs to run on the PDP-11, well before the disk arrived. However, B did not take over instantly. Only passing thought was given to rewriting the operating system in B rather than assembler, and the same was true of most of the utilities. Even the assembler was rewritten in assembler. This approach was taken mainly because of the slowness of the interpretive code. Of smaller but still real importance was the mismatch of the word-oriented B language with the byte-addressed PDP-11.

Thus, in 1971, work began on what was to become the C language [14]. The story of the language developments from BCPL through B to C is told elsewhere [15], and need not be repeated here. Perhaps the most important watershed occurred during 1973, when the operating system kernel was rewritten in C. It was at this point that the system assumed its modern form; the most far-reaching change was the introduction of multi-programming. There were few externally-visible changes, but the internal structure of the system became much more rational and general. The success of this effort convinced us that C was useful as a nearly universal tool for systems programming, instead of just a toy for simple applications.

Today, the only important Unix program still written in assembler is the assembler itself; virtually all the utility programs are in C, and so are most of the applications programs, although there are sites with many in Fortran, Pascal, and Algol 68 as well. It seems certain that much of the success of Unix follows from the readability, modifiability, and portability of its software that in turn follows from its expression in high-level languages.

Conclusion

One of the comforting things about old memories is their tendency to take on a rosy glow. The programming environment provided by the early versions of Unix seems, when described here, to be extremely harsh and primitive. I am sure that if forced back to the PDP-7 I would find it intolerably limiting and lacking in conveniences. Nevertheless, it did not seem so at the time; the memory fixes on what was good and what lasted, and on the joy of helping to create the improvements that made life better. In ten years, I hope we can look back with the same mixed impression of progress combined with continuity.

Acknowledgements

I am grateful to S. P. Morgan, K. Thompson, and M. D. McIlroy for providing early documents and digging up recollections.

Because I am most interested in describing the evolution of ideas, this paper attributes ideas and work to individuals only where it seems most important. The reader will not, on the average, go far wrong if he reads each occurrence of ‘we’ with unclear antecedent as ‘Thompson, with some assistance from me.’

References

1. D. M. Ritchie and K. Thompson, ‘The Unix Time-sharing System, C. ACM **17** No. 7 (July 1974),

- pp 365-37.
2. L. P. Deutch and B. W. Lampson, 'SDS 930 Time-sharing System Preliminary Reference Manual,' Doc. 30.10.10, Project Genie, Univ. Cal. at Berkeley (April 1965).
 3. R. J. Feiertag and E. I. Organick, 'The Multics input-output system,' Proc. Third Symposium on Operating Systems Principles, October 18-20, 1971, pp. 35-41.
 4. *The Multiplexed Information and Computing Service: Programmers' Manual*, Mass. Inst. of Technology, Project MAC, Cambridge MA, (1969).
 5. K. Thompson, 'Unix Implementation,' Bell System Tech J. **57** No. 6, (July-August 1978), pp. 1931-46.
 6. S. C. Johnson and D. M. Ritchie, Portability of C Programs and the Unix System,' Bell System Tech J. **57** No. 6, (July-August 1978), pp. 2021-48.
 7. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna. 'Document Preparation,' Bell Sys. Tech. J., **57** No. 6, pp. 2115-2135.
 8. B. W. Kernighan and L. L. Cherry, 'A System for Typesetting Mathematics,' J. Comm. Assoc. Comp. Mach. **18**, pp. 151-157 (March 1975).
 9. M. E. Lesk and B. W. Kernighan, 'Computer Typesetting of Technical Journals on Unix,' Proc. AFIPS NCC **46** (1977), pp. 879-88.
 10. *Systems Programmers Manual for the Dartmouth Time Sharing System for the GE 635 Computer*, Dartmouth College, Hanover, New Hampshire, 1971.
 11. R. M. McClure, 'TMG--A Syntax-Directed Compiler,' Proc 20th ACM National Conf. (1968), pp. 262-74.
 12. S. C. Johnson and B. W. Kernighan, 'The Programming Language B,' Comp. Sci. Tech. Rep. #8, Bell Laboratories, Murray Hill NJ (1973).
 13. M. Richards, 'BCPL: A Tool for Compiler Writing and Systems Programming,' Proc. AFIPS SJCC **34** (1969), pp. 557-66.
 14. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs NJ, 1978. Second Edition, 1979.
 15. D. M. Ritchie, S. C. Johnson, and M. E. Lesk, 'The C Programming Language,' Bell Sys. Tech. J. **57** No. 6 (July-August 1978) pp. 1991-2019.

[Reprinted from *AT&T Bell Laboratories Technical Journal* **63**, No. 8 Part 2 (October, 1984), pp. 1897-1910. The current implementation of the stream mechanisms differs slightly from that described here, but the structure remains the same. Copyright © 1984 AT&T.]

A Stream Input-Output System

Dennis M. Ritchie

ABSTRACT

In a new version of the Unix operating system, a flexible coroutine-based design replaces the traditional rigid connection between processes and terminals or networks. Processing modules may be inserted dynamically into the stream that connects a user's program to a device. Programs may also connect directly to programs, providing inter-process communication.

Introduction

The part of the Unix operating system that deals with terminals and other character devices has always been complicated. In recent versions of the system it has become even more so, for two reasons.

- 1) Network connections require protocols more ornate than are easily accommodated in the existing structure. A notion of "line disciplines" was only partially successful, mostly because in the traditional system only one line discipline can be active at a time.
- 2) The fundamental data structure of the traditional character I/O system, a queue of individual characters (the "clist"), is costly because it accepts and dispenses characters one at a time. Attempts to avoid overhead by bypassing the mechanism entirely or by introducing *ad hoc* routines succeeded in speeding up the code at the expense of regularity.

Patchwork solutions to specific problems were destroying the modularity of this part of the system. The time was ripe to redo the whole thing. This paper describes the new organization.

The system described here runs on about 20 machines in the Information Sciences Research Division of Bell Laboratories. Although it is being investigated by other parts of Bell Labs, it is not generally available.

Overview

This section summarizes the nomenclature, components, and mechanisms of the new I/O system.

Streams

A *stream* is a full-duplex connection between a user's process and a device or pseudo-device. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions. The modules in a stream communicate almost exclusively by passing messages to their neighbors. Except for some conventional variables used for flow control, modules do not require access to the storage of their neighbors. Moreover, a module provides only one entry point to each neighbor, namely a routine that accepts messages.

At the end of the stream closest to the process is a set of routines that provide the interface to the rest of the system. A user's *write* and I/O control requests are turned into messages sent to the stream, and *read* requests take data from the stream and pass it to the user. At the other end of the stream is a device driver module. Here, data arriving from the stream is sent to the device; characters and state transitions detected by the device are composed into messages and sent into the stream towards the user program. Intermediate

modules process the messages in various ways.

The two end modules in a stream become connected automatically when the device is opened; intermediate modules are attached dynamically by request of the user's program. Stream processing modules are symmetrical; their read and write interfaces are identical.

Queues

Each stream processing module consists of a pair of *queues*, one for each direction. A queue comprises not only a data queue proper, but also two routines and some status information. One routine is the *put procedure*, which is called by its neighbor to place messages on the data queue. The other, the *service procedure*, is scheduled to execute whenever there is work for it to do. The status information includes a pointer to the next queue downstream, various flags, and a pointer to additional state information required by the instantiation of the queue. Queues are allocated in such a way that the routines associated with one half of a stream module may find the queue associated with the other half. (This is used, for example, in generating echos for terminal input.)

Message blocks

The objects passed between queues are blocks obtained from an allocator. Each contains a *read pointer*, a *write pointer*, and a *limit pointer*, which specify respectively the beginning of information being passed, its end, and a bound on the extent to which the write pointer may be increased.

The header of a block specifies its type; the most common blocks contain data. There are also control blocks of various kinds, all with the same form as data blocks and obtained from the same allocator. For example, there are control blocks to introduce delimiters into the data stream, to pass user I/O control requests, and to announce special conditions such as line break and carrier loss on terminal devices.

Although data blocks arrive in discrete units at the processing modules, boundaries between them are semantically insignificant; standard subroutines may try to coalesce adjacent data blocks in the same queue. Control blocks, however, are never coalesced.

Scheduling

Although each queue module behaves in some ways like a separate process, it is not a real process; the system saves no state information for a queue module that is not running. In particular queue processing routines do not block when they cannot proceed, but must explicitly return control. A queue may be *enabled* by mechanisms described below. When a queue becomes enabled, the system will, as soon as convenient, call its service procedure entry, which removes successive blocks from the associated data queue, processes them, and places them on the next queue by calling its put procedure. When there are no more blocks to process, or when the next queue becomes full, the service procedure returns to the system. Any special state information must be saved explicitly.

Standard routines make enabling of queue modules largely automatic. For example, the routine that puts a block on a queue enables the queue service routine if the queue was empty.

Flow Control

Associated with each queue is a pair of numbers used for flow control. A high-water mark limits the amount of data that may be outstanding in the queue; by convention, modules do not place data on a queue above its limit. A low-water mark is used for scheduling in this way: when a queue has exceeded its high-water mark, a flag is set. Then, when the routine that takes blocks from a data queue notices that this flag is set and that the queue has dropped below the low-water mark, the queue upstream of this one is enabled.

Simple Examples

Figure 1 depicts a stream device that has just been opened. The top-level routines, drawn as a pair of half-open rectangles on the left, are invoked by users' *read* and *write* calls. The writer routine sends messages to the device driver shown on the right. Data arriving from the device is composed into messages sent to the top-level reader routine, which returns the data to the user process when it executes *read*.

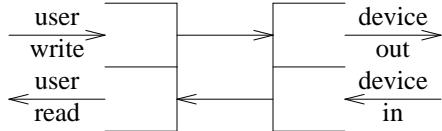


Figure 1. Configuration after device open.

Figure 2 shows an ordinary terminal connected by an RS-232 line. Here a processing module (the pair of rectangles in the middle) is interposed; it performs the services necessary to make terminals usable, for example echoing, character-erase and line-kill, tab expansion as required, and translation between carriage-return and new-line. It is possible to use one of several terminal handling modules. The standard one provides services like those of the Seventh Edition system [1]; another resembles the Berkeley “new tty” driver [2].

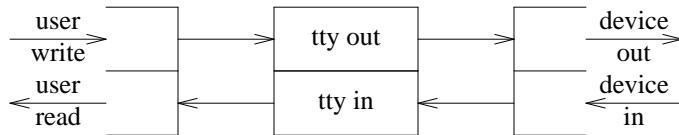


Figure 2. Configuration for normal terminal attachment.

The processing modules in a stream are thought of as a stack whose top (shown here on the left) is next to the user program. Thus, to install the terminal processing module after opening a terminal device, the program that makes such connections executes a “push” I/O control call naming the relevant stream and the desired processing module. Other primitives pop a module from the stack and determine the name of the topmost module.

Most of the machines using the version of the operating system described here are connected to a network based on the Datakit packet switch [3]. Although there is a variety of host interfaces to the network, most of ours are primitive, and require network protocols to be conducted by the host machine, rather than by a front-end processor. Therefore, when terminals are connected to a host through the network, a setup like that shown in Fig. 3 is used; the terminal processing module is stacked on the network protocol module. Again, there is a choice of protocol modules, both a current standard and an older protocol that is being phased out.

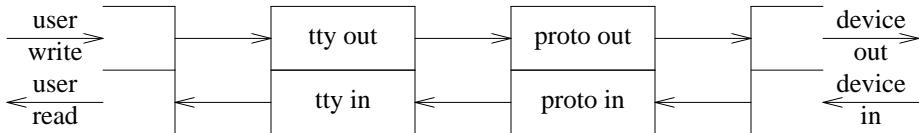


Figure 3. Configuration for network terminals.

A common fourth configuration (not illustrated) is used when the network is used for file transfers or other purposes when terminal processing is not needed. It simply omits the “tty” module and uses only the protocol module. Some of our machines, on the other hand, have front-end processors programmed to conduct standard network protocol. Here a connection for remote file transfer will resemble that of Fig. 1, because the protocol is handled outside the operating system; likewise network terminal connections via the front end will be handled as shown in Fig. 2.

Messages

Most of the messages between modules contain data. The allocator that dispenses message blocks takes an argument specifying the smallest block its caller is willing to accept. The current allocator maintains an inventory of blocks 4, 16, 64, and 1024 characters long. Modules that allocate blocks choose a size by balancing space loss in block linkage overhead against unused space in the block. For example, the top-level *write* routine requests either 64- or 1024-character blocks, because such calls usually transmit many characters; the network input routine allocates 16-byte blocks because data arrives in packets of that size. The smallest blocks are used only to carry arguments to the control messages discussed below.

Besides data blocks, there are also several kinds of control messages. The following messages are queued along with data messages, in order to ensure that their effect occurs at the appropriate time.

BREAK	is generated by a terminal device on detection of a line break signal. The standard terminal input processor turns this message into an interrupt request. It may also be sent to a terminal device driver to cause it to generate a break on the output line.
HANGUP	is generated by a device when its remote connection drops. When the message arrives at the top level it is turned into an interrupt to the process, and it also marks the stream so that further attempts to use it return errors.
DELIM	is a delimiter in the data. Most of the stream I/O system is prepared to provide true streams, in which record boundaries are insignificant, but there are various situations in which it is desirable to delimit the data. For example, terminal input is read a line at a time; DELIM is generated by the terminal input processor to demarcate lines.
DELAY	tells terminal drivers to generate a real-time delay on output; it allows time for slow terminals react to characters previously sent.
IOCTL	messages are generated by users' <i>ioctl</i> system calls. The relevant parameters are gathered at the top level, and if the request is not understood there, it and its parameters are composed into a message and sent down the stream. The first module that understands the particular request acts on it and returns a positive acknowledgement. Intermediate modules that do not recognize a particular IOCTL request pass it on; stream-end modules return a negative acknowledgement. The top-level routine waits for the acknowledgement, and returns any information it carries to the user.

Other control messages are asynchronous and jump over queued data and non-priority control messages.

IOCACK	
IOCNAK	acknowledge IOCTL messages. The device end of a stream must respond with one of these messages; the top level will eventually time out if no response is received.
SIGNAL	messages are generated by the terminal processing module and cause the top level to generate process signals such as <i>quit</i> and <i>interrupt</i> .
FLUSH	messages are used to throw away data from input and output queues after a signal or on request of the user.
STOP	
START	messages are used by the terminal processor to halt and restart output by a device, for example to implement the traditional control-S/control-Q (X-on/X-off) flow control mechanism.

Queue Mechanisms and Interfaces

Associated with each direction of a full-duplex stream module is a queue data structure with the following form (somewhat simplified for exposition).

```
struct queue {
    int    flag;        /* flag bits */
    void  (*putp)();   /* put procedure */
    void  (*servp)();  /* service procedure */
    struct queue *next; /* next queue downstream */
    struct block *first; /* first data block on queue */
    struct block *last; /* last data block on queue */
    int    hiwater;    /* max characters on queue */
    int    lowater;    /* wakeup point as queue drains */
    int    count;      /* characters now on queue */
    void  *ptr;        /* pointer to private storage */
};
```

The `flag` word contains several bits used by low-level routines to control scheduling: they show whether the downstream module wishes read data, or the upstream module wishes to write, or the queue is already enabled. One bit is examined by the upstream module; it tells whether this queue is full.

The `first` and `last` members point to the head and tail of a singly-linked list of data and control blocks that form the queue proper; `hiwater` and `lowater` are initialized when the queue is created, and when compared against `count`, the current size of the queue, determine whether the queue is full and whether it has emptied sufficiently to enable a blocked writer.

The `ptr` member stores an untyped pointer that may be used by the queue module to keep track of the location of storage private to itself. For example, each instantiation of the terminal processing module maintains a structure containing various mode bits and special characters; it stores a pointer to this structure here. The type of `ptr` is artificial. It should be a union of pointers to each possible module state structure.

Stream processing modules are written in one of two general styles. In the simpler kind, the queue module acts nearly as a classical coroutine. When it is instantiated, it sets its put procedure `putp` to a system-supplied default routine, and supplies a service procedure `servp`. Its upstream module disposes of blocks by calling this module's `putp` routine, which places the block on this module's queue (by manipulating the `first` and `last` pointers.) The standard put procedure also enables the current module; a short time later the current module's service procedure `servp` is called by the scheduler. In pseudo-code, the outline of a typical service routine is:

```
service(q)
struct queue *q

while (q is not empty and q->next is not full) {
    get a block from q
    process message block
    call q->next->putp to dispose of
        new or transformed block
}
```

This mechanism is appropriate in cases in which messages can be processed independently of each other. For example, it is used by the terminal output module. All the scheduling details are taken care of by standard routines.

More complicated modules need finer control over scheduling. A good example is terminal input. Here the device module upstream produces characters, usually one at a time, that must be gathered into a line to allow for character erase and kill processing. Therefore the stream input module provides a put procedure to be called by the device driver or other module downstream from it; here is an outline of this routine and its accompanying service procedure:

```
putproc(q, bp)
struct queue *q; struct block *bp

    put bp on q
    echo characters in bp's data
    if (bp's data contains new-line or carriage return)
        enable q

service(q)
struct queue *q

    take data from q until new-line or carriage return,
    processing erase and kill characters
    call q->next->putp to hand line to upstream queue
    call q->next->putp with DELIM message
```

The put procedure generates the echo characters as promptly as possible; when the terminal module is attached to a device handler, they are created during the input interrupt from the device, because the put procedure is called as a subroutine of the handler. On the other hand, line-gathering and erase and kill processing, which can be lengthy, are done during the service procedure at lower priority.

Connection with the Rest of the System

Although all the drivers for terminal and network devices, and all protocol handlers, were rewritten, only minor changes were required elsewhere in the system. Character devices and a character device switch, as described by Thompson [4], are still present. A pointer in the character device switch structure, if null, causes the system to treat the device as always; this is used for raw disk and tape, for example. If not null, it points to initialization information for the stream device; when a stream device is opened, the queue structure shown in Fig. 1 is created, using this information, and a pointer to the structure naming the stream is saved (in the “inode table”).

Subsequently, when the user process makes *read*, *write*, *ioctl*, or *close* calls, presence of a non-null stream pointer directs the system to use a set of stream routines to generate and receive queue messages; these are the “top-level routines” referred to previously.

Only a few changes in user-level code are necessary, most because opening a terminal puts it in the “very raw” mode shown in Fig. 1. In order to install the terminal-processing handler, it is necessary for programs such as *init* to execute the appropriate *iocctl* call.

Interprocess Communication

As previously described, the stream I/O system constitutes a flexible communication path between user processes and devices. With a small addition, it also provides a mechanism for interprocess communication. A special device, the “pseudo-terminal” or PT, connects processes. PT files come in even-odd pairs; data written on the odd member of the pair appears as input for the even member, and vice versa. The idea is not new; it appears in Tenex [5] and its successors, for example. It is analogous to pipes, and especially to named pipes [6]. PT files differ from traditional pipes in two ways: they are full-duplex, and control information passes through them as well as data. They differ from the usual pseudo-terminal files [2] by not having any of the usual terminal processing mechanisms inherently attached to them; they are pure transmitters of control and data messages. PT files are adequate for setting up a reasonably general mechanism for explicit process communication, but by themselves are not especially interesting.

A special *message* module provides more intriguing possibilities. In one direction, the message processor takes control and data messages, such as those discussed above, and transforms them into data blocks starting with a header giving the message type, and followed by the message content. In the other direction, it parses similarly-structured data messages and creates the corresponding control blocks. Figure 4 shows a configuration in which a user process communicates through the terminal module, a PT file pair, and the message module with another user-level process that simulates a device driver. Because PT files are transparent, and the message module maps bijectively between device-process data and stream control

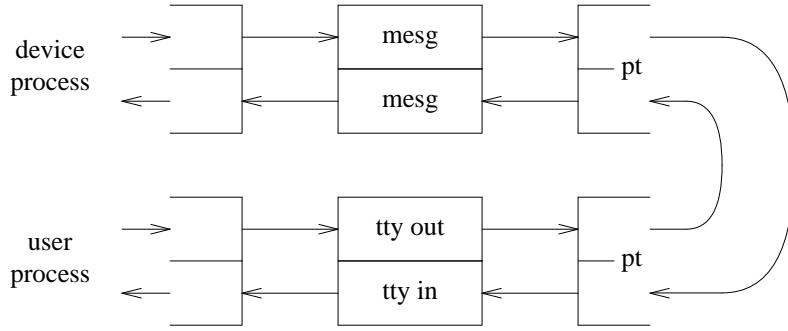


Figure 4. Configuration for device simulator.

messages, the device simulator may be completely faithful up to details of timing. In particular, user's *ioctl* requests are sent to the device process and are handled by it, even if they are not understood by the operating system.

The usefulness of this setup is not so much to simulate new devices, but to provide ways for one program to control the environment of another. Pike [6] shows how these mechanisms are used to create multiple virtual terminals on one physical terminal. In another application, inter-machine connections in which a user on one computer logs into another make use of the message module. Here the *ioctl* requests generated by programs on the remote machine are translated by this module into data messages that can be sent over the network. The local callout program translates them back into terminal control commands.

Evaluation

My intent in rewriting the character I/O system was to improve its structure by separating functions that had been intertwined, and by allowing independent modules to be connected dynamically across well-defined interfaces. I also wanted to make the system faster and smaller. The most difficult part of the project was the design of the interface. It was guided by these decisions:

- 1) It seemed to be necessary for efficiency that the objects passed between modules be references to blocks of data. The most important consequences of this principle, and those that proved deciding, are that data need not be copied as it passes across a module interface, and that many characters can be handled during a single intermodule transmission. Another effect, undesirable but accepted, is that each module must be prepared to handle discrete chunks of data of unpredictable size. For example, a protocol that expects records containing (say) an 8-byte header must be prepared to paste together smaller data blocks and split a block containing both a header and following data. A related, although not necessarily consequent, decision was to make the code assume that the data is addressable.
- 2) I decided, with regret, that each processing module could not act as an independent process with its own call record. The numbers seemed against it: on large systems it is necessary to allow for as many as 1000 queues, and I saw no good way to run this many processes without consuming inordinate amounts of storage. As a result, stream server procedures are not allowed to block awaiting data, but instead must return after saving necessary status information explicitly. The contortions required in the code are seldom serious in practice, but the beauty of the scheme would increase if servers could be written as a simple read-write loop in the true coroutine style.
- 3) The characteristic feature of the design the server and put procedures was the most difficult to work out. I began with a belief that the intermodule interface should be identical in the read and write directions. Next, I observed that a pure call model (put procedure only) would not work; queueing would be necessary at some point. For example, if the *write* system entry called through the terminal processing module to the device driver, the driver would need to queue characters internally lest output be completely synchronous. On the other hand, a pure queueing model (service procedure only; upstream modules always place their data in an input queue) also appeared impractical. As discussed

above, a module (for example terminal input) must often be activated at times that depend on its input data.

After considerable churning of details, the model presented here emerged. In general its performance by various measures lives up to hopes.

The improvement in modularity is hard to measure, but seems real; for example, the number of included header files in stream modules drops to about one half of those required by similar routines in the base system (4.1 BSD). Certainly stream modules may be composed more freely than were the "line disciplines" of older systems.

The program text size of the version of the operating system described here is about 106 kilobytes on the VAX; the base system was about 130KB. The reduction was achieved by rewriting the various device drivers and protocols and eliminating the Seventh Edition multiplexed files [1], most (though not all) of whose functions are subsumed by other mechanisms. On the other hand, the data space has increased. On a VAX 11/750 configured for 32 users about 32KB are used for storage of the structures for streams, queues, and blocks. The traditional character lists seem to require less; similar systems from Berkeley and AT&T use between 14 and 19KB. The tradeoff of program for data seems desirable.

Proper time comparisons have not been made, because of the difficulty of finding a comparable configuration. On a VAX 11/750, printing a large file on a directly-connected terminal consumes 346 microseconds per character using the system described here; this is about 10 per cent slower than the base system. On the other hand, that system's per-character interrupt routine is coded in assembly language, and the rest of its terminal handler is replete with nonportable interpolated assembly code; the current system is written completely in C. Printing the same file on a terminal connected through a primitive network interface requires 136 microseconds per character, half as much as the older network routines. Pike [7] observes that among the three implementations of Blit connection software, the one based on the stream system is the only one that can download programs at anything approaching line speed through a 19.2 Kbps connection. In general I conclude that the new organization never slows comparable tasks much, and that considerable speed improvements are sometimes possible.

Although the new organization performs well, it has several peculiarities and limitations. Some of them seem inherent, some are fixable, and some are the subject of current work.

I/O control calls turn into messages that require answers before a result can be returned to the user. Sometimes the message ultimately goes to another user-level process that may reply tardily or never. The stream is write-locked until the reply returns, in order to eliminate the need to determine which process gets which reply. A timeout breaks the lock, so there is an unjustified error return if a reply is late, and a long lockup period if one is lost. The problem can be ameliorated by working harder on it, but it typifies the difficulties that turn up when direct calls are replaced by message-passing schemes.

Several oddities appear because time spent in server routines cannot be assigned to any particular user or process. It is impossible, for example, for devices to support privileged *ioctl* calls, because the device has no idea who generated the message. Accounting and scheduling become less accurate; a short census of several systems showed that between 4 and 8 per cent of non-idle CPU time was being spent in server routines. Finally, the anonymity of server processing most certainly makes it more difficult to measure the performance of the new I/O system.

In its current form the stream I/O system is purely data-driven. That is, data is presented by a user's *write* call, and passes through to the device; conversely, data appears unbidden from a device and passes to the top level, where it is picked up by *read* calls. Wherever possible flow control throttles down fast generators of data, but nowhere except at the consumer end of a stream is there knowledge of precisely how much data is desired. Consider a command to execute possibly interactive program on another machine connected by a stream. The simplest such command sets up the connection and invokes the remote program, and then copies characters from its own standard input to the stream, and from the stream to its standard output. The scheme is adequate in practice, but breaks when the user types more than the remote program expects. For example, if the remote program reads no input at all, any typed-ahead characters are sent to the remote system and lost. This demonstrates a problem, but I know of no solution inside the stream I/O mechanism itself; other ideas will have to be applied.

Streams are linear connections; by themselves, they support no notion of multiplexing, fan-in or fan-

out. Except at the ends of a stream, each invocation of a module has a unique "next" and "previous" module. Two locally-important applications of streams testify to the importance of multiplexing: Blit terminal connections, where the multiplexing is done well, though at some performance cost, by a user program, and remote execution of commands over a network, where it is desired, but not now easy, to separate the standard output from error output. It seems likely that a general multiplexing mechanism could help in both cases, but again, I do not yet know how to design it.

Although the current design provides elegant means for controlling the semantics of communication channels already opened, it lacks general ways of establishing channels between processes. The PT files described above are just fine for Blit layers, and work adequately for handling a few administrator-controlled client-server relationships. (Yes, we have multi-machine mazewar.) Nevertheless, better naming mechanisms are called for.

In spite of these limitations, the stream I/O system works well. Its aim was to improve design rather than to add features, in the belief that with proper design, the features come cheaply. This approach is arduous, but continues to succeed.

References

1. *Unix Programmers's Manual, Seventh Edition*, Bell Laboratories, Murray Hill, NJ, (January, 1979).
2. *Unix Programmer's Manual, Virtual VAX-11 Version*, University of California, Berkeley (June 1981).
3. A. G. Fraser, "Datakit--A Modular Network for Synchronous and Asynchronous Traffic," *Proc. Int. Conf. on Communication*, Boston, MA (June 1979).
4. K. Thompson, "The Unix Time-sharing System--Unix Implementation," B.S.T.J. **57** No 6, (July-Aug 1978), pp. 1931-1946.
5. D.G. Bobrow, J.D. Burchfiel, D.L. Murphy, and R.S Tomlinson, "Tenex--a Paged Time Sharing System for the PDP-10," C. ACM **15** No. 3, (March 1972), pp. 135-143.
6. T.A. Dolotta, S.B. Olsson,, and A.G.Petrucelli, *Unix User's Manual, Release 3.0*, Bell Laboratories, Murray Hill, NJ (June 1980).
7. R. Pike, "The Blit: A Multiplexed Graphics Terminal," AT&T Tech. J. **63** No. 8 Part 2, October 1984.

The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11

CR Categories: 4.30, 4.32

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973. Authors' address: Bell Laboratories, Murray Hill, NJ 07974.

The electronic version was recreated by Eric A. Brewer, University of California at Berkeley, brewer@cs.berkeley.edu. Please notify me of any deviations from the original; I have left errors in the original unchanged.

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969–70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40 and /45 [1] system since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February 1971, about 40 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as \$40,000, and less than two man years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. It is hoped, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are: assembler, text editor based on QED [2], linking loader, symbolic debugger, compiler for a language resembling BCPL [3] with types and structures (C), interpreter for a dialect of BASIC, text formatting program, Fortran compiler, Snobol interpreter, top-down compiler-compiler (TMG) [4], bottom-up compiler-compiler (YACC), form letter generator, macro processor (M6) [5], and permuted index program.

There is also a host of maintenance, utility, recreation, and novelty programs. All of these programs were written locally. It is worth noting that the system is totally self-supporting. All UNIX software is maintained under UNIX; likewise, UNIX documents are generated and formatted by the UNIX editor and text formatting program.

2. Hardware and Software Environment

The PDP-11/45 on which our UNIX installation is implemented is a 16-bit word (8-bit byte) computer with 144K bytes of core memory; UNIX occupies 42K bytes. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the

software mentioned above can require as little as 50K bytes of core altogether.

The PDP-11 has a 1M byte fixed-head disk, used for file system storage and swapping, four moving-head disk drives which each provide 2.5M bytes on removable disk cartridges, and a single moving-head disk drive which uses removable 40M byte disk packs. There are also a high-speed paper tape reader-punch, nine-track magnetic tape, and D-tape (a variety of magnetic tape facility in which individual records may be addressed and rewritten). Besides the console typewriter, there are 14 variable-speed communications interfaces attached to 100-series datasets and a 201 dataset interface used primarily for spooling printout to a communal line printer. There are also several one-of-a-kind devices including a Picturephone® interface, a voice response unit, a voice synthesizer, a phototypesetter, a digital switching network, and a satellite PDP-11/20 which generates vectors, curves, and characters on a Tektronix 611 storage-tube display.

The greater part of UNIX software is written in the above-mentioned C language [6]. Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we considered this increase in size quite acceptable.

3. The File System

The most important job of UNIX is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary Files

A file contains whatever information the user places on it, for example symbolic or binary (object) programs. No particular structuring is expected by the system. Files of text consist simply of a string of characters, with lines demarcated by the new-line character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure: the assembler generates and the loader expects an object file in a particular format. However, the structure of files is controlled by the programs which use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his

own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the *root* directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the root. Another system directory contains all the programs provided for general use; that is, all the *commands*. As will be seen however, it is by no means necessary that a program reside in this directory for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes "/" and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name */alpha/beta/gamma* causes the system to search the root for directory *alpha*, then to search *alpha* for *beta*, finally to find *gamma* in *beta*. *Gamma* may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name *alpha/beta* specifies the file named *beta* in subdirectory *alpha* of the current directory. The simplest kind of name, for example *alpha*, refers to a file which itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same nondirectory file may appear in several directories under possibly different names. This feature is called *linking*; a directory entry for a file is sometimes called a link. UNIX differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name in each directory refers to the directory itself. Thus a program may read the current directory under the name "." without knowing its complete path name. The name ".." by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "." and "..", each directory must appear as an entry in exactly one other, which is its parent. The reason for this is to simplify the writing of programs which visit subtrees of the directory

structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special Files

Special files constitute the most unusual feature of the UNIX file system. Each I/O device supported by UNIX is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory */dev*, although a link may be made to one of these files just like an ordinary file. Thus, for example, to punch paper tape, one may write on the file */dev/ppt*. Special files exist for each communication line, each disk, each tape drive, and for physical core memory. Of course, the active disks and the core special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable File Systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a *mount* system request which has two arguments: the name of an existing ordinary file, and the name of a direct-access special file whose associated storage volume (e.g. disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of *mount* is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, *mount* replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the *mount*, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on the fixed-head disk, and the large disk drive, which contains user's files, is mounted by the system initialization program, the four smaller disk drives are available to users for mounting their own disk packs. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping which would otherwise be required to assure removal of the links when the removable volume is finally dismounted. In

particular, in the root directories of all file systems, removable or not, the name “..” refers to the directory itself instead of to its parent.

3.5 Protection

Although the access control scheme in UNIX is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of seven protection bits. Six of these specify independently read, write, and execute permission for the owner of the file and for all other users.

If the seventh bit is on, the system will temporarily change the user identification of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program which calls for it. The set-user-ID feature provides for privileged programs which may use files inaccessible to other users. For example, a program may keep an accounting file which should neither be read nor changed except by the program itself. If the set-user-identification bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands which call privileged system entries. For example, there is a system entry invocable only by the “super-user” (below) which creates an empty directory. As indicated above, directories are expected to have entries for “.” and “..”. The command which creates a directory is owned by the super user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for “.” and “..”.

Since anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed in [7].

The system recognizes one particular user ID (that of the “super-user”) as exempt from the usual constraints on file access; thus (for example) programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O Calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between “random” and sequential I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the

highest byte written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O in UNIX, Some of the basic calls are summarized below in an anonymous language which will indicate the required parameters without getting into the complexities of machine language programming. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

```
filep = open (name, flag)
```

Name indicates the name of the file. An arbitrary path name may be given. The *flag* argument indicates whether the file is to be read, written, or “updated”, that is read and written simultaneously.

The returned value *filep* is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate it.

To create a new file or completely rewrite an old one, there is a *create* system call which creates the given file if it does not exist, or truncates it to zero length if it does exist. *Create* also opens the new file for writing and, like *open*, returns a file descriptor.

There are no user-visible locks in the file system, nor is there any restriction on the number of users who may have a file open for reading or writing; although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice, difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file which another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor which makes a copy of the file being edited.

It should be said that the system has sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in such inconvenient activities as writing on the same file, creating files in the same directory or deleting each other's open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the first following byte. For each open file there is a pointer, maintained by the system, which indicates the next byte to be read or written. If *n* bytes are read or written, the pointer advances by *n* bytes.

Once a file is open, the following calls may be used:

```
n = read(filep, buffer, count)  
n = write(filep, buffer, count)
```

Up to *count* bytes are transmitted between the file specified by *filep* and the byte array specified by *buffer*. The returned value *n* is the number of bytes actually transmitted. In the *write* case, *n* is the same as *count* except under exceptional conditions like I/O errors or end of physical medium on special files; in a read, however, *n* may without error be less than *count*. If the read pointer is so near the end of the file that reading *count* characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like devices never return more than one line of input. When a *read* call returns with *n* equal to zero, it indicates the end of the file. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a typewriter by use of an escape sequence which depends on the device used.

Bytes written on a file affect only those implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is grown as needed.

To do random (direct access) I/O, it is only necessary to move the read or write pointer to the appropriate location in the file.

```
location = seek(filep, base, offset)
```

The pointer associated with *filep* is moved to a position *offset* bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on *base*. *Offset* may be negative. For some devices (e.g. paper tape and typewriters) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in *location*.

3.6.1 Other I/O Calls. There are several additional system entries having to do with I/O and with the file system which will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

4. Implementation of the File System

As mentioned in §3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its *i-number* is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry thereby found (the file's *i-node*) contains the description of the file as follows.

1. Its owner.
2. Its protection bits.
3. The physical disk or tape addresses for the file contents.
4. Its size.

5. Time of last modification
6. The number of links to the file, that is, the number of times it appears in a directory.
7. A bit indicating whether the file is a directory.
8. A bit indicating whether the file is a special file.
9. A bit indicating whether the file is “large” or “small.”

The purpose of an *open* or *create* system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the *open* or *create*. Thus the file descriptor supplied during a subsequent call to read or write the file may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made which contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is deallocated.

The space on all fixed or removable disks which contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit which depends on the device. There is space in the i-node of each file for eight device addresses. A *small* (nonspecial) file fits into eight or fewer blocks; in this case the addresses of the blocks themselves are stored. For *large* (nonspecial) files, each of the eight device addresses may point to an indirect block of 256 addresses of blocks constituting the file itself. These files may be as large as 8·256·512, or 1,048,576 (2^{20}) bytes.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last seven device address words are immaterial, and the list is interpreted as a pair of bytes which constitute an internal *device* name. These bytes specify respectively a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar typewriter interfaces.

In this environment, the implementation of the *mount* system call (§3.4) is quite straightforward. *Mount* maintains a system table whose argument is the i-number and device name of the ordinary file specified during the *mount*, and whose corresponding value is the device name of the indicated special file. This table is searched for each (i-number, device)-pair which turns up while a path name is being scanned during an *open* or *create*; if a match is found, the i-number is replaced by 1 (which is the i-number of the root

directory on all file systems), and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is immediately after return from a *read* call the data are available, and conversely after a *write* the user's workspace may be reused. In fact the system maintains a rather complicated buffering mechanism which reduces greatly the number of I/O operations required to access a file. Suppose a *write* call is made specifying transmission of a single byte.

UNIX will search its buffers to see whether the affected disk block currently resides in core memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer, and an entry is made in a list of blocks to be written. The return from the *write* call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

A program which reads or writes files in units of 512 bytes has an advantage over a program which reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. A program which is used rarely or which does no great volume of I/O may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name which is related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, since it need only scan the linearly-organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, since all directory entries for a file have equal status. Charging the owner of a file is unfair, in general, since one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. The current version of UNIX avoids the issue by not charging any fees at all.

4.1 Efficiency of the File System

To provide an indication of the overall efficiency of UNIX and of the file system in particular, timings were made of the assembly of a 7621-line program. The assembly was run alone on the machine; the total clock time was 35.9 sec, for a rate of 212 lines per sec. The time was divided as follows: 63.5 percent assembler execution time, 16.5 percent system overhead, 20.0 percent disk wait time. We will not attempt any interpretation of these figures nor any comparison with other systems, but merely note that we are generally satisfied with the overall performance of the system.

5. Processes and Images

An *image* is a computer execution environment. It includes a core image, general register values, status of open files, current directory, and the like. An image is the current state of a pseudo computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in core; during the execution of other processes it remains in core unless the appearance of an active, higher-priority process forces it to be swapped out to the fixed-head disk.

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

5.1 Processes

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by use of the *fork* system call:

```
processid = fork (label)
```

When *fork* is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, and share any open files. The new processes differ only in that one is considered the parent process: in the parent, control returns directly from the *fork*, while in the child, control is passed to location *label*. The *processid* returned by the *fork* call is the identification of the other process.

Because the return points in the parent and child process are not the same, each image existing after a *fork* may determine whether it is the parent or child process.

5.2 Pipes

Processes may communicate with related processes using the same system *read* and *write* calls that are used for file system I/O. The call

```
filep = pipe( )
```

returns a file descriptor *filep* and creates an interprocess channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the *fork* call. A *read* using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although interprocess communication via pipes is a quite valuable tool (see §6.2), it is not a completely general mechanism since the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of Programs

Another major system primitive is invoked by

```
execute(file, arg1, arg2, ..., argn)
```

which requests the system to read in and execute the program named by *file*, passing it string arguments *arg₁*, *arg₂*, ..., *arg_n*. Ordinarily, *arg₁* should be the same string as *file*, so that the program may determine the name by which it was invoked. All the code and data in the process using *execute* is replaced from the file, but open files, current directory, and interprocess relationships are unaltered. Only if the call fails, for example because *file* could not be found or because its execute-permission bit was not set, does a return take place from the *execute* primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process Synchronization

Another process control system call

```
processid = wait( )
```

causes its caller to suspend execution until one of its children has completed execution. Then *wait* returns the *processid* of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available. *Wait* may also present status from a grandchild or more distant ancestor; see §5.5.

5.5 Termination

Lastly,

```
exit (status)
```

terminates a process, destroys its image, closes its open files, and generally obliterates it. When the parent is notified through the *wait* primitive, the indicated *status* is available to the parent; if the parent has already terminated, the status is available to the grandparent, and so on. Processes

may also terminate as a result of various illegal actions or user-generated signals (§7 below).

6. The Shell

For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg₁ arg₂ ··· arg_n

The Shell splits up the command name and the arguments into separate strings. Then a file with name *command* is sought; *command* may be a path name including the “/” character to specify any file in the system. If *command* is found, it is brought into core and executed. The arguments collected by the Shell are accessible to the command. When the command is finished, the Shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file *command* cannot be found, the Shell prefixes the string */bin/* to command and attempts again to find the file. Directory */bin* contains all the commands intended to be generally used.

6.1 Standard I/O

The discussion of I/O in §3 above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the Shell, however, start off with two open files which have file descriptors 0 and 1. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's typewriter. Thus programs which wish to write informative or diagnostic information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs which wish to read messages typed by the user usually read this file.

The Shell is able to change the standard assignments of these file descriptors from the user's typewriter printer and keyboard. If one of the arguments ‘to a command is prefixed by “>”, file descriptor 1 will, for the duration of the command, refer to the file named after the “>”. For example,

ls

ordinarily lists, on the typewriter, the names of the files in the current directory. The command

ls >there

creates a file called *there* and places the listing there. Thus the argument “>*there*” means, “place output on *there*.” On the other hand,

ed

ordinarily enters the editor, which takes requests from the user via his typewriter. The command

ed <script

interprets *script* as a file of editor commands; thus “<*script*” means, “take input from *script*.”

Although the file name following “<” or “>” appears to be an argument to the command, in fact it is interpreted completely by the Shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the Shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line

ls | pr -2 | opr

ls lists the names of the files in the current directory; its output is passed to *pr*, which paginates its input with dated headings. The argument “-2” means double column. Likewise the output from *pr* is input to *opr*. This command spools its input onto a file for off-line printing.

This process could have been carried out more clumsily by

```
ls >temp1  
pr -2 <temp1 >temp2  
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the *ls* command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as *ls* to provide such a wide variety of output options.

A program such as *pr* which copies its standard input to its standard output (with processing) is called a *filter*. Some filters which we have found useful perform character transliteration, sorting of the input, and encryption and decryption.

6.3 Command Separators: Multitasking

Another feature provided by the Shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons.

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by “&”, the Shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example,

```
as source >output &
```

causes *source* to be assembled, with diagnostic output going to *output*; no matter how long the assembly takes, the Shell returns immediately. When the Shell does not wait for the completion of a command, the identification of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The “&” may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In the examples above using “&”, an output file other than the typewriter was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The Shell also allows parentheses in the above operations. For example,

```
(date; ls) >x &
```

prints the current date and time followed by a list of the current directory onto the file *x*. The Shell also returns immediately for another request.

6.4 The Shell as a Command: Command files

The Shell is itself a command, and may be called recursively. Suppose file *tryout* contains the lines

```
as source  
mv a.out testprog  
testprog
```

The *mv* command causes the file *a.out* to be renamed *testprog*. *a.out* is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the console, *source* would be assembled, the resulting program named *testprog*, and *testprog* executed. When the lines are in *tryout*, the command

```
sh <tryout
```

would cause the Shell *sh* to execute the commands sequentially.

The Shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It is

also possible to execute commands conditionally on character string comparisons or on existence of given files and to perform transfers of control within filed command sequences.

6.5 Implementation of the Shell

The outline of the operation of the Shell can now be understood. Most of the time, the Shell is waiting for the user to type a command. When the new-line character ending the line is typed, the Shell’s *read* call returns. The Shell analyzes the command line, putting the arguments in a form appropriate for *execute*. Then *fork* is called. The child process, whose code of course is still that of the Shell, attempts to perform an *execute* with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the *fork*, which is the parent process, *waits* for the child process to die. When this happens, the Shell knows the command is finished, so it types its prompt and reads the typewriter to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains “&”, the Shell merely refrains from waiting for the process which it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the *fork* primitive, it inherits not only the core image of its parent but also all the files currently open in its parent, including those with file descriptors 0 and 1. The Shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with “<” or “>” is given however, the offspring process, just before it performs *execute*, makes the standard I/O file descriptor 0 or 1 respectively refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is *opened* (or *created*); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after “<” or “>” and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the Shell need not know the actual names of the files which are its own standard input and output since it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the Shell never terminates. (The main loop includes that branch of the return from *fork* belonging to the parent process; that is, the branch which does a *wait*, then reads another command line.) The one thing which causes the Shell to terminate is discovering an end-of-file condition on its input file. Thus,

when the Shell is executed as a command with a given input file, as in

```
sh <comfile
```

the commands in *comfile* will be executed until the end of *comfile* is reached; then the instance of the Shell invoked by *sh* will terminate. Since this Shell process is the child of another instance of the Shell, the *wait* executed in the latter will return, and another command may be processed.

6.6 Initialization

The instances of the Shell to which users type commands are themselves children of another process. The last step in the initialization of UNIX is the creation of a single process and the invocation (via *execute*) of a program called *init*. The role of *init* is to create one process for each typewriter channel which may be dialed up by a user. The various subinstances of *init* open the appropriate typewriters for input and output. Since when *init* was invoked there were no files open, in each process the typewriter keyboard will receive file descriptor 0 and the printer file descriptor 1. Each process types out a message requesting that the user log in and waits, reading the typewriter, for a reply. At the outset, no one is logged in, so each process simply hangs. Finally someone types his name or other identification. The appropriate instance of *init* wakes up, receives the log-in line, and reads a password file. If the user name is found, and if he is able to supply the correct password, *init* changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an *execute* of the Shell. At this point the Shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of *init* (the parent of all the subinstances of itself which will later become Shells) does a *wait*. If one of the child processes terminates, either because a Shell found an end of file or because a user typed an incorrect name or password, this path of *init* simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another login message. Thus a user may log out simply by typing the end-of-file sequence in place of a command to the Shell.

6.7 Other Programs as Shell

The Shell as described above is designed to allow users full access to the facilities of the system since it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged.

Recall that after a user has successfully logged in by supplying his name and password, *init* ordinarily invokes the Shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after login instead of the Shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system specify that the editor *ed* is to be

used instead of the Shell. Thus when editing system users log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking UNIX programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g. chess, blackjack, 3D tic-tac-toe) available on UNIX illustrate a much more severely restricted environment. For each of these an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the Shell. People who log in as a player of one of the games find themselves limited to the game and unable to investigate the presumably more interesting offerings of UNIX as a whole.

7. Traps

The PDP-11 hardware detects a number of program faults, such as references to nonexistent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. When an illegal action is caught, unless other arrangements have been made, the system terminates the process and writes the user's image on file *core* in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs which are looping, which produce unwanted output, or about which the user has second thoughts may be halted by the use of the *interrupt* signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a core image file.

There is also a *quit* signal which is used to force a core image to be produced. Thus programs which loop unexpectedly may be halted and the core image examined without rearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by the process. For example, the Shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating point hardware, unimplemented instructions are caught, and floating point instructions are interpreted.

8. Perspective

Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any pre-defined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer

facilities, discovered a little-used system PDP-7 and set out to create a more hospitable environment. This essentially personal effort was sufficiently successful to gain the interest of the remaining author and others, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. Then in turn the 11/20 was outgrown, UNIX had proved useful enough to persuade management to invest in the PDP-11/45. Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations which influenced the design of UNIX are visible in retrospect.

First, since we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second there have always been fairly severe size constraints on the system and its software. Given the partiality antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third, nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Since all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, only tens of instructions long, which buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable from a space-efficiency standpoint to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load routines for dealing with each device with all programs, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process control scheme and command interface have proved both convenient and efficient. Since the Shell operates as an ordinary, swappable user program, it consumes no wired-down space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the Shell executes as a process which spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

8.1 Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The *fork* operation, essentially as we implemented it, was present in the Berkeley time-sharing system [8]. On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls [9] and both the name of the Shell and its general functions. The notion that the Shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX [10].

9. Statistics

The following statistics from UNIX are presented to show the scale of the system and to show how a system of this scale is used. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

9.1 Overall

72 user population
14 maximum simultaneous users
300 directories
4400 files
34000 512-byte secondary storage blocks used

of them are caused by hardware-related difficulties such as power dips and inexplicable processor interrupts to random locations. The remainder are software failures. The longest uninterrupted up time was about two weeks. Service calls average one every three weeks, but are heavily clustered. Total up time has been about 98 percent of our 24-hour, 365-day schedule.

9.2 Per day (24-hour day, 7-day week basis)

There is a “background” process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant $e - 2$, and is now generating composite pseudoprimes (base 2).

1800 commands
4.3 CPU hours (aside from background)
70 connect hours
30 different users
75 logins

9.3 Command CPU Usage (cut off at 1%)

15.7%	C compiler	1.7%	Fortran compiler
15.2%	users' programs	1.6%	remove file
11.7%	editor	1.6%	tape archive
5.8%	Shell (used as a command, including command times)	1.6%	file system consistency check
5.3%	chess	1.4%	library maintainer
3.3%	list directory	1.3%	concatenate/print files
3.1%	document formatter	1.3%	paginate and print file
1.6%	backup dumper	1.1%	print disk usage
1.8%	assembler	1.0%	copy file

9.4 Command Accesses (cut off at 1%)

15.3%	editor	1.6%	debugger
9.6%	list directory	1.6%	Shell (used as a command)
6.3%	remove file	1.5%	print disk availability
6.3%	C compiler	1.4%	list processes executing
6.0%	concatenate/print file	1.4%	assembler
6.0%	users' programs	1.4%	print arguments
3.3%	list people logged on system	1.2%	copy file
3.2%	rename/move file	1.1%	paginate and print file
3.1%	file status	1.1%	print current date/time
1.8%	library maintainer	1.1%	file system consistency check
1.8%	document formatter	1.0%	tape archive

9.5 Reliability

Our statistics on reliability are much more subjective than the others. The following results are true to the best of our combined recollections. The time span is over one year with a very early vintage 11/45.

There has been one loss of a file system (one disk out of five) caused by software inability to cope with a hardware problem causing repeated power fail traps. Files on that disk were backed up three days.

A “crash” is an unscheduled system reboot or halt. There is about one crash every other day; about two-thirds

Acknowledgments. We are grateful to R.H. Canaday, L.L. Cherry, and L.E. McMahon for their contributions to UNIX. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M.D. McIlroy, and J.F. Ossanna.

References

1. Digital Equipment Corporation. PDP-11/40 Processor Handbook, 1972, and PDP-11/45 Processor Handbook. 1971.
2. Deutsch, L.P., and Lampson, B.W. An online editor. *Comm. ACM* 10, 12 (Dec, 1967) 793–799, 803.
3. Richards, M. BCPL: A tool for compiler writing and system programming. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 557–566.
4. McClure, R.M. TMG—A syntax directed compiler. Proc. ACM 20th Nat. Conf., ACM, 1965, New York, pp. 262–274.
5. Hall, A.D. The M6 macroprocessor. Computing Science Tech. Rep. #2, Bell Telephone Laboratories, 1969.
6. Ritchie, D.M. C reference manual. Unpublished memorandum, Bell Telephone Laboratories, 1973.
7. Aleph-null. Computer Recreations. *Software Practice and Experience* 1, 2 (Apr.–June 1971), 201–204.
8. Deutsch, L.P., and Lampson, B.W. SDS 930 time-sharing system preliminary reference manual. Doc. 30.10.10, Project GENIE, U of California at Berkeley, Apr. 1965.
9. Feiertag, R.J., and Organick, E.I. The Multics input-output system. Proc. Third Symp. on Oper. Syst. Princ., Oct. 18–20, 1971, ACM, New York, pp. 35–41.
10. Bobrow, D.C., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM* 15, 3 (Mar. 1972) 135–143.

On the Security of UNIX

Dennis M. Ritchie

Bell Laboratories, Murray Hill, N. J.

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls—there may be bugs in this area, but none are known—but rather in the lack of any checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
: loop
mkdir x
chdir x
goto loop
```

Either a panic will occur because all the i-nodes on the device are used up or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

Processes are another resource on which the only limit is total exhaustion. For example, the sequence

```
command&
command&
command&
```

if continued long enough will use up all the slots in the system's process table and prevent anyone from executing any commands. Alternatively, if the commands use much core, swap space may run out, causing a panic. Incidentally, because of the implementation of process termination, the above sequence is effective in stopping the system no matter how short a time it takes each command to terminate. (The process-table slot is not freed until the terminated process is waited for; if no commands without “&” are executed, the Shell never executes a “wait.”)

It should be evident that unbounded consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user

identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to execute the file to the user himself, to members of the user's group, and to all other users. Each process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a "super user" who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Unfortunately, UNIX software is exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. This means that more or less continuous attention must be paid to adjusting modes properly. If one wants to keep one's files completely secret, it is possible to remove all permissions from the directory in which they live, which is easy and effective; but if it is desired to give general read permission while preventing writing, things are more complicated. The main problem is that write permission in a directory means precisely that; it has nothing to do with write permission for a file in that directory. Thus a writeable file in a read-only directory may be changed, or even truncated, though not removed. This fact is perfectly logical, though in this case unfortunate. A case can be made for requiring write permission for the directory of a file as well as for the file itself before allowing writing. (This possibility is more complicated than it seems at first; the system has to allow users to change their own directories while forbidding them to change the user-directory directory.)

A situation converse to the above-discussed difficulty is also present_ it is possible to delete a file if one has write permission for its directory independently of any permissions for the file. This problem is related more to self-protection than protection from others. It is largely mitigated by the fact that the two major commands which delete named files (mv and rm) ask confirmation before deleting unwritable files.

It follows from this discussion that to maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's directory inaccessible to others. The lack of sufficiency could follow

from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below).

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is feasible up to a point. As a practical test of the possibilities in this area, 67 encrypted passwords were collected from 10 UNIX installations. These were tested against all five-letter combinations, all combinations of letters and digits of length four or less, and all words in Webster's Second unabridged dictionary; 60 of the 67 passwords were found. The whole process took about 12 hours of machine time. This experience suggests that passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login:" on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives. (It is this kind of possibility that makes it evident that UNIX was not designed to be secure.)

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user (*su*) command is writable, anyone can copy the shell onto it and get a password-free version of *su*. A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. In some systems, for example, the *mail* command is set-UID and owned by the super-user. The notion is that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble is that *mail* is rather dumb: anyone can mail someone else's private file to himself. Much more serious, is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writeable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su*; other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.

C Reference Manual

Dennis M. Ritchie

*Bell Telephone Laboratories
Murray Hill, New Jersey 07974*

1. Introduction

C is a computer language based on the earlier language B [1]. The languages and their compilers differ in two major ways: C introduces the notion of types, and defines appropriate extra syntax and semantics; also, C on the PDP-11 is a true compiler, producing machine code where B produced interpretive code.

Most of the software for the UNIX time-sharing system [2] is written in C, as is the operating system itself. C is also available on the HIS 6070 computer at Murray Hill and on the IBM System/370 at Holmdel [3]. This paper is a manual only for the C language itself as implemented on the PDP-11. However, hints are given occasionally in the text of implementation-dependent features.

The UNIX Programmer's Manual [4] describes the library routines available to C programs under UNIX, and also the procedures for compiling programs under that system. "The GCOS C Library" by Lesk and Barres [5] describes routines available under that system as well as compilation procedures. Many of these routines, particularly the ones having to do with I/O, are also provided under UNIX. Finally, "Programming in C—A Tutorial," by B. W. Kernighan [6], is as useful as promised by its title and the author's previous introductions to allegedly impenetrable subjects.

2. Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore "`_`" counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

int	break
char	continue
float	if
double	else
struct	for
auto	do
extern	while
register	switch
static	case
goto	default
return	entry
sizeof	

The `entry` keyword is not currently implemented by any compiler but is reserved for future use.

2.3 Constants

There are several kinds of constants, as follows:

2.3.1 Integer constants

An integer constant is a sequence of digits. An integer is taken to be octal if it begins with 0, decimal otherwise. The digits 8 and 9 have octal value 10 and 11 respectively.

2.3.2 Character constants

A character constant is 1 or 2 characters enclosed in single quotes “‘ ’”. Within a character constant a single quote must be preceded by a back-slash “\”. Certain non-graphic characters, and “\” itself, may be escaped according to the following table:

BS	\b
NL	\n
CR	\r
HT	\t
<i>ddd</i>	\ <i>ddd</i>
\	\\

The escape “*ddd*” consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is “\0” (not followed by a digit) which indicates a null character.

Character constants behave exactly like integers (not, in particular, like objects of character type). In conformity with the addressing structure of the PDP-11, a character constant of length 1 has the code for the given character in the low-order byte and 0 in the high-order byte; a character constant of length 2 has the code for the first character in the low byte and that for the second character in the high-order byte. Character constants with more than one character are inherently machine-dependent and should be avoided.

2.3.3 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.4 Strings

A string is a sequence of characters surrounded by double quotes ““ ””. A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end. In a string, the character ““ ”” must be preceded by a “\”; in addition, the same escapes as described for character constants may be used.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in gothic. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript “*opt*,” so that

{ expression_{opt} }

would indicate an optional expression in braces.

4. What's in a Name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a function, and are discarded on return; static variables are local to a function, but retain their values independently of invocations of the function; external variables are independent of any function. Register variables are stored in the fast registers of the machine; like automatic variables they are local to each function and disappear on return.

C supports four fundamental types of objects: characters, integers, single-, and double-precision floating-point numbers.

Characters (declared, and hereinafter called, `char`) are chosen from the ASCII set; they occupy the right-most seven bits of an 8-bit byte. It is also possible to interpret `char`s as signed, 2's complement 8-bit numbers.

Integers (`int`) are represented in 16-bit 2's complement notation.

Single precision floating point (`float`) quantities have magnitude in the range approximately $10^{\pm 38}$ or 0; their precision is 24 bits or about seven decimal digits.

Double-precision floating-point (`double`) quantities have the same range as `floats` and a precision of 56 bits or about 17 decimal digits.

Besides the four fundamental types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- arrays* of objects of most types;
- functions* which return objects of a given type;
- pointers* to objects of a given type;
- structures* containing objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An object is a manipulatable region of storage; an lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if E is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which E points. The name “lvalue” comes from the assignment expression “`E1 = E2`” in which the left operand `E1` must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions.

6.1 Characters and integers

A `char` object may be used anywhere an `int` may be. In all cases the `char` is converted to an `int` by propagating its sign through the upper 8 bits of the resultant integer. This is consistent with the two's complement representation used for both characters and integers. (However, the sign-propagation feature disappears in other implementations.)

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double` by zero-padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length.

6.3 Float and double; integer and character

All `ints` and `chars` may be converted without loss of significance to `float` or `double`. Conversion of `float` or `double` to `int` or `char` takes place with truncation towards 0. Erroneous results can be expected if the magnitude of the result exceeds 32,767 (for `int`) or 127 (for `char`).

6.4 Pointers and integers

Integers and pointers may be added and compared; in such a case the `int` is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions referred to as the operands of `+` (§7.4) are those expressions defined in §§7.1_7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in an appendix.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects.

7.1 Primary expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

7.1.1 *identifier*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is “array of ...”, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is “pointer to ...”. Moreover, an array identifier is not an lvalue expression.

Likewise, an identifier which is declared “function returning ...”, when used except in the function-name position of a call, is converted to “pointer to function returning ...”.

7.1.2 *constant*

A decimal, octal, character, or floating constant is a primary expression. Its type is `int` in the first three cases, `double` in the last.

7.1.3 *string*

A string is a primary expression. Its type is originally “array of `char`”; but following the same rule as in §7.1.1 for identifiers, this is modified to “pointer to `char`” and the result is a pointer to the first character in the string.

7.1.4 (*expression*)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

7.1.5 primary-expression [expression]

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type “pointer to …”, the subscript expression is `int`, and the type of the result is “…”. The expression “`E1[E2]`” is identical (by definition) to “`*((E1)+(E2))`”. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1.1, 7.2.1, and 7.4.1 on identifiers, `*`, and `+` respectively; §14.3 below summarizes the implications.

7.1.6 primary-expression (expression-list_{opt})

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type “function returning …”, and the result of the function call is of type “…”. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` are converted to `int`.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot possibly affect the values of the actual parameters. On the other hand, it is perfectly possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points.

Recursive calls to any function are permissible.

7.1.7 primary-lvalue . member-of-structure

An lvalue expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the lvalue is assumed to have the same form as the structure containing the structure member. The result of the expression is an lvalue appropriately offset from the origin of the given lvalue whose type is that of the named structure member. The given lvalue is not required to have any particular type.

Structures are discussed in §8.5.

7.1.8 primary-expression -> member-of-structure

The primary-expression is assumed to be a pointer which points to an object of the same form as the structure of which the member-of-structure is a part. The result is an lvalue appropriately offset from the origin of the pointed-to structure whose type is that of the named structure member. The type of the primary-expression need not in fact be pointer; it is sufficient that it be a pointer, character, or integer.

Except for the relaxation of the requirement that `E1` be of pointer type, the expression “`E1->MOS`” is exactly equivalent to “`(*E1).MOS`”.

7.2 Unary operators

Expressions with unary operators group right-to-left.

7.2.1 * expression

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is “pointer to …”, the type of the result is “…”.

7.2.2 & lvalue-expression

The result of the unary `&` operator is a pointer to the object referred to by the lvalue-expression. If the type of the lvalue-expression is “…”, the type of the result is “pointer to …”.

7.2.3 - expression

The result is the negative of the expression, and has the same type. The type of the expression must be `char`, `int`, `float`, or `double`.

7.2.4 $! \ expression$

The result of the logical negation operator $!$ is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is `int`. This operator is applicable only to `ints` or `chars`.

7.2.5 $\sim \ expression$

The \sim operator yields the one's complement of its operand. The type of the expression must be `int` or `char`, and the result is `int`.

7.2.6 $\text{++ } lvalue\text{-expression}$

The object referred to by the lvalue expression is incremented. The value is the new value of the lvalue expression and the type is the type of the lvalue. If the expression is `int` or `char`, it is incremented by 1; if it is a pointer to an object, it is incremented by the length of the object. `++` is applicable only to these types. (Not, for example, to `float` or `double`.)

7.2.7 $\text{-- } lvalue\text{-expression}$

The object referred to by the lvalue expression is decremented analogously to the `++` operator.

7.2.8 $lvalue\text{-expression } \text{++}$

The result is the value of the object referred to by the lvalue expression. After the result is noted, the object referred to by the lvalue is incremented in the same manner as for the prefix `++` operator: by 1 for an `int` or `char`, by the length of the pointed-to object for a pointer. The type of the result is the same as the type of the lvalue-expression.

7.2.9 $lvalue\text{-expression } \text{--}$

The result of the expression is the value of the object referred to by the lvalue expression. After the result is noted, the object referred to by the lvalue expression is decremented in a way analogous to the postfix `++` operator.

7.2.10 `sizeof expression`

The `sizeof` operator yields the size, in bytes, of its operand. When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

7.3 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right.

7.3.1 $expression \ * \ expression$

The binary `*` operator indicates multiplication. If both operands are `int` or `char`, the result is `int`; if one is `int` or `char` and one `float` or `double`, the former is converted to `double`, and the result is `double`; if both are `float` or `double`, the result is `double`. No other combinations are allowed.

7.3.2 $expression \ / \ expression$

The binary `/` operator indicates division. The same type considerations as for multiplication apply.

7.3.3 $expression \ \% \ expression$

The binary `%` operator yields the remainder from the division of the first expression by the second. Both operands must be `int` or `char`, and the result is `int`. In the current implementation, the remainder has the same sign as the dividend.

7.4 Additive operators

The additive operators `+` and `-` group left-to-right.

7.4.1 *expression + expression*

The result is the sum of the expressions. If both operands are `int` or `char`, the result is `int`. If both are `float` or `double`, the result is `double`. If one is `char` or `int` and one is `float` or `double`, the former is converted to `double` and the result is `double`. If an `int` or `char` is added to a pointer, the former is converted by multiplying it by the length of the object to which the pointer points and the result is a pointer of the same type as the original pointer. Thus if `P` is a pointer to an object, the expression “`P+1`” is a pointer to another object of the same type as the first and immediately following it in storage.

No other type combinations are allowed.

7.4.2 *expression - expression*

The result is the difference of the operands. If both operands are `int`, `char`, `float`, or `double`, the same type considerations as for `+` apply. If an `int` or `char` is subtracted from a pointer, the former is converted in the same way as explained under `+` above.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators `<<` and `>>` group left-to-right.

7.5.1 *expression << expression*

7.5.2 *expression >> expression*

Both operands must be `int` or `char`, and the result is `int`. The second operand should be non-negative. The value of “`E1<<E2`” is `E1` (interpreted as a bit pattern 16 bits long) left-shifted `E2` bits; vacated bits are 0-filled. The value of “`E1>>E2`” is `E1` (interpreted as a two’s complement, 16-bit quantity) arithmetically right-shifted `E2` bit positions. Vacated bits are filled by a copy of the sign bit of `E1`. [Note: the use of arithmetic rather than logical shift does not survive transportation between machines.]

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; “`a<b<c`” does not mean what it seems to.

7.6.1 *expression < expression*

7.6.2 *expression > expression*

7.6.3 *expression <= expression*

7.6.4 *expression >= expression*

The operators `<` (less than), `>` (greater than), `<=` (less than or equal to) and `>=` (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the `+` operator except that pointers of any kind may be compared; the result in this case depends on the relative locations in storage of the pointed-to objects. It does not seem to be very meaningful to compare pointers with integers other than 0.

7.7 Equality operators

7.7.1 *expression == expression*

7.7.2 *expression != expression*

The `==` (equal to) and the `!=` (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus “`a<b == c<d`” is 1 whenever `a<b` and `c<d` have the same truth-value).

7.8 *expression & expression*

The `&` operator groups left-to-right. Both operands must be `int` or `char`; the result is an `int` which is the bit-wise logical and function of the operands.

7.9 *expression* \wedge *expression*

The \wedge operator groups left-to-right. The operands must be `int` or `char`; the result is an `int` which is the bit-wise exclusive `or` function of its operands.

7.10 *expression* $|$ *expression*

The $|$ operator groups left-to-right. The operands must be `int` or `char`; the result is an `int` which is the bit-wise inclusive `or` of its operands.

7.11 *expression* $\&\&$ *expression*

The $\&\&$ operator returns 1 if both its operands are non-zero, 0 otherwise. Unlike `&`, `$\&\&$` guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer.

7.12 *expression* $\|$ *expression*

The $\|$ operator returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike `|`, `$\|$` guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer.

7.13 *expression* ? *expression* : *expression*

Conditional expressions group left-to-right. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If the types of the second and third operand are the same, the result has their common type; otherwise the same conversion rules as for `+` apply. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

7.14.1 *lvalue* = *expression*

The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be `int`, `char`, `float`, `double`, or pointer. If neither operand is a pointer, the assignment takes place as expected, possibly preceded by conversion of the expression on the right.

When both operands are `int` or pointers of any kind, no conversion ever takes place; the value of the expression is simply stored into the object referred to by the lvalue. Thus it is possible to generate pointers which will cause addressing exceptions when used.

7.14.2 *lvalue* $=+$ *expression*

7.14.3 *lvalue* $=-$ *expression*

7.14.4 *lvalue* $=*$ *expression*

7.14.5 *lvalue* $=/$ *expression*

7.14.6 *lvalue* $=\%$ *expression*

7.14.7 *lvalue* $=>>$ *expression*

7.14.8 *lvalue* $=<<$ *expression*

7.14.9 *lvalue* $=\&$ *expression*

7.14.10 *lvalue* $=\wedge$ *expression*

7.14.11 *lvalue* $=|$ *expression*

The behavior of an expression of the form “`E1 =op E2`” may be inferred by taking it as equivalent to “`E1 = E1 op E2`”; however, `E1` is evaluated only once. Moreover, expressions like “`i =+ p`” in which a pointer is added to an integer, are forbidden.

7.15 expression , expression

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls (§7.1.6) and lists of initializers (§10.2).

8. Declarations

Declarations are used within function definitions to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:

decl-specifiers declarator-list_{opt} ;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of at most one type-specifier and at most one storage class specifier.

decl-specifiers:

type-specifier
sc-specifier
type-specifier sc-specifier
sc-specifier type-specifier

8.1 Storage class specifiers

The sc-specifiers are:

sc-specifier:

auto
static
extern
register

The *auto*, *static*, and *register* declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the *extern* case there must be an external definition (see below) for the given identifiers somewhere outside the function in which they are declared.

There are some severe restrictions on *register* identifiers: there can be at most 3 register identifiers in any function, and the type of a register identifier can only be *int*, *char*, or pointer (not *float*, *double*, structure, function, or array). Also the address-of operator & cannot be applied to such identifiers. Except for these restrictions (in return for which one is rewarded with faster, smaller code), register identifiers behave as if they were automatic. In fact implementations of C are free to treat *register* as synonymous with *auto*.

If the sc-specifier is missing from a declaration, it is generally taken to be *auto*.

8.2 Type specifiers

The type-specifiers are

type-specifier:

int
char
float
double
struct { type-decl-list }
struct identifier { type-decl-list }
struct identifier

The *struct* specifier is discussed in §8.5. If the type-specifier is missing from a declaration, it is generally taken to be *int*.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators.

```
declarator-list:  
    declarator  
    declarator , declarator-list
```

The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

```
declarator:  
    identifier  
    * declarator  
    declarator ( )  
    declarator [ constant-expressionopt ]  
    ( declarator )
```

The grouping in this definition is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

If a declarator has the form

* D

for D a declarator, then the contained identifier has the type “pointer to ...”, where “...” is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form

D()

then the contained identifier has the type “function returning ...”, where “...” is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

D[constant-expression]

or

D[]

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is `int`. In the second the constant 1 is used. (Constant expressions are defined precisely in §15.) Such a declarator makes the contained identifier have type “array.” If the unadorned declarator D would specify a non-array of type “...”, then the declarator “D[i]” yields a 1-dimensional array with rank i of objects of type “...”. If the unadorned declarator D would specify an n -dimensional array with rank $i_1 \times i_2 \times \dots \times i_n$, then the declarator “D[i_{n+1}]” yields an $(n+1)$ -dimensional array with rank $i_1 \times i_2 \times \dots \times i_n \times i_{n+1}$.

An array may be constructed from one of the basic types, from a pointer, from a structure, or from another array (to generate a multi-dimensional array).

Finally, parentheses in declarators do not alter the type of the contained identifier except insofar as they alter the binding of the components of the declarator.

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f( ), *fip( ), (*pfi)( );
```

declares an integer *i*, a pointer *ip* to an integer, a function *f* returning an integer, a function *fip* returning a pointer to an integer, and a pointer *pfi* to a function which returns an integer. Also

```
float fa[17], *afp[17];
```

declares an array of float numbers and an array of pointers to float numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank $3 \times 5 \times 7$. In complete detail, *x3d* is an array of three items: each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions “*x3d*”, “*x3d[i]*”, “*x3d[i][j]*”, “*x3d[i][j][k]*” may reasonably appear in an expression. The first three have type “array”, the last has type int.

8.5 Structure declarations

Recall that one of the forms for a structure specifier is

```
struct { type-decl-list }
```

The *type-decl-list* is a sequence of type declarations for the members of the structure:

```
type-decl-list:
    type-declaration
    type-declaration type-decl-list
```

A type declaration is just a declaration which does not mention a storage class (the storage class “member of structure” here being understood by context).

```
type-declaration:
    type-specifier declarator-list ;
```

Within the structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each component of a structure begins on an addressing boundary appropriate to its type. On the PDP-11 the only requirement is that non-characters begin on a word boundary; therefore, there may be 1-byte, unnamed holes in a structure, and all structures have an even length in bytes.

Another form of structure specifier is

```
struct identifier { type-decl-list }
```

This form is the same as the one just discussed, except that the identifier is remembered as the *structure tag* of the structure specified by the list. A subsequent declaration may then be given using the structure tag but without the list, as in the third form of structure specifier:

```
struct identifier
```

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure which contains an instance of itself, as distinct from a pointer to an instance of itself.

A simple example of a structure declaration, taken from §16.2 where its use is illustrated more fully, is

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has

been given, the following declaration makes sense:

```
struct tnode s, *sp;
```

which declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort.

The names of structure members and structure tags may be the same as ordinary variables, since a distinction can be made by context. However, names of tags and members must be distinct. The same member name can appear in different structures only if the two members are of the same type and if their origin with respect to their structure is the same; thus separate structures can share a common initial segment.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

```
expression ;
```

Usually expression statements are assignments or function calls.

9.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

```
compound-statement:  
{ statement-list }
```

```
statement-list:  
statement  
statement statement-list
```

9.3 Conditional statement

The two forms of the conditional statement are

```
if ( expression ) statement  
if ( expression ) statement else statement
```

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the “else” ambiguity is resolved by connecting an *else* with the last encountered *elseless if*.

9.4 While statement

The *while* statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The *do* statement has the form

```
do statement while ( expression ) ;
```

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The `for` statement has the form

```
for ( expression-1opt ; expression-2opt ; expression-3opt ) statement
```

This statement is equivalent to

```
expression-1;
while ( expression-2 ) {
    statement
    expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied `while` clause equivalent to “`while(1)`”; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

```
switch ( expression ) statement
```

The expression must be `int` or `char`. The statement is typically compound. Each statement within the statement may be labelled with case prefixes as follows:

```
case constant-expression :
```

where the constant expression must be `int` or `char`. No two of the case constants in a switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

```
default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant in an undefined order. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a `default` prefix, control passes to the prefixed statement. In the absence of a `default` prefix none of the statements in the switch is executed.

Case or `default` prefixes in themselves do not alter the flow of control.

9.8 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is to the end of the loop. More precisely, in each of the statements

```
while ( . . . ) {           do {                   for ( . . . ) {  
    . . .  
    contin: ;           . . .  
} }                     contin: ;  
                        } while ( . . . ) ;  
    }                     contin: ;  
}
```

a `continue` is equivalent to “`goto contin`”.

9.10 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;  
return ( expression ) ;
```

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto expression ;
```

The expression should be a label (§§9.12, 14.4) or an expression of type “pointer to `int`” which evaluates to a label. It is illegal to transfer to a label not located in the current function unless some extra-language provision has been made to adjust the stack correctly.

9.12 Labelled statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. More details on the semantics of labels are given in §14.4 below.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the “`}`” of a compound statement or to supply a null body to a looping statement such as `while`.

10. External definitions

A C program consists of a sequence of external definitions. External definitions may be given for functions, for simple variables, and for arrays. They are used both to declare and to reserve storage for objects. An external definition declares an identifier to have storage class `extern` and a specified type. The type-specifier (§8.2) may be empty, in which case the type is taken to be `int`.

10.1 External function definitions

Function definitions have the form

```
function-definition:  
  type-specifieropt function-declarator function-body
```

A function declarator is similar to a declarator for a “function returning ...” except that it lists the formal parameters of the function being defined.

```
function-declarator:  
  declarator ( parameter-listopt )
```

```
parameter-list:
```

identifier
identifier , parameter-list

The function-body has the form

function-body:
type-decl-list function-statement

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be declared only here.

The function-statement is just a compound statement which may have declarations at the start.

function-statement:
{ declaration-list_{opt} statement-list }

A simple example of a complete function definition is

```
int max( a, b, c )
int a, b, c;
{
    int m;
    m = ( a > b )? a : b ;
    return ( m > c? m : c );
}
```

Here “int” is the type-specifier; “max(a, b, c)” is the function-declarator; “int a, b, c;” is the type-decl-list for the formal parameters; “{ ... }” is the function-statement.

C converts all float actual parameters to double, so formal parameters declared float have their declaration adjusted to read double. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared “array of ...” are adjusted to read “pointer to ...”. Finally, because neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted).

A free return statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

10.2 External data definitions

An external data definition has the form

data-definition:
extern_{opt} type-specifier_{opt} init-declarator-list_{opt} ;

The optional extern specifier is discussed in § 11.2. If given, the init-declarator-list is a comma-separated list of declarators each of which may be followed by an initializer for the declarator.

init-declarator-list:
init-declarator
init-declarator , init-declarator-list

init-declarator:
declarator initializer_{opt}

Each initializer represents the initial value for the corresponding object being defined (and declared).

initializer:
constant
{ constant-expression-list }

```
constant-expression-list:  
    constant-expression  
    constant-expression , constant-expression-list
```

Thus an initializer consists of a constant-valued expression, or comma-separated list of expressions, inside braces. The braces may be dropped when the expression is just a plain constant. The exact meaning of a constant expression is discussed in §15. The expression list is used to initialize arrays; see below.

The type of the identifier being defined should be compatible with the type of the initializer: a double constant may initialize a float or double identifier; a non-floating-point expression may initialize an int, char, or pointer.

An initializer for an array may contain a comma-separated list of compile-time expressions. The length of the array is taken to be the maximum of the number of expressions in the list and the square-bracketed constant in the array's declarator. This constant may be missing, in which case 1 is used. The expressions initialize successive members of the array starting at the origin (subscript 0) of the array. The acceptable expressions for an array of type "array of ..." are the same as those for type "...". As a special case, a single string may be given as the initializer for an array of chars; in this case, the characters in the string are taken as the initializing values.

Structures can be initialized, but this operation is incompletely implemented and machine-dependent. Basically the structure is regarded as a sequence of words and the initializers are placed into those words. Structure initialization, using a comma-separated list in braces, is safe if all the members of the structure are integers or pointers but is otherwise ill-advised.

The initial value of any externally-defined object not explicitly initialized is guaranteed to be 0.

11. Scope rules

A complete C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

C is not a block-structured language; this may fairly be considered a defect. The lexical scope of names declared in external definitions extends from their definition through the end of the file in which they appear. The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function.

It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

11.2 Scope of externals

If a function declares an identifier to be `extern`, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and extent specified in the definition are compatible with those specified by each function which references the data.

In PDP-11 C, it is explicitly permitted for (compatible) external definitions of the same identifier to be present in several of the separately-compiled pieces of a complete program, or even twice within the same program file, with the important limitation that the identifier may be initialized in at most one of the definitions. In other operating systems, however, the compiler must know in just which file the storage for the identifier is allocated, and in which file the identifier is merely being referred to. In the implementations of C for such systems, the appearance of the `extern` keyword before an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the file where storage is allocated.

In PDP-11 C none of this nonsense is necessary and the `extern` specifier is ignored in external definitions.

12. Compiler control lines

When a line of a C program begins with the character #, it is interpreted not by the compiler itself, but by a pre-processor which is capable of replacing instances of given identifiers with arbitrary token-strings and of inserting named files into the source program. In order to cause this preprocessor to be invoked, it is necessary that the very first line of the program begin with #. Since null lines are ignored by the preprocessor, this line need contain no other information.

12.1 Token replacement

A compiler-control line of the form

```
# define identifier token-string
```

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens (except within compiler control lines). The replacement token-string has comments removed from it, and it is surrounded with blanks. No rescanning of the replacement string is attempted. This facility is most valuable for definition of “manifest constants”, as in

```
# define tabsize 100
...
int table[tabsize];
```

12.2 File inclusion

Large C programs often contain many external data definitions. Since the lexical scope of external definitions extends to the end of the program file, it is good practice to put all the external definitions for data at the start of the program file, so that the functions defined within the file need not repeat tedious and error-prone declarations for each external identifier they use. It is also useful to put a heavily used structure definition at the start and use its structure tag to declare the `auto` pointers to the structure used within functions. To further exploit this technique when a large C program consists of several files, a compiler control line of the form

```
# include "filename"
```

results in the replacement of that line by the entire contents of the file *filename*.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. Sometimes the storage class is supplied by the context: in external definitions, and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, since `auto` functions are meaningless (C being incapable of compiling code into the stack). If the type of an identifier is “function returning ...”, it is implicitly declared to be `extern`.

In an expression, an identifier followed by (and not currently declared is contextually declared to be “function returning `int`”.

Undefined identifiers not followed by (are assumed to be labels which will be defined later in the function. (Since a label is not an lvalue, this accounts for the “Lvalue required” error message sometimes noticed when an undeclared identifier is used.) Naturally, appearance of an identifier as a label declares it as such.

For some purposes it is best to consider formal parameters as belonging to their own storage class. In practice, C treats parameters as if they were automatic (except that, as mentioned above, formal parameter arrays and `floats` are treated specially).

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures

There are only two things that can be done with a structure: pick out one of its members (by means of the `.` or `->` operators); or take its address (by unary `&`). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f( );
...
g( f );
```

Then the definition of `g` might read

```
g( funcp )
int (*funcp)( );
{
    ...
    (*funcp)( );
    ...
}
```

Notice that `f` was declared explicitly in the calling routine since its first appearance was not followed by `.`

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator `[]` is interpreted in such a way that “`E1[E2]`” is identical to “`*((E1)+(E2))`”. Because of the conversion rules which apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If `E` is an n -dimensional array of rank $i \times j \times \dots \times k$, then `E` appearing in an expression is converted to a pointer to an $(n-1)$ -dimensional array with rank $j \times \dots \times k$. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$ -dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression “`x[i]`”, which is equivalent to “`*((x)+(i))`”, `x` is first converted to a pointer as described; then `i` is converted to the type of `x`, which involves multiplying `i` by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

14.4 Labels

Labels do not have a type of their own; they are treated as having type “array of `int`”. Label variables should be declared “pointer to `int`”; before execution of a `goto` referring to the variable, a label (or an expression deriving from a label) should be assigned to the variable.

Label variables are a bad idea in general; the `switch` statement makes them almost always unnecessary.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and `sizeof` expressions, possibly connected by the binary operators

`+ - * / % & | ^ << >>`

or by the unary operators

`- ~`

Parentheses can be used for grouping, but not for function calls.

A bit more latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external scalars, and to external arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted external arrays. The rule here is that initializers must evaluate either to a constant or to the address of an external identifier plus or minus a constant.

16. Examples.

These examples are intended to illustrate some typical C constructions as well as a serviceable style of writing C programs.

16.1 Inner product

This function returns the inner product of its array arguments.

```
double inner(v1, v2, n)
double v1[], v2[];
{
    double sum;
    int i;
    sum = 0.0;
    for (i=0; i<n; i++)
        sum += v1[i] * v2[i];
    return (sum);
}
```

The following version is somewhat more efficient, but perhaps a little less clear. It uses the facts that parameter arrays are really pointers, and that all parameters are passed by value.

```
double inner(v1, v2, n)
double *v1, *v2;
{
    double sum;
    sum = 0.0;
    while (n--)
        sum += *v1++ * *v2++;
    return (sum);
}
```

The declarations for the parameters are really exactly the same as in the last example. In the first case array declarations “`[]`” were given to emphasize that the parameters would be referred to as arrays; in the second, pointer declarations were given because the indirection operator and `++` were used.

16.2 Tree and character processing

Here is a complete C program (courtesy of R. Haight) which reads a document and produces an alphabetized list of words found therein together with the number of occurrences of each word. The method keeps a binary tree of words such that the left descendant tree for each word has all the words lexicographically smaller than the given word, and the right descendant has all the larger words. Both the insertion and the printing routine are recursive.

The program calls the library routines `getchar` to pick up characters and `exit` to terminate execution. `Printf` is

called to print the results according to a format string. A version of *printf* is given below (§16.3).

Because all the external definitions for data are given at the top, no *extern* declarations are necessary within the functions. To stay within the rules, a type declaration is given for each non-integer function when the function is used before it is defined. However, since all such functions return pointers which are simply assigned to other pointers, no actual harm would result from leaving out the declarations; the supposedly *int* function values would be assigned without error or complaint.

```
# define nwords 100                                /* number of different words */
# define wsize 20                                   /* max chars per word */
struct tnode {                                      /* the basic structure */
    char tword[wsize];
    int count;
    struct tnode *left;
    struct tnode *right;
};

struct tnode space[ nwords ];                         /* the words themselves */
int nnodes nwords;                                  /* number of remaining slots */
struct tnode *spacep space;                          /* next available slot */
struct tnode *freep;                                /* free list */
/*
 * The main routine reads words until end-of-file ('\'0' returned from "getchar")
 * "tree" is called to sort each word into the tree.
 */
main( )
{
    struct tnode *top, *tree( );
    char c, word[wsize];
    int i;

    i = top = 0;
    while (c=getchar( ))
        if ( 'a'<=c && c<='z' || 'A'<=c && c <='Z' ) {
            if ( i<wsize-1 )
                word[ i++ ] = c;
        } else
            if ( i ) {
                word[ i++ ] = '\0';
                top = tree( top, word );
                i = 0;
            }
    tprint( top );
}
/*
 * The central routine. If the subtree pointer is null, allocate a new node for it.
 * If the new word and the node's word are the same, increase the node's count.
 * Otherwise, recursively sort the word into the left or right subtree according
 * as the argument word is less or greater than the node's word.
 */
struct tnode *tree( p, word )
struct tnode *p;
char word[ ];
{
    struct tnode *alloc( );
    int cond;

    /* Is pointer null? */
    if ( p==0 ) {
        p = alloc( );
        if ( word[ 0 ] < p->tword[ 0 ] )
            p->left = tree( p, word );
        else
            p->right = tree( p, word );
        p->count++;
    }
}
```

```

        copy( word, p->tword ) ;
        p->count = 1 ;
        p->right = p->left = 0 ;
        return( p ) ;
    }
    /* Is word repeated? */
    if ( ( cond=compar( p->tword, word ) ) == 0 ) {
        p->count++ ;
        return( p ) ;
    }
    /* Sort into left or right */
    if ( cond<0 )
        p->left = tree( p->left, word ) ;
    else
        p->right = tree( p->right, word ) ;
    return( p ) ;
}
/*
 * Print the tree by printing the left subtree, the given node, and the right subtree.
 */
tprint( p )
struct tnode *p ;
{
    while ( p ) {
        tprint( p->left ) ;
        printf( "%d: %s\n", p->count, p->tword ) ;
        p = p->right ;
    }
}
/*
 * String comparison: return number ( >, =, < ) 0
 * according as s1 ( >, =, < ) s2.
 */
compar( s1, s2 )
char *s1, *s2 ;
{
    int c1, c2 ;
    while( ( c1 = *s1++ ) == ( c2 = *s2++ ) )
        if ( c1=='\0' )
            return( 0 ) ;
    return( c2-c1 ) ;
}
/*
 * String copy: copy s1 into s2 until the null
 * character appears.
 */
copy( s1, s2 )
char *s1, *s2 ;
{
    while( *s2++ = *s1++ ) ;
}
/*
 * Node allocation: return pointer to a free node.
 * Bomb out when all are gone. Just for fun, there
 * is a mechanism for using nodes that have been
 * freed, even though no one here calls "free."
 */
struct tnode *alloc( )

```

```

{
    struct tnode *t;
    if (freep) {
        t = freep;
        freep = freep->left;
        return(t);
    }
    if (--nnodes < 0) {
        printf("Out of space\n");
        exit();
    }
    return(spacep++);
}
/*
 * The uncalled routine which puts a node on the free list.
 */
free(p)
struct tnode *p;
{
    p->left = freep;
    freep = p;
}

```

To illustrate a slightly different technique of handling the same problem, we will repeat fragments of this example with the tree nodes treated explicitly as members of an array. The fundamental change is to deal with the subscript of the array member under discussion, instead of a pointer to it. The `struct` declaration becomes

```

struct tnode {
    char tword[wsize];
    int count;
    int left;
    int right;
};

```

and `alloc` becomes

```

alloc( )
{
    int t;
    t = --nnodes;
    if (t<=0) {
        printf("Out of space\n");
        exit();
    }
    return(t);
}

```

The `free` stuff has disappeared because if we deal with exclusively with subscripts some sort of map has to be kept, which is too much trouble.

Now the `tree` routine returns a subscript also, and it becomes:

```

tree(p, word)
char word[ ];
{
    int cond;
    if (p==0) {
        p = alloc();
        copy(word, space[p].tword);
    }
}

```

```

        space[ p ].count = 1;
        space[ p ].right = space[ p ].left = 0;
        return( p );
    }
    if ( ( cond=compar( space[ p ].tword, word ) ) == 0 ) {
        space[ p ].count++;
        return( p );
    }
    if ( cond<0 )
        space[ p ].left = tree( space[ p ].left, word );
    else
        space[ p ].right = tree( space[ p ].right, word );
    return( p );
}

```

The other routines are changed similarly. It must be pointed out that this version is noticeably less efficient than the first because of the multiplications which must be done to compute an offset in *space* corresponding to the subscripts.

The observation that subscripts (like “*a[i]*”) are less efficient than pointer indirection (like “**ap*”) holds true independently of whether or not structures are involved. There are of course many situations where subscripts are indispensable, and others where the loss in efficiency is worth a gain in clarity.

16.3 Formatted output

Here is a simplified version of the *printf* routine, which is available in the C library. It accepts a string (character array) as first argument, and prints subsequent arguments according to specifications contained in this format string. Most characters in the string are simply copied to the output; two-character sequences beginning with “%” specify that the next argument should be printed in a style as follows:

%d	decimal number
%o	octal number
%c	ASCII character, or 2 characters if upper character is not null
%s	string (null-terminated array of characters)
%f	floating-point number

The actual parameters for each function call are laid out contiguously in increasing storage locations; therefore, a function with a variable number of arguments may take the address of (say) its first argument, and access the remaining arguments by use of subscripting (regarding the arguments as an array) or by indirection combined with pointer incrementation.

If in such a situation the arguments have mixed types, or if in general one wishes to insist that an lvalue should be treated as having a given type, then *struct* declarations like those illustrated below will be useful. It should be evident, though, that such techniques are implementation dependent.

Printf depends as well on the fact that *char* and *float* arguments are widened respectively to *int* and *double*, so there are effectively only two sizes of arguments to deal with. *Printf* calls the library routines *putchar* to write out single characters and *ftoa* to dispose of floating-point numbers.

```

printf( fmt, args )
char fmt[ ];
{
    char *s;
    struct { char **charpp; };
    struct { double *doublep; };
    int *ap, x, c;

    ap = &args; /* argument pointer */
    for ( ; ; ) {
        while( ( c = *fmt++ ) != '%' ) {
            if ( c == '\0' )
                return;

```

```

        putchar( c ) ;
    }
switch ( c = *fmt++ ) {
/* decimal */
case 'd':
    x = *ap++;
    if( x < 0 ) {
        x = -x;
        if( x<0 ) { /* is - infinity */
            printf( "-32768" );
            continue;
        }
        putchar( '-' );
    }
    printd( x );
    continue;
/* octal */
case 'o':
    printo( *ap++ );
    continue;
/* float, double */
case 'f':
    /* let ftoa do the real work */
    ftoa( *ap.doublep++ );
    continue;
/* character */
case 'c':
    putchar( *ap++ );
    continue;
/* string */
case 's':
    s = *ap.charpp++;
    while( c = *s++ )
        putchar( c );
    continue;
}
putchar( c );
}
/*
 * Print n in decimal; n must be non-negative
 */
printd( n )
{
    int a;
    if ( a=n/10 )
        printd( a );
    putchar( n%10 + '0' );
}
/*
 * Print n in octal, with exactly 1 leading 0
 */
printo( n )
{
    if ( n )
        printo( (n>>3)&017777 );
    putchar( (n&07)+'0' );
}

```

REFERENCES

1. Johnson, S. C., and Kernighan, B. W. "The Programming Language B." Comp. Sci. Tech. Rep. #8., Bell Laboratories, 1972.
2. Ritchie, D. M., and Thompson, K. L. "The UNIX Time-sharing System." C. ACM 7 , 17, July, 1974, pp. 365-375.
3. Peterson, T. G., and Lesk, M. E. "A User's Guide to the C Language on the IBM 370." Internal Memorandum, Bell Laboratories, 1974.
4. Thompson, K. L., and Ritchie, D. M. *UNIX Programmer's Manual*. Bell Laboratories, 1973.
5. Lesk, M. E., and Barres, B. A. "The GCOS C Library." Internal memorandum, Bell Laboratories, 1974.
6. Kernighan, B. W. "Programming in C- A Tutorial." Unpublished internal memorandum, Bell Laboratories, 1974.

APPENDIX 1

Syntax Summary

1. Expressions.

expression:

```

primary
* expression
& expression
- expression
! expression
~expression
++ lvalue
-- lvalue
lvalue ++
lvalue --
sizeof expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

```

primary:

```

identifier
constant
string
( expression )
primary ( expression-listopt )
primary [ expression ]
lvalue . identifier
primary --> identifier

```

lvalue:

```

identifier
primary [ expression ]
lvalue . identifier
primary --> identifier
* expression
( lvalue )

```

The primary-expression operators

() [] . —>

have highest priority and group left-to-right. The unary operators

& - ! ~ ++ -- sizeof

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators and the conditional operator all group left-to-right, and have priority decreasing as indicated:

binop:

*	/	%
+	-	
>>	<<	
<	>	<= >=
==	!=	
&		

```

^
|
&&
||
?   :

```

Assignment operators all have the same priority, and all group right-to-left.

asgnop:

```
=  =+  ==  *=  =/  =%  =>>  =<<  =&  =^  =|
```

The comma operator has the lowest priority, and groups left-to-right.

2. Declarations.

declaration:

```
decl-specifiers declarator-listopt ;
```

decl-specifiers:

```
type-specifier
sc-specifier
type-specifier sc-specifier
sc-specifier type-specifier
```

sc-specifier:

```
auto
static
extern
register
```

type-specifier:

```
int
char
float
double
struct { type-decl-list }
struct identifier { type-decl-list }
struct identifier
```

declarator-list:

```
declarator
declarator , declarator-list
```

declarator:

```
identifier
* declarator
declarator ( )
declarator [ constant-expressionopt ]
( declarator )
```

type-decl-list:

```
type-declaration
type-declaration type-decl-list
```

type-declaration:

```
type-specifier declarator-list ;
```

3. Statements.

statement:

```
expression ;
{ statement-list }
```

```
if ( expression ) statement
if ( expression ) statement else statement
while ( expression ) statement
for ( expressionopt; expressionopt; expressionopt ) statement
switch ( expression ) statement
case constant-expression : statement
default : statement
break ;
continue ;
return ;
return ( expression ) ;
goto expression ;
identifier : statement
;

statement-list:
statement
statement statement-list
```

4. External definitions.

```
program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
type-specifieropt function-declarator function-body

function-declarator:
declarator ( parameter-listopt )

parameter-list:
identifier
identifier , parameter-list

function-body:
type-decl-list function-statement

function-statement:
{ declaration-listopt statement-list }

data-definition:
externopt type-specifieropt init-declarator-listopt ;

init-declarator-list:
init-declarator
init-declarator , init-declarator-list

init-declarator:
declarator initializeropt

initializer:
constant
{ constant-expression-list }
```

```
constant-expression-list:  
    constant-expression  
    constant-expression , constant-expression-list  
  
constant-expression:  
    expression
```

5. Preprocessor

```
# define identifier token-string  
# include "filename"
```

APPENDIX 2

Implementation Peculiarities

This Appendix briefly summarizes the differences between the implementations of C on the PDP-11 under UNIX and on the HIS 6070 under GCOS; it includes some known bugs in each implementation. Each entry is keyed by an indicator as follows:

- h hard to fix
- g GCOS version should probably be changed
- u UNIX version should probably be changed
- d Inherent difference likely to remain

This list was prepared by M. E. Lesk, S. C. Johnson, E. N. Pinson, and the author.

A. Bugs or differences from C language specifications

- hg A.1) GCOS does not do type conversions in “?:”.
- hg A.2) GCOS has a bug in `int` and `real` comparisons; the numbers are compared by subtraction, and the difference must not overflow.
- g A.3) When `x` is a `float`, the construction “`test ? -x : x`” is illegal on GCOS.
- hg A.4) “`p1->p2 =+ 2`” causes a compiler error, where `p1` and `p2` are pointers.
- u A.5) On UNIX, the expression in a `return` statement is *not* converted to the type of the function, as promised.
- hug A.6) `entry` statement is not implemented at all.

B. Implementation differences

- d B.1) Sizes of character constants differ; UNIX: 2, GCOS: 4.
- d B.2) Table sizes in compilers differ.
- d B.3) `chars` and `ints` have different sizes; `chars` are 8 bits on UNIX, 9 on GCOS; words are 16 bits on UNIX and 36 on GCOS. There are corresponding differences in representations of `floats` and `doubles`.
- d B.4) Character arrays stored left to right in a word in GCOS, right to left in UNIX.
- g B.5) Passing of floats and doubles differs; UNIX passes on stack, GCOS passes pointer (hidden to normal user).
- g B.6) Structures and strings are aligned on a word boundary in UNIX, not aligned in GCOS.
- g B.7) GCOS preprocessor supports `#rename`, `#escape`; UNIX has only `#define`, `#include`.
- u B.8) Preprocessor is not invoked on UNIX unless first character of file is ‘#’.
- u B.9) The external definition “`static int ...`” is legal on GCOS, but gets a diagnostic on UNIX. (On GCOS it means an identifier global to the routines in the file but invisible to routines compiled separately.)
- g B.10) A compound statement on GCOS must contain one “;” but on UNIX may be empty.
- g B.11) On GCOS case distinctions in identifiers and keywords are ignored; on UNIX case is significant everywhere, with keywords in lower case.

C. Syntax Differences

- g C.1) UNIX allows broader classes of initialization; on GCOS an initializer must be a constant, name, or string. Similarly, GCOS is much stickier about wanting braces around initializers and in particular they must be present for array initialization.
- g C.2) “`int extern`” illegal on GCOS; must have “`extern int`” (storage class before type).
- g C.3) Externals on GCOS must have a type (not defaulted to `int`).
- u C.4) GCOS allows initialization of internal `static` (same syntax as for external definitions).
- g C.5) `integer->...` is not allowed on GCOS.
- g C.6) Some operators on pointers are illegal on GCOS (<, >).

- g C.7) register storage class means something on UNIX, but is not accepted on GCOS.
- g C.8) Scope holes: “int x; f() {int x;}” is illegal on UNIX but defines two variables on GCOS.
- g C.9) When function names are used as arguments on UNIX, either “fname” or “&fname” may be used to get a pointer to the function; on GCOS “&fname” generates a doubly-indirect pointer. (Note that both are wrong since the “&” is supposed to be supplied for free.)

D. Operating System Dependencies

- d D.1) GCOS allocates external scalars by SYMREF; UNIX allocates external scalars as labelled common; as a result there may be many uninitialized external definitions of the same variable on UNIX but only one on GCOS.
- d D.2) External names differ in allowable length and character set; on UNIX, 7 characters and both cases; on GCOS 6 characters and only one case.

E. Semantic Differences

- hg E.1) “int i, *p; p=i; i=p;” does nothing on UNIX, does something on GCOS (destroys right half of i).
- d E.2) “>>” means arithmetic shift on UNIX, logical on GCOS.
- d E.3) When a char is converted to integer, the result is always positive on GCOS but can be negative on UNIX.
- d E.4) Arguments of subroutines are evaluated left-to-right on GCOS, right-to-left on UNIX.

◆ The Inferno™ Operating System

*Sean M. Dorward, Rob Pike, David Leo Presotto,
Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom*

The Inferno™ operating system facilitates the creation and support of distributed services in the new and emerging world of network environments, such as those typified by CATV and direct satellite broadcasting systems, as well as the Internet. In addition, as the entertainment, telecommunications, and computing industries converge and interconnect, different types of data networks are arising, each one as potentially useful and profitable as the telephone network. However, unlike the telephone system, which started with standard terminals and signaling, these new networks are developing in a world of diverse terminals, network hardware, and protocols. Inferno is designed so that it can insulate the diverse providers of content and services from the equally varied transport and presentation platforms. The Inferno Business Unit of Lucent Technologies and the Computing Sciences Research Center of Bell Labs, the R&D arm of Lucent, designed it specifically as a commercial product. It is intended for licensing in the marketplace and for use in conjunction with new Lucent offerings. Inferno incorporates many years of Bell Labs research in operating systems, languages, on-the-fly compilers, graphics, security, networking, and portability in providing an effective and economical network operating system.

Introduction

The Inferno™ operating system¹ is designed to be used in a variety of network environments—for example, those supporting advanced telephones, hand-held devices, TV set-top boxes attached to cable or satellite systems, and inexpensive Internet computers—but also in conjunction with traditional computing systems.

Among the most visible new environments are CATV, direct satellite broadcasting, and the Internet. As the entertainment, telecommunications, and computing industries converge and interconnect, data networks in various forms are emerging, each potentially as useful and profitable as the telephone system. Unlike the telephone system, which started with standard terminals and signaling, these networks are developing in a world of diverse terminals, network hardware, and protocols. Only a well-designed and economical operating system can insulate the diverse

providers of content and services from the equally varied transport and presentation platforms. Inferno is a network operating system for this new world.

Inferno's definitive strength lies in the following areas:

- *Portability across processors.* Currently, Inferno runs on Intel, SPARC,* MIPS, ARM, HP-PA, Power PC,* and AMD 29K* architectures and is readily portable to others.
- *Portability across environments.* Inferno runs as a stand-alone operating system on small terminals and also as a user application under the Windows NT,* Windows 95,* UNIX* (Irix,* Solaris,* Linux, AIX,* HP/UX,* NetBSD), and Plan 9™ systems. In all these environments, Inferno applications see an identical interface.
- *Distributed design.* The identical environment is established at both a user's terminal and a

Panel 1. Abbreviations, Acronyms, and Terms

ATM—asynchronous transfer mode
CA—certifying authority
CATV—cable television
DESCBC—DES chain block coding
DES—Data Encryption Standard
DESECB—DES electronic code book
GIF—Graphics Interchange Format
I/O—input/output
IP—Internet Protocol
JPEG—Joint Photographic Experts Group
MPEG—Motion Picture Experts Group
OA&M—operations, administration, and maintenance
POTS—"plain old telephone service"
PPP—Point-to-Point Protocol
SHA—Secure Hash Algorithm
SSL—Secure Sockets Layer
STS—Station-to-Station
TCP/IP—Transmission Control Protocol/Internet Protocol

server, and each environment may import the resources of the other (for example, the attached I/O devices or networks). Aided by the communications facilities of the run-time system, applications may be split easily (and even dynamically) between client and server.

- *Minimal hardware requirements.* Inferno runs useful applications as stand-alone programs on machines with as little as 1 MB of memory, and it does not require memory-mapping hardware.
- *Portable applications.* Inferno applications are written in the type-safe Limbo™ language, whose binary representation is identical over all platforms.
- *Dynamic adaptability.* Depending on the hardware or other resources available, applications may load different program modules to perform a specific function. For example, a video player application might use any of several different decoder modules.

Underlying the design of Inferno is a model of the diversity-of-application areas it intends to stimulate. Many suppliers are interested in purveying media and services—telephone network service providers, Web

servers, cable companies, merchants, and various information services firms. Currently, many connection technologies are available—for example, ordinary telephone modems, ISDN, ATM, the Internet, analog broadcast TV or CATV, cable modems, digital video on demand, and other interactive TV systems.

Applications more clearly related to Lucent Technologies' current and planned product offerings include control of switches and routers and the associated operations system facilities needed to support them. For example, Inferno software will control an IP switch/router for voice and data being developed by Lucent's Bell Labs Research and Network Systems organizations. An Inferno-based firewall called Signet is being used to secure outside access to the research organization's Internet connection.

Finally, existing or potential hardware endpoints must be considered. Some are located in consumers' homes in the form of PCs, game consoles, and newer set-top boxes. Others are located inside the networks themselves in the form of nodes for billing, network monitoring, or provisioning. The higher ends of these spectra, such as fully interactive TV with video on demand, may be fascinating, but they have developed more slowly than expected. One reason is the cost of the set-top box—especially its memory requirements. Portable terminals are similarly constrained because of weight and cost considerations.

Inferno is parsimonious enough in its resource requirements to support interesting applications on today's hardware while being versatile enough to grow into the future. In particular, it enables developers to create applications that will work across a range of facilities. An example of such an application is an interactive shopping catalog that works in text mode over a POTS modem, shows still pictures (perhaps with audio) of the merchandise over ISDN, and includes video clips over digital cable.

Clearly, not everyone who deploys an Inferno-based solution will want to span the whole range of possibilities. However, the system architecture should be constrained only by the desired markets and the available interconnection and server technologies, not by the software.

Inferno Interfaces

The role of the Inferno system is to *create* several standard interfaces for its applications:

- Applications use various resources internal to the system. These resources include a consistent virtual machine that runs the application programs together with library modules that perform services as simple as string manipulation through more sophisticated graphics services for dealing with text, pictures, higher-level toolkits, and video.
- Applications exist in an external environment containing such resources as data files that can be read and manipulated, together with objects that are named and manipulated like files but are more active. Devices (for example a handheld remote control, an MPEG decoder, or a network interface) present themselves to the application as files.
- Standard protocols exist for communication within and between separate machines running Inferno so that applications can cooperate.

At the same time, Inferno *uses* interfaces supplied by an existing environment, either bare hardware or standard operating systems and protocols.

Typically, an Inferno-based service would consist of many relatively inexpensive terminals running Inferno as a native system and fewer large machines running it as a hosted system. On these server machines, Inferno might interface to databases, transaction systems, existing OA&M facilities, and other resources provided under the native operating system. The Inferno applications themselves would run either on the client or server machines or on both.

External Environment of Inferno Applications

The purpose of most Inferno applications is to present information or media to the user. Thus, applications must locate the information sources in the network and construct a local representation of them. The information flow, however, is not one way. The user's terminal (whether it is a network computer, TV set-top box, PC, or videophone) is also an information source, and its devices represent resources to applications. Inferno draws heavily on the design of the

Plan 9 operating system² in the way it presents resources to these applications.

The design has three principles:

- All resources are named and accessed like files in a forest of hierarchical file systems.
- The disjoint resource hierarchies provided by different services are joined into a single private and hierarchical *name space*.
- A communication protocol called Styx is applied uniformly to access the resources regardless of whether they are local or remote.

In practice, most applications see a fixed set of files organized as a directory tree. Some of the files contain ordinary data but others represent more active resources. Devices are represented as files, and device drivers (such as modems, MPEG decoders, network interfaces, or TV screens) attached to a particular hardware box present themselves as small directories. These directories typically contain two files, `data` and `ctl`, which respectively perform actual device I/O and control operations. System services also live behind filenames. For example, an Internet domain name server might be attached to an agreed-on name (say `/net/dns`). After writing to this file, which is a string representing a symbolic Internet domain name, a subsequent read from the file would return the corresponding numeric Internet address.

The glue that connects the separate parts of the resource name space together is the Styx protocol. Within an instance of Inferno, all the device drivers and other internal resources respond to the procedural version of Styx. The Inferno kernel implements a *mount driver* that transforms file system operations into remote procedure calls for transport over a network. On the other side of the connection, a server unwraps the Styx messages and implements them using resources local to it. Therefore, it is possible to import parts of the name space (and thus resources) from other machines.

To extend the example above, it is unlikely that a set-top box would store the code needed for an Internet domain name-server within itself. Instead, an Internet browser would import the `/net/dns` resource into its own name space from a server machine across a network.

The Styx protocol lies above and is independent of the communications transport layer. It is readily carried over the TCP/IP, PPP, ATM, or various modem transport protocols.

Internal Environment of Inferno Applications

Inferno applications are written in Limbo,³ a new language that was designed specifically for the Inferno environment. Its syntax is influenced by C and Pascal, and it supports the standard data types common to them together with several higher-level data types, such as lists, tuples (groups of values), strings, dynamic arrays, and simple abstract data types.

In addition, Limbo supplies several advanced constructs, which are carefully integrated into the Inferno virtual machine. In particular, a communication mechanism called a *channel* is used to connect different Limbo tasks on the same machine or across the network. A channel transports typed data in a machine-independent fashion so that complex data structures (including channels themselves) may be passed between Limbo tasks or attached to files in the name space for language-level communication between machines.

Multi-tasking is supported directly by the Limbo language. Independently scheduled threads of control may be spawned, and an `alt` statement is used to coordinate the channel communication between tasks (that is, `alt` is used to select one of several channels that are ready to communicate). By building channels and tasks into the language and its virtual machine, Inferno encourages a communication style that is safe and easy to use.

Limbo programs are built of *modules*, which are self-contained units having a well-defined interface containing functions (methods), abstract data types, and constants defined by the module and visible outside it. Modules are accessed dynamically—that is, when one module wishes to make use of another, it dynamically executes a `load` statement naming the desired module and it uses a returned handle to access the new module. When the module is no longer in use, its storage and code will be released. The flexibility of the modular structure contributes to the smallness of typical Inferno applications, as well

as to their adaptability. For example, in the shopping catalog described above, the application's main module checks dynamically for the existence of the video resource. If it is not available, the video-decoder module is never loaded.

Limbo is fully type-checked at both compile and run time. For example, pointers—besides being more restricted than in C—are checked before being dereferenced, and the type-consistency of a dynamically loaded module is checked when it is loaded. Limbo programs run safely on a machine without memory-protection hardware. Moreover, all Limbo data and program objects are subject to a garbage collector built deeply into the Limbo run-time system. All system data objects are tracked by the virtual machine and freed as soon as they become idle. For example, if an application task creates a graphics window and then terminates, the window automatically disappears the instant the last reference to it goes away.

Limbo programs are compiled into byte codes representing instructions for a virtual machine called DisTM. The architecture of the arithmetic part of Dis is a simple three-address machine supplemented with a few specialized operations for handling some of the higher-level data types, such as arrays and strings. Garbage collection is handled below the level of the machine language. Task scheduling is similarly hidden. When loaded into memory for execution, the byte codes are expanded into a format more efficient for execution. In addition, an optional on-the-fly compiler turns a Dis instruction stream into native machine instructions for the appropriate real hardware. This can be done efficiently because Dis instructions closely match the instruction-set architecture of today's machines. The resulting code executes at a speed approaching that of compiled C.

Underlying Dis is the Inferno kernel, which contains both the interpreter and on-the-fly compiler, as well as memory management, scheduling, device drivers, protocol stacks, and the like. The kernel also contains the core of the file system (the name evaluator and the code that turns file system operations into remote procedure calls over communications links) and the small file systems implemented internally.

Finally, the Inferno virtual machine implements

several standard modules internally. These modules include `Sys`, which provides system calls and a small library of useful routines (for example, creation of network connections and string manipulations). Module `Draw` is a basic graphics library that handles raster graphics, fonts, and windows. Module `Prefab` builds on `Draw` to provide structured complexes containing images and text inside windows. These elements may be scrolled, selected, and changed by the methods of `Prefab`. Module `Tk` is an all-new implementation of the Tk graphics toolkit⁴ with a Limbo interface. A `Math` module encapsulates the procedures for numerical programming.

Environment of the Inferno System

Inferno creates a standard environment for applications. Identical application programs can run under any instance of this environment—even in distributed fashion—and see the same resources. Several versions of the Inferno kernel, Dis/Limbo interpreter, and device driver set can be used depending on the environment within which Inferno itself is implemented.

When running as the native operating system, the kernel includes all the low-level glue (interrupt handlers, graphics, and other device drivers) needed to implement the abstractions presented to applications. For a hosted system—for example, one hosted under UNIX, Windows NT or Windows 95—Inferno runs as a set of ordinary processes. Instead of mapping its device-control functionality to real hardware, it adapts to the resources provided by the operating system under which it runs. Under UNIX, for instance, the graphics library might be implemented using the X Window System* and the networking using the socket interface. Under Windows,* the library uses the native Windows graphics and WinSock calls.

To the extent possible, Inferno is written in standard C language, and most of its components are independent of the many operating systems that can host it.

Security Issues

Inferno provides security of communication, resource control, and system integrity. Each external communication channel may be transmitted in the clear, accompanied by message digests to prevent cor-

ruption, or encrypted to prevent corruption and interception. Once communication is established, channel encryption is transparent to the application. Key exchange is provided through standard public key mechanisms. After key exchange, message digesting and line encryption both use standard symmetric mechanisms.

Inferno is secure against erroneous or malicious applications and encourages safe collaboration between mutually suspicious service providers and clients. The resources available to applications appear exclusively in the name space of the application, and standard protection modes are available. This applies to data, communication resources, and the executable modules that constitute the applications. Security-sensitive resources of the system are accessible only by calling the modules that provide them. In particular, adding new files and servers to the name space—an authenticated operation—is controlled. For example, if the network resources are removed from an application's name space, then it is impossible for it to establish new network connections.

Object modules may be signed by trusted authorities who guarantee their validity and behavior. These signatures may be checked by the system the modules are accessing.

Although Inferno provides a rich variety of authentication and security mechanisms as detailed below, few application programs need to be aware of them or to include coding explicitly to make use of them. Most often, access to resources across a secure communications link is arranged in advance by the larger system in which the application operates. For example, when a client system uses a server system and connection authentication or link encryption is appropriate, the server resources will most naturally be supplied as part of the application's name space. The communications channel that carries the Styx protocol can be set to authenticate or encrypt. Thereafter, all use of the resource is automatically protected.

Security Mechanisms

Authentication and digital signatures are performed using public key cryptography. Public keys are certified by Inferno-based or other certifying authori-

ties who sign the public keys with their own private key. Inferno uses encryption for:

- Mutual authentication of communicating parties,
- Authentication of messages between these parties, and
- Encryption of messages between these parties.

The encryption algorithms Inferno provides include the SHA, MD4, and MD5 secure hashes; Elgamal public key signatures and signature verification;⁵ RC4 encryption; DES encryption; and public key exchange based on the Diffie-Hellman scheme. The public key signatures use keys with moduli up to 4,096 bits (512 bits by default).

No generally accepted national or international authority exists for storing or generating public or private encryption keys. Thus, Inferno includes tools for using or implementing a trusted authority, but it does not itself provide the authority, which is an administrative function. An organization using Inferno (or any other security and key-distribution scheme) must design a system to suit its own needs. In particular, an organization must decide whom to trust as a certifying authority (CA). However, the Inferno design is sufficiently flexible and modular to accommodate the protocols likely to be attractive in practice.

The CA that signs a user's public key determines the size of the key and the public key algorithm used. Tools provided with Inferno use these signatures for authentication. Library interfaces are provided for Limbo programs to sign and verify signatures.

Generally, authentication is performed using public key cryptography. Parties register by having their public keys signed by the CA. The signature covers a secure hash (SHA, MD4, or MD5) of the name of the party, the party's public key, and an expiration time. The signature—which contains the name of the signer—along with the signed information, is termed a *certificate*.

When parties communicate, they use the STS protocol⁶ to establish the identities of the two parties and to create a mutually known secret. The STS protocol uses the Diffie-Hellman algorithm⁷ to cre-

ate this shared secret. The protocol is protected against replay attacks by choosing new random parameters for each conversation. It is secured against man-in-the-middle attacks by requiring the parties to exchange certificates and then digitally signing key parts of the protocol. To masquerade as another party, an attacker must be able to forge that party's signature.

Line Security

A network conversation can be secured against modification alone or against both modification and snooping. To secure against modification, Inferno can append a secure MD5 or SHA hash (called a digest),

hash(secret, message, messageid)

to each message. *Messageid* is a 32-bit number that starts at 0 and is incremented by one for each message sent. Thus, messages cannot be changed, removed, reordered, or inserted into the stream without knowing the secret or breaking the secure hash algorithm.

To secure against snooping, Inferno supports encryption of the complete conversation using either RC4 or the DES with either DES chain block coding (DESCBC) or the DES electronic code book (DESECB).

Inferno uses the same encapsulation format as Netscape's SSL protocol. It is possible to encapsulate a message stream in multiple encapsulations to provide varying degrees of security.

Random Numbers

The strength of cryptographic algorithms depends in part on the strength of the random numbers used for choosing keys, Diffie-Hellman parameters, and initialization vectors. Inferno achieves this strength in two steps. First, a slow (100- to 200-b/s) random bit-stream comes from sampling the low-order bits of a free-running counter whenever a clock ticks. The clock must be unsynchronized or at least poorly synchronized with the counter. This generator is then used to alter the state of a faster pseudo-random number generator. Both the slow and fast generators were tested on a number of architectures using self correlation, random walk, and repeatability tests.

Introduction to Limbo

The application programming language for the Inferno operating system is Limbo. Although Limbo looks syntactically like C, it has a number of features that make it easier to use, safer, and more suited to the heterogeneous and networked Inferno environment. For instance, Limbo has a rich set of basic types, strong typing, garbage collection, concurrency, communications, and modules. It may be interpreted or compiled just in time for efficient and portable execution.

This paper introduces the language by studying an example of a complete and useful Limbo program. The program illustrates general programming, as well as aspects of concurrency, graphics, module loading, and other features of Limbo and Inferno.

The Problem

Our example program is a stripped-down version of the Inferno program `view`, which displays graphical image files on the screen—one per window. This version sacrifices some functionality, generality, and error-checking but still performs the basic job. The files may be configured either in the GIF^{8,9} or JPEG¹⁰ format, and they must be converted before display; or they may already be encoded in the Inferno standard format that needs no conversion. `View` “sniffs” each file to determine what processing it requires, maps the colors if necessary, creates a new window, and copies the converted image to the window. Each window is given a title bar across the top to identify it and to store or hold the buttons that move and delete the window.

The Source

The complete Limbo source for our version of `view` is shown in **Panel 2**. The source is annotated with line numbers for easy reference (Limbo, of course, does not use line numbers). Subsequent sections explain the workings of the program. Although the program is too large to absorb as a first example without some assistance, we recommend skimming the program before moving on to the next section to become familiar with the style of the language. Control syntax derives from C¹¹

while declaration syntax comes from the Pascal family of languages.¹² Limbo borrows features from a number of languages (for example, tuples on lines 45 and 48) and introduces a few new ones (such as explicit module loading on lines 90 and 92).

Modules

Limbo programs are composed of modules that are loaded and linked at run time. Each Limbo source file is the implementation of a single module. In Panel 2, line 1 states that this file implements a module called `View` whose declaration appears in the `module` declaration on lines 15 through 18. The declaration states that the module has one publicly visible element—the function `init`. Other functions and variables defined in the file will be compiled into the module but they will only be accessible internally.

The function `init` has a type signature (argument and return types) that makes it callable from the Inferno shell, a convention not made explicit here. The type of `init` allows `View` to be invoked by typing, for example,

```
view *.jpg
```

at the Inferno command prompt to view all the JPEG files in a directory. This interface is the only requirement that enables the shell to call the module. All programs are constructed from modules, and the shell can directly call some modules because of their type. In fact, the shell invokes `View` by loading it and calling `init`—not, for example, through the services of a system `exec` function as in a traditional operating system.

Of course, not all modules implement shell commands. Modules are also used to construct libraries, services, and other program components. The module `View` uses the services of other modules for I/O, graphics, file format conversion, and string processing. These modules are identified on lines 2 through 14. Each module’s interface is stored in a public include file that holds a definition of a module in much the same manner as lines 15 through 18 of the `View` program. For example, the following is an excerpt from the include file `sys.m:`

```

Sys: module
{
    PATH:   con      "$Sys";
}

FD: adt    # File descriptor
{
    fd:   int;
};

OREAD:  con 0;
OWRITE: con 1;
ORDWR:  con 2;

open:   fn(s: string, mode: int):
        ref FD;
print:  fn(s: string, *): int;
read:   fn(fd: ref FD,
          buf: array of byte,
          n: int): int;
write:  fn(fd: ref FD,
          buf: array of byte,
          n: int): int;
};

```

This example defines a module type called `Sys` that has functions with such familiar names as `open` and `print`, constants like `OREAD` to specify the mode for opening a file, an aggregate type (`adt`) called `FD` returned by `open`, and a constant string called `PATH`.

After including the definition of each module, View declares variables to access each module. Line 3, for instance, declares the variable `sys` to have type `Sys`. It will be used to hold a reference to the implementation of the module. Line 6 imports a number of types from the `draw` (graphics) module to simplify their use. This line states that by default, the implementation of these types is to be that provided by the module referenced by the variable `draw`. Without such an `import` statement, calls to methods of these types would require explicit mention of the module providing the implementation.

Unlike most module languages, which resolve unbound references to modules automatically, Limbo requires explicit loading of module implementations. Although this requires more bookkeeping, it allows a

program to have fine control over the loading (and unloading) of modules, an important property in the small-memory systems in which Inferno is intended to run. Additionally, it allows easy garbage collection of unused modules and permits multiple implementations to serve a single interface, a style of programming we will exploit in `View`.

Declaring a module variable, such as `sys`, is not sufficient to access a module. An implementation must also be loaded and bound to the variable. Lines 21 through 25 load the implementations of the standard modules used by `View`. The `load` operator—for example,

```
sys = load Sys Sys->PATH;
```

takes a type (`Sys`), the filename of the implementation (`Sys->PATH`), and loads it into memory. If the implementation matches the specified type, a reference to the implementation is returned and stored in the variable (`sys`). If not, the constant `nil` will be returned to indicate an error. Conventionally, the `PATH` constant defined by a module names the default implementation. Because `Sys` is a built-in module provided by the system, it has a special form of name. Other modules' `PATH` variables name files containing actual code—for example, `Wmlib->PATH` is `"/dis/lib/wmlib.dis"`. Note, though, that the name of the implementation of the module in a `load` statement can be any string.

Line 26 initializes the `wmlib` module by invoking its `init` function (unrelated to the `init` of `View`). Note the use of the `->` operator to access the member function of the module. The next two lines load modules and also introduce some new notation—they *declare* and *initialize* the module variables storing the reference. Limbo declarations have the general form

```
var : type = value;
```

If the type is missing, it is taken to be the type of the value. So, for example,

```
bufio := load Bufio Bufio->PATH;
```

on line 28 declares a variable of type `Bufio` and initializes it to the result of the `load` expression.

The Main Loop

The `init` function takes two parameters: a graphics context (`ctxt`) for the program, and a list of

Panel 2. An Example of a Limbo Program

```

1  implement View;
2  include "sys.m";
3  sys:Sys;
4  include "draw.m";
5  draw: Draw;
6  Rect, Display, Image: import draw;
7  include "bufio.m";
8  include "imagefile.m";
9  include "tk.m";
10 tk: Tk;
11 include "wmlib.m";
12 wmlib: Wmlib;
13 include "lib.m";
14 str: String;
15 View: module
16 {
17   init: fn(ctxt: ref Draw->Context,
18           argv: list of string);
19   init(ctxt: ref Draw->Context,
20         argv: list of string)
21   {
22     sys = load Sys Sys->PATH;
23     draw = load Draw Draw->PATH;
24     tk = load Tk Tk->PATH;
25     wmlib = load Wmlib Wmlib->PATH;
26     str = load String String->PATH;
27     wmlib->init();
28     imageremap := load Imageremap
29                   Imageremap->PATH;
30     bufio := load Bufio Bufio->PATH;
31
32     argv = tl argv;
33     if(argv != nil
34       && str->prefix("-x ", hd argv))
35     argv = tl argv;
36
37     viewer := 0;
38     while(argv != nil){
39       file := hd argv;
40       argv = tl argv;
41
42       im := ctxt.display.open(file);
43       if(im == nil){
44         idec := filetype(file);
45         if(idec == nil)
46           continue;
47
48         idec->init(bufio);
49         (ri, err) := idec->read(fd);
50         if(ri == nil)
51           continue;
52
53         (im, err) = imageremap->remap(
54           ri, ctxt.display, 1);
55         if(im == nil)
56           continue;
57
58         spawn view(ctxt, im, file,
59                     viewer++);
60
61       }
62
63     }
64
65     corner := string(25+20*(viewer%5));
66     t := tk->toplevel(ctxt.screen,
67                         "-x "+corner+" -y "+corner+
68                         " -bd 2 -relief raised");
69
70     (nil, file) = str->splitr(file,
71                               "/");
72     menubar := wmlib->titlebar(t,
73                               "View: "+file, Wmlib->Hide);
74
75     event := chan of string;
76     tk->namechan(t, event, "event");
77     tk->cmd(t, "frame . im -height " +
78             string im.r.dy() +
79             " -width " +
80             string im.r.dx());
81     tk->cmd(t, "bind . <Configure> "+
82             "{send event resize}");
83     tk->cmd(t, "bind . <Map> "+
84             "{send event resize}");
85     tk->cmd(t, "pack .Wm_t -fill x");
86     tk->cmd(t, "pack .im -side bottom"+
87             " -fill both -expand 1");
88     tk->cmd(t, "update");
89
90     t.image.draw(posn(t), im,
91                 ctxt.display.ones, im.r.min);
92     for(;;) alt{
93       menu := <-menubar =>
94         if(menu == "exit")
95           return;
96         wmlib->titlectl(t, menu);
97       <-event =>
98         t.image.draw(posn(t), im,
99                     ctxt.display.ones, im.r.min);
100    }
101
102    posn(t: ref Tk->Toplevel): Rect
103    {
104      minx := int tk->cmd(t,
105                            ".im cget -actx");
106      miny := int tk->cmd(t,
107                            ".im cget -acty");
108      maxx := minx + int tk->cmd(t,
109                                ".im cget -actwidth");
110      maxy := miny + int tk->cmd(t,
111                                ".im cget -actheight");
112
113      return ((minx, miny), (maxx, maxy));
114    }
115
116    filetype(file: string): RImagefile
117    {
118      if(len file>4
119        && file [len file-4:]=="gif")
120        r := load RImagefile
121                    RImagefile->READGIFPATH;
122      if(len file>4
123        && file [len file-4:]=="jpg")
124        r = load RImagefile
125                    RImagefile->READJPGPATH;
126
127      return r;
128    }

```

command-line argument strings (`argv`). `Argv` is a list of string. Strings are a built-in type in Limbo, and lists are a built-in form of constructor. Lists have several operations defined: `hd` (head) returns the first element in the list, `tl` (tail) returns the remainder after the head, and `len` (length) returns the number of elements in the list.

In Panel 2, line 29 throws away the first element of `argv`, which is the conventional name of the program being invoked by the shell; lines 30 and 31 ignore a geometry argument passed by the window system. The loop from lines 33 to 53 processes each file named in the remaining arguments. When `argv` is a `nil` list, the loop is complete. Line 34 picks off the next filename, and line 35 updates the list.

Line 36 is the first method call we have seen:

```
im := ctxt.display.open(file);
```

The parameter `ctxt` is an adt that contains all the relevant information for the program to access its graphics environment. One of its elements called `display` represents the connection to the frame buffer on which the program may write. The adt `display` (whose type is imported on line 6) has a member function `open` that reads a named image file into the memory associated with the frame buffer, returning a reference to the new image. (In X¹³ terminology, `display` represents a connection to the server and `open` reads a pixmap from a file and instantiates it on that server.)

The `display.open` method succeeds only if the file exists and is configured in the standard Inferno image format. If it fails, it will return `nil`, and lines 38 through 50 will attempt to convert the file into the right form.

Decoding the File

Line 38 in Panel 2 calls `filetype` to determine what format the file has. The simple version shown on lines 87 through 94 just looks at the file suffix to determine the type. A realistic implementation would work harder but even this version illustrates the utility of program-controlled loading of modules.

The decoding interface for an image file format is specified by the module type `RImagefile`. However, unlike the other modules we have examined,

`RImagefile` has a number of implementations. If the file is a GIF file, `filetype` returns the implementation of `RImagefile` that decodes GIFs. If it is a JPEG file, `filetype` returns an implementation that decodes JPEGs. In either case, the `read` method has the same interface. Because reference variables like `r` are implicitly initialized to `nil`, that is what `filetype` will return if it does not recognize the image format. Thus, `filetype` accepts a filename and returns the implementation of a module to decode it.

Two other aspects of `filetype` are worth mentioning. First, the expression `file [len file-4:]` is a *slice* of the string `file`. It creates a string holding the last four characters of the filename. The colon separates the starting and ending indices of the slice. The missing second index defaults to the end of the string. As with lists, `len` returns the number of characters (not bytes; Limbo uses Unicode¹⁴ throughout) in the string.

Second and more importantly, this version of `filetype` loads the decoder module anew every time it is called, which is clearly inefficient. It's easy to do better, though. Just store the module in a global, as in this fragment:

```
readjpg: RImagefile;
filetype(...){...}
{
    if(isjpg()){
        if(readjpg == nil)
            readjpg = load RImagefile
            RImagefile->READJPGPATH;
        return readjpg;
    }
}
```

The program can form its own policies on loading and unloading modules based on time/space or other tradeoffs. The system does not impose its own policies.

Returning to the main loop, after the type of the file has been discovered, line 41 opens the file for I/O using the buffered I/O package. Line 44 calls the `init` function of the decoder module, passing it the instance of the buffered I/O module being used (if we were caching decoder modules, this call to `init` would be done only when the decoder is first

loaded.) Finally, the Limbo-characteristic line 45 reads in the file:

```
(ri, err) := idec->read(fd);
```

The read method of the decoder does the hard job of cracking the image format, which is beyond the scope of this paper. The result is a *tuple* or pair of values. The first element of the pair is the image while the second element is an error string. If all goes well, the `err` will be `nil`. If a problem surfaces, however, `err` may be printed by the application to report what went wrong. The interesting property of this style of error reporting—common to Limbo programs—is that an error can be returned even if the decoding was successful (that is, even if `ri` is non-`nil`). For example, the error may be recoverable. In this case, it is worth returning the result but also worth reporting that an error did occur, leaving the application to decide whether to display the error or ignore it. (View ignores it, for brevity.)

In a similar manner, line 48 remaps the colors from the incoming color map associated with the file to the standard Inferno color map. The result is an image ready to be displayed.

Creating a Process

By line 52 in the main loop (see Panel 2), we have an image ready in the variable `im` and use the Limbo primitive `spawn` to create a new process to display that image on the screen. `Spawn` operates on a function call, creating a new process to execute that function. The process doing the spawning—here the main loop—continues immediately while the new process begins execution in the specified function with the specified parameters. Thus, line 52 begins a new process in the function `view` with these arguments: the graphics context, the image to display, the filename, and a unique identification number used in placing the windows.

The new process shares with the calling process all variables except the stack. Therefore, shared memory can be used to communicate between them. For synchronization, a more sophisticated mechanism is needed, a subject we will cover in the “Communications” section.

Starting Tk

The function `view` uses the Inferno Tk graphics

toolkit (a reimplementation for Limbo of the Tcl/Tk toolkit³) to place the image on the screen in a new window. In Panel 2, line 57 computes the position of the corner of the window using the viewer number to stagger the positions of successive windows. The `string` keyword is a conversion. In this example, the conversion does an automatic translation from an integer expression into a decimal representation of the number. Thus, `corner` is a string variable, a form more useful in the calls to the Tk library.

The Inferno Tk implementation uses Limbo as its controlling language. Rather than building a rich procedural interface, the interface passes strings to a generic Tk command processor, which returns strings as results. This process is similar to the use of Tk within Tcl but with most of the control flow, arithmetic, and so on written in Limbo.

A good introduction to the style is the function `posn` on lines 79 through 86. The calls to `tk->cmd` evaluate the textual command in the context defined by the `Tk->Toplevel` variable `t` (created on line 58 and passed to `posn`). The result is a decimal integer, which the explicit `int` conversion converts to binary. On line 85, all the coordinates of the rectangle are known, and the function returns a nested tuple defining the rectangular position of the `.im` component of the top level. This tuple is automatically promoted to the `Rect` type by the return statement.

Back in function `view`, line 58 calls `tk->toplevel` to create the window on the display. The arguments are `ctxt.screen`, a data structure representing the window stack on the frame buffer, and a string specifying the size and properties of the new window. The `+` operator on strings performs concatenation. The return value from `tk->toplevel` is a reference to a top-level widget—a window—on which the program will assemble its display.

Line 59 uses a function from the higher-level `String` module to strip off the basename of the filename for use in the banner of the window. Note that one component of the tuple is `nil`; the value of this component is discarded. Line 60 calls the window manager function `wmlib->titlebar` to establish a title bar on the window labeled “View:”

and on the file basename with a control button to hide the window. Title bars always include a control button to dismiss the window.

Communications

The return value from `wmlib->titlebar` is a built-in Limbo type called a *channel* (`chan` is the keyword). A channel is a communications mechanism in the manner of communicating sequential processes.¹⁵ Two processes that wish to communicate do so using a shared channel. Data sent on the channel by one process may be received by another process. The communication is *synchronous*—that is, both processes must be ready to communicate before the data changes hands. If one process is not ready, the other blocks until it is. Channels are a feature of the Limbo language. They have a declared type (for example, `chan of int` and `chan of list of string`), and only data of the correct type may be sent. No restriction limits what may be sent. One may even send a channel on a channel. Therefore, channels serve both to communicate and to synchronize.

Channels are used throughout Inferno to provide interfaces to system functions. The threading and communications primitives in Limbo are not designed to implement efficient multicomputer algorithms but rather to provide an elegant way to build active interfaces to devices and other programs.

One example is the `menubut` channel returned by `wmlib->titlebar`, a channel of textual commands sent by the window manager. The expression

```
menu := <-menubut
```

on line 71 in Panel 2 receives the next message on the channel and assigns it to the variable `menu`. The communications operator, `<-`, receives a datum when prefixed to channel and transmits a datum when combined with an assignment operator (for example, `channel<--=2`). This use of `menubut` appears inside an `alt` (alternation) statement, a construct we will discuss later.

Lines 61 and 62 create and register a new channel, `event`, to be used by the Tk module to report user interface events. Lines 63 through 68 use simple Tk operations to make the window in which the image may be drawn. Lines 64 and 65 bind events within this window to messages to be sent on the

channel event. For example, line 64 defines that when the configuration of the window is changed—presumably by actions of the window manager—the string “`resize`” is to be transmitted on `event` for interpretation by the application. This translation of events into messages on explicit channels is fundamental to the Limbo style of programming.

Displaying the Image

The payoff occurs on line 69 in Panel 2, which steps outside the Tk model to draw the image `im` directly on the window:

```
t.image.draw(posn(t), im,
             ctxt.display.ones, im.r.min);
```

`Posn` calculates where on the screen the image is to go. The `draw` method is the fundamental graphics operation in Inferno whose design is outside the scope of this discussion. In this statement, it just copies the pixels from `im` to the window’s own image, `t.image`. The argument `ctxt.display.ones` is a mask that selects every pixel.

Multi-Way Communications

Once the image is on the screen, `view` waits for any changes in the status of the window. Two things might happen: either the buttons on the title bar may be used, in which case a message will appear on `menubut`, or a configuration or mapping operation will apply to the window, in which case a message will appear on `event`.

The Limbo `alt` statement provides control when more than one communication may proceed. Analogous to a `case` statement, the `alt` evaluates a set of expressions and executes the statements associated with the correct expression. Unlike a `case`, though, the expressions in an `alt` each must be a communication, and the `alt` will execute the statements associated with the communication that can first proceed. If none can proceed, the `alt` waits until one can. If more than one statement can proceed, it chooses one randomly.

Thus, the loop on lines 70 through 77 in Panel 2 processes messages received by the two classes of actions. When the window is moved or resized, line 75 will receive a “`resize`” message due to the bindings on lines 64 and 65. The message is discarded but the action of receiving it triggers the repainting of the

newly placed window on line 76. Similarly, messages triggered by buttons on the title bar send a message on menubut. The value of menubut is then examined to see if it is "exit", which should be handled locally, or anything else, which can be passed on to the underlying library.

Cleanup

If the exit button is pushed, line 73 in Panel 2 will return from `view`. Because `view` was the top-level function in this process, the process will exit and free all its resources. All memory, open file descriptors, windows, and other resources the process holds will be collected as garbage when the return executes.

The Limbo garbage collector uses a hybrid scheme that combines reference counting with a real-time sweeping algorithm. Reference counting allows reclamation of memory the instant its last reference disappears; the sweeping algorithm runs as an idle-time process to reclaim unreferenced circular structures. The instant-free property means that system resources like file descriptors and windows can be tied to the collector for recovery as soon as they are no longer used. This property allows Inferno to run in smaller memory arenas than those required for efficient mark-and-sweep algorithms, and it also provides an extra level of programmer convenience.

Summary

Inferno supplies a rich environment for constructing distributed applications that are portable—in fact, identical—even when running on widely divergent underlying hardware. Its unique advantage over other solutions is that it encompasses not only a virtual machine but also a complete virtual operating system, including network facilities.

Acknowledgment

The cryptographic elements of Inferno owe much to the cryptographic library of Jack Lacy, Don Mitchell, and William Schell.¹⁶

*Trademarks

AIX and Power PC are registered trademarks of International Business Machines Corp.

AMD 29K is a registered trademark of Advanced Micro Devices, Inc.

HP/UX is a registered trademark of Hewlett-Packard Inc.

Irix is a trademark of Silicon Graphics, Inc.

Solaris is a registered trademark of Sun Microsystems.

SPARC is a registered trademark of SPARC International.

UNIX is a registered trademark of Novell.

Windows, Windows NT, and Windows 95 are registered trademarks of Microsoft Corp.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

References

1. R. Pike, D. L. Presotto, S. M. Dorward, B. Flandrena, K. Thompson, H. W. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Journal of Computing Systems*, Vol. 8, No. 3, Summer 1995, pp. 221-254.
2. S. M. Dorward, R. Pike, and P. Winterbottom, "Programming in Limbo," *Proceedings of the IEEE Computer Conference (COMPON)*, San Jose, California, 1997.
3. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, New York, 1994.
4. T. Elgamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *Advances in Cryptography: Proceedings of CRYPTO 84*, Springer-Verlag, New York, 1985, pp. 10-18.
5. B. Schneier, *Applied Cryptography*, Chapter 22, John Wiley, New York, 1996, p. 516.
6. D. Stinson, *Cryptography, Theory and Practice*, CRC Press, Cleveland, Ohio, 1996, p. 271.
7. S. M. Dorward, R. Pike, D. M. Ritchie, H. W. Trickey, and P. Winterbottom, "Inferno," *Proceedings of the IEEE Computer Conference (COMPON)*, San Jose, California, Feb. 1997.
8. *GIF Graphics Interchange Format: A Standard Defining a Mechanism for the Storage and Transmission of Bitmap-Based Graphics Information*, CompuServe Inc., Columbus, Ohio, 1987.
9. *GIF Graphics Interchange Format: Version 89a*, CompuServe Inc., Columbus, Ohio, 1990.
10. W. B. Pennebaker and J. L. Mitchell, *JPEG Still-Image Data Compression*, Van Nostrand Reinhold, New York, 1992.
11. *Programming Languages - C*, International Standards Organization (ISO), revision and redesignation of American National Standards Institute (ANSI) X3.159-1989, Amendment 1, 1990.
12. K. Jensen and N. Wirth, *Pascal—User Manual and Report*, Springer-Verlag, New York, 1974.
13. R. W. Scheifler, J. Gettys, and R. Newman,

- X Window System*, Digital Press, Bedford, Massachusetts, 1988.
14. The Unicode Consortium, *The Unicode Standard, Version 2.0*, Addison-Wesley, New York, 1996.
 15. C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the Association for Computing Machinery (ACM)*, Vol. 21, No. 8, 1978, pp. 666-677.
 16. J. B. Lacy, D. P. Mitchell, and W. M. Schell, "CryptoLib: Cryptography in Software," *Proceedings of the UNIX Security Symposium IV*, USENIX, Santa Clara, California, 1993, pp. 1-17.

Further Reading

- *Inferno—A Complete Platform to Develop and Deploy Intelligent Devices in Any Networked Environment*, Bell Labs Computing Sciences Research Center, Murray Hill, New Jersey, 1997.
<http://www.lucent.com/inferno>

(Manuscript approved March 1997)

SEAN M. DORWARD is a member of technical staff in the Computing Structures Research Department at Bell Labs in Murray Hill, New Jersey. He has worked in the areas of protocol verification, network authentication, compiler technology, and audio compression. Currently, his chief responsibility is the Inferno operating system. He also conducts research on programming languages and compilers, as well as on audio and video algorithms. Mr. Dorward received a B.S. degree in computer science from Princeton University in New Jersey.

ROB PIKE is a distinguished member of technical staff in the Computing Sciences Research Department at Bell Labs in Murray Hill, New Jersey. In 1981, he wrote the first bitmap window system for UNIX and has since written ten additional systems. Mr. Pike was a principal designer and implementer of both the Plan 9 and Inferno operating systems. In addition, he designed a gamma-ray telescope, co-designed the Bilt terminal, and co-authored The UNIX Programming Environment. He has never written a program that uses cursor addressing.

DAVID LEO PRESOTTO is a member of technical staff in the Computing Structures Research Department at Bell Labs in Murray Hill, New Jersey. He is responsible for research into and development of a technology known as electronic glue. Mr. Presotto received a Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley.

DENNIS M. RITCHIE is head of the Systems Software Research Department at Bell Labs in Murray Hill, New Jersey. He joined Bell Labs after receiving graduate and undergraduate degrees from Harvard University in Cambridge, Massachusetts. He is a co-developer of the UNIX operating system and is the primary designer of C language in which UNIX and many other systems are written. A Bell Labs Fellow and a member of the U. S. National Academy of Engineering, Mr. Ritchie has received several other honors, including the ACM Turing award, the IEEE Piore, Hamming, and Pioneer awards, and the NEC C&C Foundation award. He continues to work in the areas of operating systems and languages.

HOWARD W. TRICKEY is a member of technical staff in the Computing Architectures Research Department at Bell Labs in Murray Hill, New Jersey. His work involves research and development of the Inferno operating system. Mr. Trickey holds a B.A.Sc. degree in science and an M.A.Sc. degree in electrical engineering from the University of Toronto in Canada, and a Ph.D. in computer science from Stanford University in Palo Alto, California.

PHILIP WINTERBOTTOM is a member of technical staff in the Computing Sciences Research Department at Bell Labs in Murray Hill, New Jersey. He works in the areas of compilers, languages, operating systems, and networking hardware. Before coming to Bell Labs, he attended Kings College in London, England, then continued on to the City University of London where he was a Lloyds Research Fellow building parallel computers. ♦

Portability of C Programs and the UNIX System*

*S. C. JOHNSON
D. M. RITCHIE*

ABSTRACT

Computer programs are portable to the extent that they can be moved to new computing environments with much less effort than it would take to rewrite them. In the limit, a program is perfectly portable if it can be moved at will with no change whatsoever. Recent C language extensions have made it easier to write portable programs. Some tools have also been developed that aid in the detection of nonportable constructions. With these tools many programs have been moved from the PDP-11 on which they were developed to other machines. In particular, the UNIX† operating system and most of its software have been transported to the Interdata 8/32. The source-language representation of most of the code involved is identical in all environments.

I. INTRODUCTION

A program is portable to the extent that it can be easily moved to a new computing environment with much less effort than would be required to write it afresh. It may not be immediately obvious that lack of portability is, or needs to be, a problem. Of course, practically no assembly-language programs are portable. The fact is, however, that most programs, even in high-level languages, depend explicitly or implicitly on assumptions about such machine-dependent features as word and character sizes, character set, file system structure and organization, peripheral device handling, and many others. Moreover, few computer languages are understood by more than a handful of kinds of machines, and those that are (for example, Fortran and Cobol) tend to be rather limited in their scope, and, despite strong standards efforts, still differ considerably from one machine to another.

The economic advantages of portability are very great. In many segments of the computer industry, the dominant cost is development and maintenance of software. Any large organization, certainly including the Bell System, will have a variety of computers and will want to run the same program at many locations. If the program must be rewritten for each machine and maintained for each, software costs must increase. Moreover, the most effective hardware for a given job is not constant as time passes. If a nonportable program remains tied to obsolete hardware to avoid the expense of moving it, the costs are equally real even if less obvious. Finally, there can be considerable benefit in using machines from several manufacturers simply to avoid being utterly dependent on a single supplier.

Most large computer systems spend most of their time executing application programs; circuit design and analysis, network routing, simulation, data base applications, and text processing are particularly important at Bell Laboratories. For years, application programs have been written in high-level languages, but the programs that provide the basic software environment of computers (for example, operating systems, compilers, text editors, etc.) are still usually coded in assembly language. When the costs of hardware were large relative to the costs of software, there was perhaps some justification for this approach; perhaps an equally important reason was the lack of appropriate, adequately supported languages. Today hardware is relatively cheap, software is expensive, and any number of languages are capable of expressing well the algorithms required for basic system software. It is a mystery why the vast majority of computer manufacturers continue to generate so much assembly-language software.

The benefits of writing software in a well-designed language far exceed the costs. Aside from potential portability, these benefits include much smaller development and maintenance costs. It is true that a

* Originally published in The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978, pp 2021-2048.
Copyright © 1978 American Telephone and Telegraph Company.

† At the time UNIX was a trademark of Bell Laboratories; today the mark is owned by the Open Group.

penalty must be paid for using a high-level language, particularly in memory space occupied. The cost in time can usually be controlled: experience shows that the time-critical part of most programs is only a few percent of the total code. Careful design allows this part to be efficient, while the remainder of the program is unimportant.

Thus, we take the position that essentially all programs should be written in a language well above the level of machine instructions. While many of the arguments for this position are independent of portability, portability is itself a very important goal; we will try to show how it can be achieved almost as a by-product of the use of a suitable language.

We have recently moved the UNIX system kernel, together with much of its software, from its original host machine (DEC PDP-11) to a very different machine (Interdata 8/32). Almost all the programs involved are written in the C language [1, 2] and almost all are identical on the two systems. This paper discusses some of the problems encountered, and how they were solved by changing the language itself and by developing tools to detect and resolve nonportable constructions. The major lessons we have learned, and that we hope to teach, are that portable programs are good programs for more reasons than that they are portable, and that making programs portable costs some intellectual effort but need not degrade their performance.

II. HISTORY

The Computing Science Research Center at Bell Laboratories has been interested in the problems and technologies of program portability for over a decade. Altran [3] is a substantial (25,000 lines) computer algebra system, written in Fortran, which was developed with portability as one of its primary goals. Altran has been moved to many incompatible computer systems; the effort involved for each move is quite moderate. Out of the Altran effort grew a tool, the PFORT verifier [4], that checks Fortran programs for adherence to a strict set of programming conventions. Most importantly, it detects (where possible) whether the program conforms to the ANSI standard for Fortran [5], but because many compilers fail to accept even standard-conforming programs, it also remarks upon several constructions that are legal but nevertheless nonportable. Successful passage of a program through PFORT is an important step in assuring that it is portable. More recently, members of the Computer Science Research Center and the Computing Technology Center jointly created the PORT library of mathematical software [6]. Implementation of PORT required research not merely into the language issues, but also into deeper questions of the model of floating point computations on the various target machines. In parallel with this work, the development at Bell Laboratories of Snobol4 [7] marks one of the first attempts at making a significant compiler portable. Snobol4 was successfully moved to a large number of machines, and, while the implementation was sometimes inefficient, the techniques made the language widely available and stimulated additional work leading to more efficient implementations.

III. PORTABILITY OF C PROGRAMS - INITIAL EXPERIENCES

C was developed for the PDP-11 on the UNIX system in 1972. Portability was not an explicit goal in its design, even though limitations in the underlying machine model assumed by the predecessors of C made us well aware that not all machines were the same [2]. Less than a year later, C was also running on the Honeywell 6000 system at Murray Hill. Shortly thereafter, it was made available on the IBM 310 series machines as well. The compiler for the Honeywell was a new product[8]. but the IBM compiler was adapted from the PDP-11 version, as were compilers for several other machines.

As soon as C compilers were available on other machines, a number of programs, some of them quite substantial, were moved from UNIX to the new environments. In general, we were quite pleased with the ease with which programs could be transferred between machines. Still, a number of problem areas were evident. To begin with, the C language was growing and developing as experience suggested new and desirable features. It proved to be quite painful to keep the various C compilers compatible, the Honeywell version was entirely distinct from the PDP-11 version, and the IBM version had been adapted, with many changes, from a by-then obsolete version of the PDP-11 compiler. Most seriously, the operating system interface caused far more trouble for portability than the actual hardware or language differences themselves. Many of the UNIX primitives were impossible to imitate on other operating systems; moreover, some conventions on these other operating systems (for example, strange file formats and record-oriented

I/O) were difficult to deal with while retaining compatibility with UNIX. Conversely, the I/O library commonly used sometimes made UNIX conventions excessively visible--for example, the number 518 often found its way into user programs as the size, in bytes, of a particularly efficient I/O buffer structure.

Additional problems in the compilers arose from the decision to use the local assemblers, loaders, and library editors on the host operating systems. Surprisingly often, they were unable to handle the code most naturally produced by the C compilers. For example, the semantics of possibly initialized external variables in C was quite consciously designed to be implementable in a way identical to Fortran's COMMON blocks to guarantee its portability. It was an unpleasant surprise to discover that the Honeywell assembler would allow at most 61 such blocks (and hence external variables) and that the IBM link-editor preferred to start external variables on even 4096-byte boundaries. Software limitations in the target systems complicated the compilers and, in one case, the problems with external variables just mentioned, forced changes in the C language itself.

IV. THE UNIX PORTABILITY PROJECT

The realization that the operating systems of the target machines were as great an obstacle to portability as their hardware architecture led us to a seemingly radical suggestion: to evade that part of the problem altogether by moving the operating system itself. Transportation of an operating system and its software between non-trivially different machines is rare, but not unprecedented [9-13]. Our own situation was a bit different in that we already had a moderately large, complete, and mature system in wide use at many installations. We could not (or at any rate did not want to) start afresh and redesign the language, the operating system interfaces, and the software. It seemed, though, that despite some problems in each we had a good base to build on.

Our project had three major goals:

- (i) To write a compiler for C that could be changed without grave difficulty to generate code for a variety of machines.
- (ii) To refine and extend the C language to make most C programs portable to a wide variety of machines, mechanically identifying non-portable constructions where possible.
- (iii) To revise or recode a substantial portion of UNIX in portable C, detecting and isolating machine dependencies, and demonstrate its portability by moving it to another machine.

By pursuing each goal, we hoped to attain a corresponding benefit:

- (i) A C compiler adaptable to other machines (independently of UNIX), that puts into practice some recent developments in the theory of code generation.
- (ii) Improved understanding of the proper design of languages that, like C, operate on a level close to that of real machines but that can be made largely machine-independent.
- (iii) A relatively complete and usable implementation of UNIX on at least one other machine, with the hope that subsequent implementations would be fairly straightforward.

We selected the Interdata 8/32 computer to serve as the initial target for the system portability research. It is a 32-bit computer whose design resembles that of the IBM System/360 and /370 series machines, although its addressing structure is rather different; in particular, it is possible to address any byte in virtual memory without use of a base register. For the portability research, of course, its major feature is that it is *not* a PDP-11. In the longer term, we expect to find it especially useful for solving problems, often drawn from numerical analysis, that cannot be handled on the PDP-11 because of its limited address space.

Two portability projects besides those referred to above are particularly interesting. In the period 1976-1977, T. L. Lyon and his associates at Princeton adapted the UNIX kernel to run in a virtual machine partition under VM/370 on an IBM System/370 [14]. Enough software was also moved to demonstrate the feasibility of the effort, though no attempt was made to produce a complete, working system. In the midst of our own work on the Interdata 8/32, we learned that a UNIX portability project, for the similar Interdata 7/32, was under way at the University of Wollongong in Australia [15]. Since everything we know of this effort was discovered in discussion with its major participant, Richard Miller [16], we will remark only that the transportation route chosen was markedly different from ours. In particular, an Interdata C compiler was

adapted from the PDP-11 compiler, and was moved as soon as possible to the Interdata, where it ran under the manufacturer's operating system. Then the UNIX kernel was moved in pieces, first running with dummy device drivers as a task under the Interdata system, and only at the later stages independently. This approach, the success of which must be scored as a real *tour de force*, was made necessary by the 100 kilometers separating the PDP-11 in Sydney from the Interdata in Wollongong.

4.1 Project chronology

Work began in the early months of 1977 on the compiler, assembler, and loader for the Interdata machine. Soon after its delivery at the end of April 1977, we were ready to check out the compiler. At about the same time, the operating system was being scrutinized for nonportable constructions. During May, the Interdata-specific code in the kernel was written, and by June, it was working well enough to begin moving large amounts of software; T. L. Lyon aided us greatly by tackling the bulk of this work. By August, the system was unmistakably UNIX, and it was clear that, as a research project, the portability effort had succeeded, though there were still programs to be moved and bugs to be stamped out. From late summer until October 1977, work proceeded more slowly, owing to a combination of hardware difficulties and other claims on our time; by the spring of 1978 the portability work as such was complete. The remainder of this paper discusses how success was achieved.

V. SOME NON-GOALS

It was and is clear that the portability achievable cannot approach that of Altran, for example, which can be brought up with a fortnight of effort by someone skilled in local conditions but ignorant of Altran itself. In principle, all one needs to implement Altran is a computer with a standard Fortran compiler and a copy of the Altran system tape; to get it running involves only defining of some constants characterizing the machine and writing a few primitive operations in assembly language.

In view of the intrinsic difficulties of our own project, we did not feel constrained to insist that the system be so easily portable. For example, the C compiler is not bootstrapped by means of a simple interpreter for an intermediate language; instead, an acceptably efficient code generator must be written. The compiler is indeed designed carefully so as to make changes easy, but for each new machine it inevitably demands considerable skill even to decide on data representations and run-time conventions, let alone the code sequences to be produced. Likewise, in the operating system, there are many difficult and inevitably machine-dependent issues, including especially the treatment of interrupts and faults, memory management, and device handling. Thus, although we took some care to isolate the machine-dependent portions of the operating system into a set of primitive routines, implementation of these primitives involves deep knowledge of the most recondite aspects of the target machine.

Moreover, we could not attempt to make the portable UNIX system compatible with software, file formats, or inadequate character sets already existing on the machine to which it is moved; to promise to do so would impossibly complicate the project and, in fact, might destroy the usefulness of the result. If UNIX is to be installed on a machine, its way of doing business must be accepted as the right way; afterwards, perhaps, other software can be made to work.

VI. THE PORTABLE C COMPILER

The original C compiler for the PDP-11 was not designed to be easy to adapt for other machines. Although successful compilers for the IBM System/370 and other machines were based on it, much of the modification effort in each case, particularly in the early stages, was concerned with ridding it of assumptions about the PDP-11. Even before the idea of moving UNIX occurred to us, it was clear that C was successful enough to warrant production of compilers for an increasing variety of machines. Therefore, one of the authors (SCJ) undertook to produce a new compiler intended from the start to be easily modified. This new compiler is now in use on the IBM System/370 under both OS and TSS, the Honeywell 6000, the Interdata 8/32, the SEL86 the Data General Nova and Eclipse, the DEC VAX-11/780, and a Bell System processor. Versions are in progress for the Intel 8086 microprocessor and other machines.

The degree of portability achieved by this compiler is satisfying. In the Interdata 8/32 version, there are roughly 8,000 lines of source code. The first pass, which does syntax and lexical analysis and symbol table management, builds expression trees, and generates a bit of machine-dependent code such as

subroutine prologues and epilogues, consists of 4,600 lines of code, of which 600 are machine-dependent. In the second pass, which does the bulk of the code generation, 1,000 out of 3,400 lines are machine-dependent. Thus, out of a total of 8,000 lines, 1,600, or 20 percent, are machine-dependent; the remaining 80 percent are shared with the Honeywell, IBM, and other compilers. As the Interdata compiler becomes more carefully tuned, the machine-dependent figures will rise somewhat; for the IBM, the machine-dependent fraction is 22 percent; for the Honeywell, 25 percent.

These figures both overstate and understate the true difficulty of moving the compiler. They represent the size of those source files that contain machine-dependent code; only a half or a third of the lines in many machine-dependent functions actually differ from machine to machine, because most of the routines involved remain similar in structure. As an example, routines to output branches, align location counters, and produce function prologues and epilogues have a clear machine-dependent component, but nevertheless are logically very similar for all the compilers. On the other hand, as we discuss below, the hardest part of moving the compiler is not reflected in the number of lines changed, but is instead concerned with understanding the code generation issues, the C language, and the target machine well enough to make the modifications effectively.

The new compiler is not only easily adapted to a new machine, it has other virtues as well. Chief among these is that all versions share so much code that maintenance of all versions simultaneously involves much less work than would maintaining each individually. For example, if a bug is discovered in the machine-independent portion, the repair can be made to all versions almost mechanically. Even if the language itself is changed, it is often the case that most of the job of installing the change is machine-independent and usable for all versions. This has allowed the compilers for all machines to remain compatible with a minimum of effort.

The interface between the two passes of the portable C compiler consists of an intermediate file containing mostly representations of expression trees together with character representations of stereotyped code for subroutine prologues and epilogues. Thus a different first pass can be substituted provided it conforms to the interface specifications. This possibility allowed S. I. Feldman to write a first pass that accepts the Fortran 77 language instead of C. At the moment, the Fortran front-end has two versions (which differ by about as much as do the corresponding first passes for C) that feed the code generators for the PDP-11 and the Interdata machines. Thus we apparently have not only the first, but the first two implementations of Fortran 77.

6.1 Design of the portable compiler

Most machine-dependent portions of a C compiler fall into three categories.

- (i) Storage allocation.
- (ii) Rather stereotyped code sequences for subroutine entry points and exits, switches, labels, and the like.
- (iii) Code generation for expressions.

For the most part, storage allocation issues are easily parameterized in terms of the number of bits required for objects of the various types and their alignment requirements. Some issues, like addressability on the IBM 360 and 310 series, cause annoyance, but generally there are few problems in this area. The calling sequence is very important to the efficiency of the result and takes considerable knowledge and imagination to design properly. However, once designed, the calling sequence code and the related issue of stack frame layout are easy to cope with in the compiler.

Generating optimal code for arithmetic expressions, even on idealized machines, can be shown theoretically to be a nearly intractable problem. For the machines we are given in real life, the problem is even harder. Thus, all compilers have to compromise a bit with optimality and engage in heuristic algorithms to some extent, in order to get acceptably efficient code generated in a reasonable amount of time.

The design of the code generator was influenced by a number of goals, which in turn were influenced by recent theoretical work in code generation. It was recognized that there was a premium in being able to get the compiler up and working quickly; it was also felt, however, that this was in many ways less important than being able to evolve and tune the compiler into a high-quality product as time went on.

Particularly with operating system code, a "quick and dirty" implementation is simply unacceptable. It was also recognized that the compiler was likely to be applied to machines not well understood by the compiler writer that might have inadequate or nonexistent debugging facilities. Therefore, one goal of the compiler was to permit it to be largely self-checking. Rather than produce incorrect code, we felt it far preferable for the compiler to detect its own inadequacies and reject the input program.

This goal was largely met. The compiler for the Interdata 8/32 was working within a couple of weeks after the machine arrived; subsequently, several months went by with very little time lost due to compiler bugs. The bug level has remained low, even as the compiler has begun to be more carefully tuned; many of the bugs have resulted from human error (e.g., misreading the machine manual) rather than actual compiler failure.

Several techniques contribute considerably to the general reliability of the compiler. First, a conscious attempt was made to separate information about the machine (e.g., facts such as "there is an add instruction that adds a constant to a register and sets the condition code") from the strategy, often heuristic, that makes use of these facts (e.g., if an addition is to be done, first compute the left-hand operand into a register). Thus, as the compiler evolves, more effort can be put into improving the heuristics and the recognition of important special cases, while the underlying knowledge about the machine operations need not be altered. This approach also improves portability, since the heuristic programs often remain largely unchanged among similar machines, while only the detailed knowledge about the format of the instructions (encoded in a table) changes.

During compilation of expressions, a model of the state of the compilation process, including the tree representing the expression being compiled and the status of the machine registers, is maintained by the compiler. As instructions are emitted, the expression tree is simplified. For example, the expression $a = b+c$ might first be transformed into $a = \text{register } + b$ as a load instruction for a is generated, then into $a = \text{register}$ when an add is produced. The possible transformations constitute the "facts" about the machine: the order in which they are applied correspond to the heuristics. When the input expression has been completely transformed into nothing, the expression is compiled. Thus, a good portion of the initial design of a new version of the compiler is concerned with making the model within the compiler agree with the actual machine by building a table of machine operations and their effects on the model. When this is done correctly, one has a great deal of confidence that the compiler will produce correct code, if it produces any at all.

Another useful technique is to partition the code generation job into pieces that interact only through well-defined paths. One module worries about breaking up large expressions into manageable pieces, and allocating temporary storage locations when needed. Another module worries about register allocation. Finally, a third module takes each "manageable" piece and the register allocation information, and generates the code. The division between these pieces is strict; if the third module discovers that an expression is "unmanageable," or a needed register is busy, it rejects the compilation. The division enforces a discipline on the compiler which, while not really restricting its power, allows for fairly rapid debugging of the compiler output.

The most serious drawback of the entire approach is the difficulty of proving any form of "completeness" property for the compiler--of demonstrating that the compiler will in fact successfully generate code for all legal C programs. Thus, for example, a needed transformation might simply be missing, so that there might be no way to further simplify some expression. Alternatively, some sequence of transformations might result in a loop, so that the same expression keeps reappearing in a chain of transformations. The compiler detects these situations by realizing that too many passes are being made over the expression tree, and the input is rejected. Unfortunately, detection of these possibilities is difficult to do in advance because of the use of heuristics in the compiler algorithms. Currently, the best way of ensuring that the compiler is acceptably complete is by extensive testing.

6.2 Testing the compiler

We ordered the Interdata 8/32 without any software at all, so we first created a very crude environment that allowed stand-alone programs to be run; all interrupts, memory mapping, etc., were turned off. The compiler, assembler, and loader ran on the PDP-11, and the resulting executable files were transferred to the Interdata for testing. Primitive routines permitted individual characters to be written on the console.

In this environment, the basic stack management of the compiler was debugged, in some cases by single-stepping the machine. This was a painful but short period.

After the function call mechanism was working, other short tests established the basic sanity of simple conditionals, assignments, and computations. At this point, the stand-alone environment could be enriched to permit input from the console and more informative output such as numbers and character strings, so ordinary C programs could be run. We solicited such programs, but found few that did not depend on the file system or other operating system features. Some of the most useful programs at this stage were simple games that pitted the computer against a human; they frequently did a large amount of computing, often with quite complicated logic, and yet restricted themselves to simple input and output. A number of compiler bugs were found and fixed by running games. After these tests, the compiler ceased to be an explicit object of testing, and became instead a tool by which we could move and test the operating system.

Some of the most subtle problems with compiler testing come in the maintenance phase of the compiler, when it has been tested, declared to work, and installed. At this stage, there may be some interest in improving the code quality as well as fixing the occasional bug. An important tool here is regression testing; a collection of test programs are saved, together with the previous compiler output.

Before a new compiler is installed, the new compiler is fed these test programs, the new output is compared with the saved output, and differences are noted. If no differences are seen, and a compiler bug has been fixed or improvement made, the testing process is incomplete, and one or more test programs are added. If differences are detected, they are carefully examined. The basic problem is that frequently, in attempting to fix a bug, the most obvious repair can give rise to other bugs, frequently breaking code that used to work. These other bugs can go undetected for some time, and are very painful both to the users and the compiler writer. Thus, regression tests attempt to guard against introducing new bugs while fixing old ones.

The portable compiler is sufficiently self-checked that many potential compiler bugs were detected before the compiler was installed by the simple expedient of turning the compiler loose on a large amount (tens of thousands of lines) of C source code. Many constructions turned up there that were undreamed of by the compiler writer, and often mishandled by the compiler.

It is worth mentioning that this kind of testing is easily carried out by means of the standard commands and features in the UNIX system. In particular, C source programs are easily identified by their names, and the UNIX shell provides features for applying command sequences automatically to each of a list of files in turn. Moreover, powerful utilities exist to compare two similar text files and produce a minimal list of differences. Finally, the compiler produces assembly code that is an ordinary text file readable by all of the usual utilities. Taken together, these features make it very simple to invent test drivers. For example, it takes only a half-dozen lines of input to request a list of differences between the outputs of two versions of the compiler applied to tens (or hundreds) of source files. Perhaps even more important, there is little or no output when the compilers compare exactly. On many systems, the "job control language" required to do this would be so unpleasant as to insure that it would not be done. Even if it were, the resulting hundreds of pages of output could make it very difficult to see the places where the compiler needed attention. The design of the portable C compiler is discussed more thoroughly in Ref. 17.

VII. LANGUAGE AND COMPILER ISSUES

We were favorably impressed, even in the early stages, by the general ease with which C programs could be moved to other machines. Some problems we did encounter were related to weaknesses in the C language itself, so we undertook to make a few extensions.

C had no way of accounting in a machine-independent way for the overlaying of data. Most frequently, this need comes up in large tables that contain some parts having variable structure. As an invented example, a compiler's table of constants appearing in a source program might have a flag indicating the type of each constant followed by the constant's value, which is either integer or floating. The C language as it existed allowed sufficient cheating to express the fact that the possible integer and floating value might be overlaid (both would not exist at once), but it could not be expressed portably because of the inability to express the relative sizes of integers and floating-point data in a machine-independent way. Therefore, the

union declaration was added; it permits such a construction to be expressed in a natural and portable manner. Declaring a union of an integer and a floating point number reserves enough storage to hold either, and forces such alignment properties as may be required to make this storage useful as both an integer and a floating point number. This storage may be explicitly used as either integer or floating point by accessing it with the appropriate descriptor tag.

Another addition was the **typedef** facility, which in effect allows the types of objects to be easily parameterized. **typedef** is used quite heavily in the operating system kernel, where the types of a number of different kinds of objects, for example, disk addresses, file offsets, device numbers, and times of day, are specified only once in a header file and assigned to a specific name; this name is then used throughout. Unlike some languages, C does not permit definition of new operations on these new types; the intent was increased parameterization rather than true extensibility. Although the C language did benefit from these extensions, the portability of the average C program is improved more by restricting the language than by extending it. Because it descended from typeless languages, C has traditionally been rather permissive in allowing dubious mixtures of various types; the most flagrant violations of good practice involved the confusion of pointers and integers. Some programs explicitly used character pointers to simulate unsigned integers; on the PDP-11 the two have the same arithmetic properties. Type unsigned was introduced into the language to eliminate the need for this subterfuge.

More often, type errors occurred unconsciously. For example, a function whose only use of an argument is to pass it to a subfunction might allow the argument to be taken to be an integer by default. If the top-level actual argument is a pointer, the usage is harmless on many machines, but not type-correct and not, in general, portable.

Violations of strict typing rules existed in many, perhaps most, of the programs making up the entire stock of UNIX system software. Yet these programs, representing many tens of thousands of lines of source code, all worked correctly on the PDP-11 and in fact would work on many other machines, because the assumptions they made were generally, though not universally, satisfied. It was not feasible simply to declare all the suspect constructions illegal. Instead, a separate program was written to detect as many dubious coding practices as possible. This program, called **lint**, picks bits of fluff from programs in much the same way as the PFORT verifier mentioned above. C programs acceptable to lint are guaranteed to be free from most common type errors; lint also checks syntax and detects some logical errors, such as uninitialized variables, unused variables, and unreachable code.

There are definite advantages in separating program-checking from compilation. First, lint was easy to produce, because it is based on the portable compiler and thus shares the machine independent code of the first pass with the other versions of the compiler. More important, the compilers, large programs anyway, are not burdened with a great deal of checking code which does not necessarily apply to the machine for which they are running. A good example of extra capability feasible in lint but probably not in the compilers themselves is checking for inter-program consistency. The C compilers all permit separate compilation of programs in several files, followed by linking together of the results. **lint** (uniquely) checks consistency of declarations of external variables, functions, and function arguments among a set of files and libraries.

Finally, **lint** itself is a portable program, identical on all machines. Although care was taken to make it easy to propagate changes in the machine-independent parts of the compilers with a minimum of fuss, it has proved useful for the sometimes complicated logic of **lint** to be totally decoupled from the compilers. **lint** cannot possibly affect their ability to produce code; if a bug in **lint** turns up, its output can be ignored and work can continue simply by ignoring the spurious complaints. This kind of separation of function is characteristic of UNIX programs in general. The compiler's one important job is to generate code; it is left to other programs to print listings, generate cross-reference tables, and enforce style rules.

VIII. THE PORTABILITY OF THE UNIX KERNEL

The UNIX operating system kernel, or briefly the operating system, is the permanently resident program that provides the basic software environment for all other programs running on the machine. It implements the "system calls" by which user's programs interact with the file system and request other services, and arranges for several programs to share the machine without interference. The structure of the UNIX operating system kernel is discussed elsewhere in this issue [18, 19]. To many people, an operating system

may seem the very model of a nonportable program, but in fact a major portion of UNIX and other well-written operating systems consists of machine-independent algorithms: how to create, read, write, and delete files, how to decide who to run and who to swap, and so forth. If the operating system is viewed as a large C program, then it is reasonable to hope to apply the same techniques and tools to it that we apply to move more modest programs. The UNIX kernel can be roughly divided into three sections according to their degree of portability.

8.1 Assembly-language primitives

At the lowest level, and least portable, is a set of basic hardware interface routines. These are written in assembly language, and consist of about 800 lines of code on the Interdata 8/32. Some of them are callable directly from the rest of the system, and provide services such as enabling and disabling interrupts, invoking the basic I/O operations, changing the memory map so as to switch execution from one process to another, and transmitting information between a user process's address space and that of the system. Most of them are machine-independent in specification, although not implementation. Other assembly-language routines are not called explicitly but instead intercept interrupts, traps, and system calls and turn them into C-style calls on the routines in the rest of the operating system.

Each time UNIX is moved to a new machine, the assembly language portion of the system must be rewritten. Not only is the assembly code itself machine-specific, but the particular features provided for memory mapping, protection, and interrupt handling and masking differ greatly from machine to machine. In moving from the PDP-11 to the Interdata 8/32, a huge preponderance of the bugs occurred in this section. One reason for this is certainly the usual sorts of difficulties found in assembly-language programming: we wrote loops that did not loop or looped forever, garbled critical constants, and wrote plausible-looking but utterly incorrect address constructions. Lack of familiarity with the machine led us to incorrect assumptions about how the hardware worked, and to inefficient use of available status information when things went wrong.

Finally, the most basic routines for multi-programming, those that pass control from one process to another, turned out (after causing months of nagging problems) to be incorrectly specified and actually unimplementable correctly on the Interdata, because they depended improperly on details of the register-saving mechanism of the calling sequence generated by the compiler. These primitives had to be redesigned; they are of special interest not only because of the problems they caused, but because they represent the only part of the system that had to be significantly changed, as distinct from expressed properly, to achieve portability.

8.2 Device drivers

The second section of the kernel consists of device drivers, the programs that provide the interrupt handling, I/O command processing, and error recovery for the various peripheral devices connected to the machine. On the Interdata 8/32 the total size of drivers for the disk, magnetic tape, console typewriter, and remote typewriters is about 1100 lines of code, all in C. These programs are, of course, machine-dependent, since the devices are.

The drivers caused far fewer problems than did the assembly language programs. Of course, they already had working models on the PDP-11, and we had faced the need to write new drivers several times in the past (there are half a dozen disk drivers for various kinds of hardware attached to the PDP-11). In adapting to the Interdata, the interface to the rest of the system survived unchanged, and the drivers themselves shared their general structure, and even much code, with their PDP-11 counterparts. The problems that occurred seem more related to the general difficulty of dealing with the particular devices than in expressing what had to be done.

8.3 The remainder of the system

The third and remaining section of the kernel is the largest. It is all written in C, and for the Interdata 8/32 contains about 7,000 lines of code. This is the operating system proper, and clearly represents the bulk of the code. We hoped that it would be largely portable, and as it turned out our hopes were justified. A certain amount of work had to be done to achieve portability. Most of it was concerned with making sure that everything was declared properly, so as to satisfy lint, and with replacing constants by parameters. For

example, macros were written to perform various unit conversions previously written out explicitly: byte counts to memory segmentation units and to disk blocks, etc. The important data types used within the system were identified and specified using `typedef`: disk offsets, absolute times, internal device names, and the like. This effort was carried out by K. Thompson.

Of the 7,000 lines in this portion of the operating system, only about 350 are different in the Interdata and PDP-11 versions; that is, they are 95 percent identical. Most of the differences are traceable to one of three areas.

- (i) On the PDP-11, the subroutine call stack grows towards smaller addresses, while on the Interdata it grows upwards. This leads to different code when increasing the size of a user stack, and especially when creating the argument list for an inter-program transfer (`exec` system call) because the arguments are placed on the stack.
- (ii) The details of the memory management hardware on the two machines are different, although they share the same general scheme.
- (iii) The routine that handles processor traps (memory faults, etc.) and system calls is rather different in detail on the two machines because the set of faults is not identical, and because the method of argument transmission in system calls differs as well.

We are extremely gratified by the ease with which this portion of the system was transferred. Only a few problems showed up in the code that was not changed; most were in the new code written specifically for the Interdata. In other words, what we thought would be portable did in fact move without trouble.

Not everything went perfectly smoothly, of course. Our first set of major problems involved the mechanics of transferring test systems and other programs from the PDP-11 to the Interdata 8/32 and debugging the result. Better communications between the machines would have helped considerably. For a period, installing a new Interdata system meant creating an 800 BPI tape on the sixth-floor PDP-11, carrying the tape to another PDP-11 on the first floor to generate a 1600 BPI version, and finally lugging the result to the fifth-floor Interdata. For debugging, we would have been much aided by a hardware interface between the PDP-11 and the front panel of the Interdata to allow remote rebooting. This class of problems is basically our own fault, in that we traded the momentary ease of not having to write communications software or build hardware for the continuing annoyance of carrying tapes and hands-on debugging.

Another class of problems seems impossible to avoid, since it stems from the basic differences in the representation of information on the two machines. In the machines at issue, only one difference is important: the PDP-11 addresses the two bytes in a 16-bit word. In the Interdata the first byte in a 16-bit half-word is the most significant 8 bits. Since all the interfaces between the two machines are byte-serial, the effect is best described by saying that when a true character stream is transmitted between them, all is well; but if integers are sent, the bytes in each half-word must be swapped. Notice that this problem does not involve portability in the sense in which it has been used throughout this paper; very few C programs are sensitive to the order in which bytes are stored on the machine on which they are running. Instead it complicates "portability" in its root meaning wherein files are carried from one machine to the other. Thus, for example, during the initial creation of the Interdata system we were obliged to create, on the PDP-11, an image of a file system disk volume that would be copied to tape and thence to the Interdata disk, where it would serve as an actual file system for the latter machine. It required a certain amount of cleverness to declare the data structures appropriately and to decide which bytes to swap.

The ordering of bytes in a word on the PDP-11 is somewhat unusual, but the problem it poses is quite representative of the difficulties of transferring encoded information from machine to machine. Another example is the difference in representation of floating-point numbers between the PDP-11 and the Interdata. The assembler for the Interdata, when it runs on the PDP-11, must invoke a routine to convert the "natural" PDP-11 notation to the foreign notation, but of course this conversion must not be done when the assembler is run on the Interdata itself. This makes the assembler necessarily non-portable, in the sense that it must execute different code sequences on the two machines. However, it can have a single source representation by taking advantage of conditional compilation depending on where it will run.

This kind of problem can get much worse: how are we to move UNIX to a target machine with a 36-bit word length, whose machine word cannot even be represented by long integers on the PDP-11? Nevertheless, it is worth emphasizing that the problem is really vicious only during the initial bootstrapping

phase; all the software should run properly if only it can be moved once!

IX. TRANSPORTATION OF THE SOFTWARE

Most UNIX code is in neither the operating system itself nor the compiler, but in the many user-level utilities implementing various commands and in subroutine libraries. The sheer bulk of the programs involved (about 50,000 lines of source) meant that the amount of work in transportation might be considerable, but our early experience, together with the small average size of each individual program, convinced us that it would be manageable. This proved to be the case.

Even before the advent of the Interdata machine, it was realized, as mentioned above, that many programs depended to an undesirable degree not only on UNIX I/O conventions but on details of particularly favorable buffering strategies for the PDP-11. A package of routines, called the "portable I/O library," was written by M. E. Lesk [20] and implemented on the Honeywell and IBM machines as well as the PDP-11 in a generally successful effort to overcome the deficiencies of earlier packages. This library too proved to have some difficulties, not in portability, but in time efficiency and space required. Therefore a new package of routines, dubbed the "standard I/O library," was prepared. Similar in spirit to the portable library, it is somewhat smaller and much faster. Thus, part of the effort in moving programs to the Interdata machine was devoted to making programs use the new standard I/O library. In the simplest cases, the effort involved was nil, since the fundamental character I/O functions have the same names in all libraries.

Next, each program had to be examined for visible lack of portability. Of course, lint was a valuable tool here. Programs were also scrutinized by eye to detect dubious constructions. Often these involved constants. For example, on the 16-bit PDP-11 the expression

x & 0177770

masks off all but the last three bits of x, since 0177770 is an octal constant. This is almost certainly better expressed

x & ~07

(where \sim is the ones-complement operator) because the latter expression actually does yield the last three bits of x independently of the word length of the machine. Better yet, the constant should be a parameter with a meaningful name.

UNIX software has a number of conventional data structures, ranging from objects returned or accepted by the operating system kernel (such as status information for a named file) to the structure of the header of an executable file. Programs often had a private copy of the declaration for each such structure they used, and often the declaration was nonportable. For example, an encoded file mode might be declared **int** on the 16-bit PDP-11, but on the 32-bit Interdata machine, it should be specified as **short**, which is unambiguously 16 bits. Therefore, another major task in making the software portable was to collect declarations of all structures common to several routines, to put the declarations in a standard place, and to use the **include** facility of the C preprocessor to insert them in the source program. The compiler for the PDP-11 and the cross-compiler for the Interdata 8/32 were adjusted to search a different standard directory to find the canned declarations appropriate to each.

Finally, an effort was made to seek out frequently occurring patches of code and replace them by standard subroutines, or create new subroutines where appropriate. It turned out, for example, that several programs had built-in subroutines to find the printable user name corresponding to a numerical user ID. Although in each case the subroutine as written was acceptably portable to other machines, the function it performed was not portable in time across changes in the format of the file describing the name-number correspondence; encapsulating the translation function insulated the program against possible changes in a data base.

X. THE MACHINE MODEL FOR C

One of the hardest parts of designing a language in which to write portable programs is deciding which properties are guaranteed to remain invariant. Likewise, in trying to develop a portable operating system, it is very hard to decide just what properties of the underlying machine can be depended on. The design questions in each case are many in number; moreover, the answer to each individual question may

involve tradeoffs that are difficult to evaluate in advance. Here we try to show the nature of these tradeoffs and what sort of compromises are required.

Designing a language in which every program is portable is actually quite simple: specify precisely the meaning of every legal program, as well as what programs are legal. Then the portability problem does not exist: by definition, if a correct program fails on some machine, the language has not been implemented properly. Unfortunately, a language like C that is intended to be used for system programming is not very adaptable to such a Procrustean approach, mainly because reasonable efficiency is required. Any well-defined language can be implemented precisely on any general-purpose computer, but the implementation may not be usable in practice if it implies use of an interpreter rather than machine instructions. Thus, with both language and operating system design, one must strike a balance between convenient and powerful features and the ease of implementing them efficiently on a variety of machines. At any point, some machine may be found on which some feature is very expensive to provide, and a decision must be made whether to modify the feature, and thus compromise the portability of programs that use it, or to insist that the meaning is immutable and must be preserved. In the latter case portability is also compromised since the cost of using the feature may be so high that no one can afford the programs that use it, or the people attempting to implement the feature on the new machine give up in despair.

Thus a language definition implies a model of the machine on which programs in the language will run. If a real machine conforms well to the model, then an implementation on that machine is likely to be efficient and easily written; if not, the implementation will be painful to provide and costly to use. Here we shall consider the major features of the abstract C machine that have turned out to be most relevant so far.

10.1 Integers

Probably the most frequent operations are on integers consisting of various numbers of bits. Variables declared **short** are at least 16 bits in length; those declared **long** are at least 32 bits. Most are declared **int**, and must be at least as precise as short integers, but may be **long** if accessing them as such is more efficient. It is interesting that the word length, which is one of the machine differences that springs first to mind, has caused rather little trouble. A small amount of code (mostly concerned with output conversion) assumes a twos complement representation.

10.2 Unsigned Integers

Unsigned integers corresponding to **short** and **int** must be provided. The most relevant properties of unsigned integers appear when they are compared or serve as numerators in division and remaindering. Unsigned arithmetic may be somewhat expensive to implement on some machines, particularly if the number representation is sign-magnitude or ones complement. No use is made of unsigned **long** integers.

10.3 Characters

A representation of characters (bytes) must be provided with at least 8 bits per byte. It is irrelevant whether bytes are signed, as in the PDP-11, or not, as in all other known machines. It is moderately important that an integer of any kind be divisible evenly into bytes. Most programs make no explicit use of this fact, but the I/O system uses it heavily. (This tends to rule out one plausible representation of characters on the DEC PDP-10, which is able to access five 7-bit characters in a 36-bit word with one bit left over. Fortunately, that machine can access four 9-bit characters equally well.) Almost all programs are independent of the order in which the bytes making up an integer are stored, but see the discussion above on this issue.

A fair number of programs assume that the character set is ASCII. Usually the dependence is relatively minor, as when a character is tested for being a lower case letter by asking if it is between a and z (which is not a correct test in EBCDIC). Here the test could be easily replaced by a call to a standard macro. Other programs that use characters to index a table would be much more difficult to render insensitive to the character set. ASCII is, after all, a U. S. national standard; we are inclined to make it a UNIX standard as well, while not ruling out C compilers for other systems based on other character sets (in fact the current IBM System/370 compiler uses EBCDIC).

10.4 Pointers

Pointers to objects of the various basic types are used very heavily. Frequent operations on pointers include assignment, comparison, addition and subtraction of an integer, and dereferencing to yield the object to which the pointer points. It was frequently assumed in earlier UNIX code that pointers and integers had a similar representation (for example, that they occupied the same space). Now this assumption is no longer made in the programs that have been moved. Nevertheless, the representation of pointers remains very important, particularly in regard to character pointers, which are used freely. A word-addressed machine that lacks any natural representation of a character pointer may suffer serious inefficiency for some programs.

10.5 Functions and the calling sequence

UNIX programs tend to be built out of many small, frequently called functions. It is not unusual to find a program that spends 20 percent of its time in the function prologue and epilogue sequence, nor one in which 20 percent of the code is concerned with preparing function argument lists. On the PDP-11/70 the calling sequence is relatively efficient (it costs about 20 microseconds to call and return from a function) so it is clear that a less efficient calling sequence will be quite expensive. Any function in C may be recursive (without special declaration) and most possess several "automatic" variables local to each invocation. These characteristics suggest strongly that a stack must be used to store the automatic variables, caller's return point, and saved registers local to each function; in turn, the attractiveness of an implementation will depend heavily on the ease with which a stack can be maintained. Machines with too few index or base registers may not be able to support the language well.

Efficiency is important in designing a calling sequence; moreover, decisions made here tend to have wide implications. For example, some machines have a preferred direction of growth for the stack. On the PDP-11, the stack is practically forced to grow towards smaller addresses; on the Interdata the stack prefers (somewhat more weakly) to grow upwards. Differences in the direction of stack growth leads to differences in the operating system, as has already been mentioned.

XI. THE MACHINE MODEL OF UNIX

The definition of C suggests that some machines are more suitable for C implementations than others; likewise, the design of the UNIX kernel fits in well with some machine architectures and poorly with others. Once again, the requirements are not absolute, but a serious enough mismatch may make an implementation unattractive. Because the system is written in C, of course, a (perhaps necessarily) slow or bulky implementation of the language will lead to a slow or bulky operating system, so the remarks in the previous section apply. But other aspects of machine design are especially relevant to the operating system.

11.1 Mapping and the user program

As discussed in other papers, [18, 21] the system provides user programs with an address space consisting of up to three logical segments containing the program text, an extensible data region, and a stack. Since the stack and the data are both allowed to grow at one edge, it is desirable (especially where the virtual address space is limited) that one grow in the negative direction, towards the other, so as to optimize the use of the address space. A few programs still assume that the data space grows in the positive direction (so that an array at its end can grow contiguously), although we have tried to minimize this usage. If the virtual address space is large, there is little loss in allowing both the data and stack areas to grow upwards.

The PDP-11 and the Interdata provide examples of what can be done. On the former machine, the data area begins at the end of the program text and grows upwards, while the stack begins at the end of the virtual address space and grows downwards; this is, happily, the natural direction of growth for the stack. On the Interdata the data space begins after the program and grows upwards; the stack begins at a fixed location and also grows upwards. The layout provides for a stack of at most 128K bytes and a data area of 852K bytes less the program size, as compared to the total data and stack space of 64K bytes possible on the PDP-11.

It is hard to characterize precisely what is required of a memory mapping scheme except by discussing, as we do here, the uses to which it is put. In general, paging or segmentation schemes seem to offer

sufficient generality to make implementation simple; a single base and limit register (or even dual registers, if it is desired to write-protect the program text) are marginal, because of the difficulty of providing independently growable data and stack areas.

11.2 Mapping and the kernel

When a process is running in the UNIX kernel, a fixed region of the kernel's address space contains data specific to that process, including its kernel stack. Switching processes essentially involves changing the address map so that the same fixed range of virtual addresses refers to the data area and stack of the new process. This implies, of course, that the kernel runs in mapped mode, so that mapping should not be tied to operating in user mode. It also means that if the machine has but a single set of mapping specification registers, these registers will have to be reloaded on each system call and certain interrupts, for example from the clock. This causes no logical problems but may affect efficiency.

11.3 Other considerations

Many other aspects of machine design are relevant to implementation of the operating system but are probably less important, because on most machines they are likely to cause no difficulty. Still, it is worthwhile to attempt a list.

- (i) The machine must have a clock capable of generating interrupts at a rate not far from 50 or 60 Hz. The interrupts are used to schedule internal events such as delays for mechanical motion on typewriters. As written, the system uses clock interrupts to maintain absolute time, so the interrupt rate should be accurate in the long run. However, changes to consult a separate time-of-day clock would be minimal.
- (ii) All disk devices should be able to handle the same, relatively small, block sizes. The current system usually reads and writes 512-byte blocks. This number is easy to change, but if it is made much larger, the efficacy of the system's cache scheme will degrade seriously unless a large amount of memory is devoted to buffers.

XII. WHAT HAS BEEN ACCOMPLISHED?

In about six months, we have been able to move the UNIX operating system and much of its software from its original host, the PDP11, to another, rather different machine, the Interdata 8/32. The standard of portability achieved is fairly high for such an ambitious project: the operating system (outside of device drivers and assembly language primitives) is about 95 percent unchanged between the two systems; inherently machine-dependent software such as the compiler, assembler, loader, and debugger are 75 to 80 percent unchanged; other user-level software (amounting to about 20,000 lines so far) is identical, with few exceptions, on the two machines.

It is true that moving a program from one machine to another does not guarantee that it can be moved to a third. There are many issues in portability about which we worried in a theoretical way without having to face them in fact. It would be interesting, for example, to tackle a machine in which pointers were a different size from integers, or in which character pointers were fundamentally different in structure from integer pointers, or with a different character set. There are probably even issues in portability that we failed to consider at all. Nevertheless, moving UNIX to a third new machine, or a fourth, will be easier than it was to the second. The operating system and the software have been carefully parameterized, and this will not have to be done again. We have also learned a great deal about the critical issues (the "hard parts").

There are deeper limitations to the generality of what we have done. Consider the use of memory mapping: if the hardware cannot support the model assumed by the code as it is written, the code must be changed. This may not be difficult, but it does represent a loss of portability. Correspondingly, the system as written does not take advantage of extra capability beyond its model, so it does not support (for example) demand paging. Again, this would require new code. More generally, algorithms do not always scale well; the optimal methods of sorting files of ten, a thousand, and a million elements do not much resemble one another. Likewise, some of the design of the system as it exists may have to be reworked to take full advantage of machines much more powerful (along many possible dimensions) than those for which it was designed. This seems to be an inherent limit to portability; it can only be handled by making the system

easy to change, rather than easily portable unchanged. Although we believe UNIX possesses both virtues, only the latter is the subject of this paper.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
2. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," *B.S.T.J.*, this issue, pp. 1991-2019.
3. W. S. Brown, *ALTRAN User's Manual*, 4th ed., Murray Hill, N.J.: Bell Laboratories, 1977.
4. B. G. Ryder, "The PFORTRAN Verifier," *Software - Practice and Experience*, (October-December 1974), pp. 359-377.
5. American National Standard FORTRAN, New York, N.Y.: American National Standards Institute, 1966. (ANSI X3.9)
6. P. A. Fox, A. D. Hall, and N. L. Schryer, "The PORT Mathematical Subroutine Library," *ACM Trans. Math. Soft.* (1978), to appear.
7. R. E. Griswold, J. Poage, and I. Polonsky, *The SNOBOL4 Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1971.
8. A. Snyder, *A Portable Compiler for the Language C*, Cambridge, Mass.: Master's Thesis, M.I.T., 1974.
9. D. Morris, G. R. Frank, and C. J. Theaker, "Machine-Independent Operating Systems," in *Information Processing 77*, North-Holland (1977), pp. 819-825.
10. J. E. Stoy and C. Strachey, "OS6--An experimental operating system for a small computer. Part 1: General principles and structure," *Comp. J.*, 15 (May 1972), pp. 117-124.
11. J. E. Stoy and C. Strachey, "OS6--An experimental operating system for a small computer. Part 2: Input/output and filing system." *Comp. J.*, 15 (August 1972), pp. 195-203.
12. D. Thalmann and B. Levrat, "SPIP, a Way of Writing Portable Operating Systems," *Proc. ACM Computing Symposium* (1977), pp. 452-459.
13. L. S. Melen, *A Portable Real-Time Executive*, Thoth. Waterloo, Ontario, Canada: Master's Thesis, Dept. of Computer Science, University of Waterloo, October 1976.
14. T. L. Lyon, private communication
15. R. Miller, "UNIX - A Portable Operating System?" *Australian Universities Computing Science Seminar* (February, 1978).
16. R. Miller, private communication.
17. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages* (January 1978).
18. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *B.S.T.J.*, this issue, pp. 1905-1929.
19. D. M. Ritchie, "UNIX Time-Sharing System: A Retrospective," *B.S.T.J.*, this issue, pp. 1947-1969. Also in *Proc. Hawaii International Conference on Systems Science*, Honolulu, Hawaii, Jan. 1977.
20. D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, "The C Programming Language," *Comp. Sci. Tech. Rep. No. 31*, Bell Laboratories (October 1975).
21. K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," *B.S.T.J.*, this issue, pp. 1931-1946.

The UNIX Time-sharing System—A Retrospective*

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

UNIX is a general-purpose, interactive time-sharing operating system for the DEC PDP-11 and Interdata 8/32 computers. Since it became operational in 1971, it has become quite widely used. This paper discusses the strong and weak points of UNIX and some areas where we have expended no effort. The following areas are touched on:

- The structure of files: a uniform, randomly-addressable sequence of bytes. The irrelevance of the notion of “record.” The efficiency of the addressing of files.
- The structure of file system devices; directories and files.
- I/O devices integrated into the file system.
- The user interface: fundamentals of the shell, I/O redirection, and pipes.
- The environment of processes: system calls, signals, and the address space.
- Reliability: crashes, losses of files.
- Security: protection of data from corruption and inspection; protection of the system from stoppages.
- Use of a high-level language—the benefits and the costs.
- What UNIX does not do: “real-time,” interprocess communication, asynchronous I/O.
- Recommendations to system designers.

UNIX is a general-purpose, interactive time-sharing operating system primarily for the DEC PDP-11 series of computers, and recently for the Interdata 8/32. Since its development in 1971, it has become quite widely used, although publicity efforts on its behalf have been minimal, and the license under which it is made available outside the Bell System explicitly excludes maintenance. Currently there are more than 300 Bell System installations, and an even larger number in universities, secondary schools, and commercial and government institutions. It is useful on a rather broad range of configurations, ranging from a large PDP-11/70 supporting 48 users to a single-user LSI-11 system.

Some General Observations

In most ways UNIX is a very conservative system. Only a handful of its ideas are genuinely new. In fact, a good case can be made that it is in essence a modern implementation of MIT’s CTSS system [1]. This claim is intended as a compliment to both UNIX and CTSS. Today, more than fifteen years after CTSS was born, few of the interactive systems we know of are superior to it in ease of use; many are inferior in basic design.

UNIX was never a “project;” it was not designed to meet any specific need except that felt by its major author, Ken Thompson, and soon after its origin by the author of this paper, for a pleasant environment in which to write and use programs. Although it is rather difficult, after the fact, to try to account for

* A version of this paper was presented at the Tenth Hawaii International Conference on the System Sciences, Honolulu, January, 1977.

its success, the following reasons seem most important.

- It is simple enough to be comprehended, yet powerful enough to do most of the things its users want.
- The user interface is clean and relatively surprise-free. It is also terse to the point of being cryptic.
- It runs on a machine that has become very popular in its own right.
- Besides the operating system and its basic utilities, a good deal of interesting software is available, including a sophisticated text-processing system that handles complicated mathematical material [2], and produces output on a typesetter or a typewriter terminal, and a LALR parser-generator [3].

This paper discusses the strong and weak points of the system and lists some areas where no effort has been expended. Only enough design details are given to motivate the discussion; more can be found elsewhere in this issue [4, 5].

One problem in discussing the capabilities and deficiencies of UNIX is that there no unique version of the system. It has evolved continuously both in time, as new functions are added and old problems repaired, and in space, as various organizations add features intended to meet their own needs. Four important versions of the system are in current use:

- The standard system maintained by the UNIX Support Group at Bell Laboratories for Bell System projects.
- The "Programmer's Workbench" version of UNIX [6, 7], also in wide use within Bell Laboratories, especially in areas in which text-processing and job-entry to other machines are important. Recently, the PWB system has become available to outside organizations as well.
- The "Sixth Edition" system (so called from the manual that describes it), which is the most widely used under Western Electric licenses by organizations outside the Bell System.
- The version currently used in the Computer Science Research Center, where UNIX was developed, and at a few other locations at Bell Laboratories.

The proliferation of versions makes some parts of this paper hard to write, especially where details (e.g., how large can a file be?) are mentioned. Although compilation of a list of differences between versions of UNIX is a useful exercise, this is not the place for such a list, so the paper will concentrate on the properties of the system as it exists for the author, in the current Research version of the system.

The existence of several variants of UNIX is, of course, a problem not only when attempting to describe the system in a paper such as this, but also to the users and administrators. The importance of this problem is not lost upon the proprietors of the various versions; indeed, vigorous effort is underway to combine the best features of the variants into a single system.

The Structure of Files

The UNIX file system is simple in structure; nevertheless, it is more powerful and general than those often found even in considerably larger operating systems. Every file is regarded as a featureless, randomly-addressable sequence of bytes. The system conceals physical properties of the device on which the file is stored, such as the size of a disk track. The size of a file is the number of bytes it contains; the last byte is determined by the high-water mark of writes to the file. It is not necessary, nor even possible, to preallocate space for a file. The system calls to read and write each come in only one form, which specifies the local name of an open file, a buffer to or from which to perform I/O, and a byte count. I/O is normally sequential, so the first byte referred to by a read or write operation immediately follows the final byte transferred by the preceding operation. "Random access" is accomplished using a "seek" system call, which moves the system's internal read (or write) pointer for the instance of the open file to another byte that the next read or write will implicitly address. All I/O appears completely synchronous; read-ahead and write-behind are performed invisibly by the system.

This particularly simple way of viewing files was suggested by the Multics I/O system [8].

The addressing mechanism for files must be carefully designed if it is to be efficient. Files can be large (about 10^{10} bytes), are grown without pre-allocation, and are randomly accessible. The overhead per file must be small, because there can be many files (the machine on which this paper was written has about 27,000 on the disk storing most user's files); many of them are small (80 per cent have ten or fewer 512-

byte blocks, and 37 per cent are only one block long). The details of the file-addressing mechanism are given elsewhere [5].

No careful study has been made of the efficiency of disk I/O, but a simple experiment suggests that the efficiency is comparable to two other systems, DEC's IAS for the PDP-11, and Honeywell's GCOS TSS system running on the H6070. The experiment consisted of timing a program that copied a file that, on the PDP-11, contained 480 blocks (245,760 bytes). The file on the Honeywell had the same number of bytes (each of nine bits rather than eight) but there were 1280 bytes per block. With otherwise idle machines, the real times to accomplish the file copies were

system	sec.	msec./block
UNIX	21	21.8
IAS	19	19.8
H6070	9	23.4

The effective transfer rates on the PDP-11s are essentially identical, and the Honeywell rate is not far off when measured in blocks per second. No general statistical significance can be ascribed to this little experiment. Seek time, for example, dominates the measured times (because the disks on the PDP-11 transfer one block of data in only .6 millisecond once positioned) and there was no attempt to optimize the placement of the input or output files. The results do seem to suggest, however, that the very flexible scheme for representing UNIX files carries no great cost compared with at least two other systems.

The real time per block of I/O observed under UNIX in this test was about 22 milliseconds. Because the system overhead per block is 6 milliseconds, most of which is overlapped, it would seem that the overall transfer rate of the copy might be nearly doubled if a block size of 1024 bytes were used instead of 512. There are some good arguments against making such a change. For example, space utilization on the disk would suffer noticeably: doubling the block size would increase the space occupied by files on the author's machine by about 15 per cent, a number whose importance becomes apparent when we observe that the free space is currently only 5 per cent of the total available. Increasing the block size would also force a decrease in the size of the system's buffer cache and lower its hit rate, but this effect has not been reliably estimated.

Moreover, the copy program is an extreme case in that it is totally I/O bound, with no processing of the data. Most programs do at least look at the data as it goes by; thus to sum the bytes in the file mentioned above required 10 seconds of real time, 5 of which were "user time" spent looking at the bytes. To read the file and ignore it completely required 9 seconds, with negligible user time. It may be concluded that the read-ahead strategy is almost perfectly effective, and that a program that spends as little as 50 microseconds per byte processing its data will not be significantly delayed waiting for I/O (unless, of course, it is competing with other processes for use of the disk).

The basic system interface conceals physical aspects of file storage, such as blocks, tracks, and cylinders. Likewise, the concept of a record is completely absent from the operating system proper and nearly so from the standard software. (By the term "record" we mean an identifiable unit of information consisting either of a fixed number of bytes or of a count together with that number of bytes.) A text file, for example, is stored as a sequence of characters with new-line characters to delimit lines. This form of storage is not only efficient in space when compared with fixed-length records, or even records described by character counts, but is also the most convenient form of storage for the vast majority of text-processing programs, which almost invariably deal with character streams. Most important of all, however, is the fact that there is only one representation of text files. One of the most valuable characteristics of UNIX is the degree to which separate programs interact in useful ways; this interaction would be seriously impaired if there were a variety of representations of the same information.

We recall with a certain horrified fascination a system whose Fortran compiler demanded as input a file with "variable-length" records each of which was required to be 80 bytes long. The prevalence of this sort of nonsense makes the following test of software flexibility (due to M. D. McIlroy) interesting to try when meeting new systems. It consists of writing a Fortran (or PL/I, or other language) program that copies itself to another file, then running the program, and finally attempting to compile the resulting output. Most systems eventually pass, but often only after an expert has been called in to mutter incantations

that convert the data file generated by the Fortran program to the format expected by the Fortran compiler. In sum, we would consider it a grave imposition to require our users or ourselves, when mentioning a file, to specify the form in which it is stored.

For the reasons discussed above, UNIX software does not use the traditional notion of “record” in relation to files, particularly those containing textual information. But certainly there are applications in which the notion has use. A program or self-contained set of programs that generates intermediate files is entitled to use any form of data representation it considers useful. A program that maintains a large data base in which it must frequently look up entries may very well find it convenient to store the entries sequentially, in fixed-size units, sorted by index number. With some changes in the requirements or usual access style, other file organizations become more appropriate. It is straightforward to implement any number of schemes within the UNIX file system precisely because of the uniform, structureless nature of the underlying files; the standard software, however, does not include mechanisms to do it. As an example of what is possible, INGRES [9] is a relational data base manager running under UNIX that supports five different file organizations.

The Structure of the File System

On each file system device such as a disk the accessing information for files is arranged in an array starting at a known place. A file may thus be identified by its device and its index within the device. The internal name of a file is, however, never needed by users or their programs. There is a hierarchically arranged directory structure in which each directory contains a list of names (character strings) and the associated file index, which refers implicitly to the same device as does the directory. Because directories are themselves files, the naming structure is potentially an arbitrary directed graph. Administrative rules restrict it to have the form of a tree, except that non-directory files may have several names (entries in various directories).

A file is named by a sequence of directories separated by “/” leading towards a leaf of the tree. The path specified by a name starting with “/” originates at the root; without an initial “/” the path starts at the current directory. Thus the simple name “x” indicates the entry “x” in the current directory; “/usr/dmr/x” searches the root for directory “usr”, searches it for directory “dmr”, and finally specifies “x” in “dmr”.

When the system is initialized, only one file system device is known (the “root device”); its name is built into the system. More storage is attached by “mounting” other devices, each of which contains its own directory structure. When a device is mounted, its root is attached to a leaf of the already-accessible hierarchy. For example, suppose a device containing a subhierarchy is mounted on the file “/usr”. From then on, the original contents of /usr are hidden from view, and in names of the form “/usr/...” the “...” specifies a path starting at the root of the newly mounted device.

This file system design is inexpensive to implement, is general enough to satisfy most demands, and has a number of virtues: for example, device self-consistency checks are straightforward. It does have a few peculiarities. For example, instantaneously-enforced space quotas, either for users or for directories, are relatively difficult to implement (it has been done at one university site). Perhaps more serious, duplicate names for the same file (“links”) while trivial to provide on a single device, do not work across devices; that is, a directory entry cannot point to a file on another device. Another limitation of the design is that an arbitrary subset of members of a given directory cannot be stored on another device. It is common for the totality of user files to be too voluminous for a given device. It is then impossible for the directories of all users to be members of the same directory, say “/usr”. Instead they must be split into groups, say “/usr1” and “/usr2”; this is somewhat inconvenient, especially when space on one device runs out so that some users must be moved. The data movement can be done expeditiously, but the change in file names from “/usr1/...” to “/usr2/...” is annoying both to those people who must learn the new name and to programs that happen to have such names built into them.

Earlier variants of this file system design stored disk block addresses as 16-bit quantities, which limited the size of a file-system volume to 65,536 blocks. This did not mean that the rest of a larger physical device was wasted, because there could be several logical devices per drive, but the limitation did aggravate the difficulty just mentioned. Recent versions of the system can handle devices with up to about 16 million blocks.

Input/Output Devices

UNIX goes to some pains to efface differences between ordinary disk files and I/O devices such as terminals, tape drives, and line printers. An entry appears in the file system hierarchy for each supported device, so that the structure of device names is the same as that of file names. The same read and write system calls apply to devices and to disk files. Moreover, the same protection mechanisms apply to devices as to files.

Besides the traditionally available devices, names exist for disk devices regarded as physical units outside the file system, and for absolutely addressed memory. The most important device in practice is the user's terminal. Because the terminal channel is treated in the same way as any file (for example, the same I/O calls apply), it is easy to redirect the input and output of commands from the terminal to another file, as explained in the next section. It is also easy to provide inter-user communication.

Some differences are inevitable. For example, the system ordinarily treats terminal input in units of lines, because character-erase and line-delete processing cannot be completed until a full line is typed. Thus if a program attempts to read some large number of bytes from a terminal, it waits until a full line is typed, and then receives a notification that some smaller number of bytes has actually been read. All programs must be prepared for this eventuality in any case, because a read operation from any disk file will return fewer bytes than requested when the end of the file is encountered. Ordinarily, therefore, reads from the terminal are fully compatible with reads from a disk file. A subtle problem can occur if a program reads several bytes, and on the basis of a line of text found therein calls another program to process the remainder of the input. Such a program works successfully when the input source is a terminal, because the input is returned a line at a time, but when the source is an ordinary file the first program may have consumed input intended for the second. At the moment the simplest solution is for the first program to read one character at a time. A more general solution, not implemented, would allow a mode of reading wherein at most one line at a time was returned no matter what the input source.*

The User Interface

The command interpreter, called the "shell," is the most important communication channel between the system and its users. The shell is not part of the operating system, and enjoys no special privileges. A part of the entry for each user in the password file read by the login procedure contains the name of the program that is to be run initially, and for most users that program is the shell. This arrangement is by now commonplace in well-designed systems, but is by no means universal. Among its advantages are the ability to swap the shell even though the kernel is not swappable, so that the size of the shell is not of great concern. It is also easy to replace the shell with another program, either to test a new version or to provide a non-standard interface.

The full language accepted by the shell is moderately complicated, because it performs a number of functions; it is discussed in more detail elsewhere in this issue [10]. Nevertheless, the treatment of individual commands is quite simple and regular: a command is a sequence of words separated by white space (spaces and tabs). The first word is the name of the command, where a command is any executable file. A full name, with "/" characters, may be used to specify the file unambiguously; otherwise, an agreed-upon sequence of directories is searched. The only distinction enjoyed by a system-provided command is that it appears in a directory in the search path of most users. (A very few commands are built into the shell.) The other words making up a command line fall into three types:

- simple strings of characters;
- a file name preceded by "<", ">", or ">>";
- a string containing a file name expansion character.

The simple arguments are passed to the command as an array of strings, and thereafter are interpreted by that program. The fact that the arguments are parsed by the shell and passed as separate strings gives at least a start towards uniformity in the treatment of arguments; we have seen several systems in which

*This suggestion may seem in conflict with our earlier disdain of "records." Not really, because it would only affect the way in which information is read, not the way it is stored. The same bytes would be obtained in either case.

arguments to various commands are separated sometimes by commas, sometimes by semicolons, and sometimes in parentheses; only a manual close at hand or a good memory tells which.

An argument beginning with "<" is taken to name a file that is to be opened by the shell and associated with the *standard input* of the command, namely the stream from which programs ordinarily read input; in the absence of such an argument, the standard input is attached to the terminal. Correspondingly, a file whose name is prefixed by ">" receives the standard output of commands; ">>" designates a variant in which the output is appended to the file instead of replacing it. For this mechanism to work it is necessary that I/O to a terminal be compatible with I/O to a file; the point here is that the redirection is specified in the shell language, in a convenient and natural notation, so that it is applicable uniformly and without exception to all commands. An argument specifying redirection is not passed to the command, which must go to some trouble even to discover whether redirection has occurred. Other systems support I/O redirection (regrettably, too few), but we know of none with such a convenient notation.

An argument containing a file name expansion character is turned into a sequence of simple arguments that are the names of files. The character "*", for example, means "any sequence of zero or more characters;" the argument "*.*c*" is expanded into a sequence of arguments that are the names of all files in the current directory whose names end with the characters ".*c*". Other expansion characters specify an arbitrary single character in a file name or a range of characters (the digits, say).

Putting this expansion mechanism into the shell has several advantages: the code only appears once, so no space is wasted and commands in general need take no special action; the algorithm is certain to be applied uniformly. The only convention required of commands that process files is to accept a sequence of file arguments even if the elementary action performed applies to only one file at a time. For example, the command that deletes a file could have been coded to accept only a single name, in which case argument expansion would be in vain; in fact, it accepts a sequence of file arguments (however generated) and deletes all of them. Only occasionally is there any difficulty. For example, suppose the command "save" transfers each of its argument files to off-line storage, so "save *" would save everything in the current directory; this works well. But the converse command "restore", which might bring all the named arguments back on-line, will not in general work analogously; "restore *" would bring back only the files that already exist in the current directory (match the "*"), rather than all saved files.

One of the most important contributions of UNIX to programming is the notion of *pipes*, and especially the notation the shell provides for using them. A pipe is, in effect, an open file connecting two processes; information written into one end of the pipe may be read from the other end, with synchronization, scheduling, and buffering handled automatically by the system. A linear array of processes (a "pipeline") thus becomes a set of routines simultaneously processing an I/O stream. The shell notation for a pipeline separates the names of the various programs by a vertical bar, so for example

```
anycommand | sort | pr
```

takes the output of *anycommand*, sorts it, and prints the result in paginated form. The ability to interconnect programs in this way has substantially changed our way of thinking about and writing utility programs in general, and especially those involved with text processing. As a dramatic example, we had three existing programs that would respectively translate characters, sort a file while casting out duplicate lines, and compare two sorted files, publishing lines in the first file but not the second. Combining these with our on-line dictionary gave a pipeline that would print all the words in a document not appearing in the dictionary; in other words, potential spelling mistakes. A simple program to generate plausible derivatives of dictionary words completed the job.

The shell syntax for pipelines forces them to be linear, although the operating system permits processes to be connected by pipes in a general graph. There are several reasons for this restriction. The most important is the lack of a notation as perspicuous as that of the simple, linear pipeline; also, processes connected in a general graph can become deadlocked as the result of the finite amount of buffering in each pipe. Finally, although an acceptable (if complicated) notation has been proposed that creates only deadlock-free graphs, the need has never been felt keenly enough to impel anyone to implement it.

Other aspects of UNIX, not closely tied to any particular program, are also valuable in providing a pleasant user interface. One thing that seems trivial, yet makes a surprising difference once one is used to it, is full-duplex terminal I/O together with read-ahead. Even though programs generally communicate

with the user in terms of lines, rather than single characters, full-duplex terminal I/O means that the user can type at any time, even if the system is typing back, without fear of losing or garbling characters. With read-ahead, one need not wait for a response to every line. A good typist entering a document becomes incredibly frustrated at having to pause before starting each new line; for anyone who knows what he wants to say any slowness in response becomes psychologically magnified if the information must be entered bit-by-bit instead of at full speed.

Both input and output of UNIX programs tends to be very terse. This can be disconcerting, especially to the beginner. The editor, for example, has essentially only one diagnostic, namely “?”, which means “you have done something wrong.” Once one knows the editor, the error or difficulty is usually obvious, and the terseness is appreciated after a period of acclimation, but certainly people can be confused at first. However, even if some fuller diagnostics might be appreciated on occasion, there is much noise that we are happy to be rid of. The command interpreter does not remark loudly that each program finished normally, or announce how much space or time it took; the former fact is whispered by an unobtrusive prompt, and anyone who wishes to know the latter may ask explicitly.

Likewise, commands seldom prompt for missing arguments; instead, if the argument is not optional, they give at most a one-line summary of their usage and terminate. We know of some systems that seem so proud of their ability to interact that they force interaction on the user whether it is wanted or not. Prompting for missing arguments is an issue of taste that can be discussed in calm tones; insistence on asking questions may cause raised voices.

Although the terseness of typical UNIX programs is, to some extent, a matter of taste, it is also connected with the way programs tend to be combined. A simple example should make the situation clear. The command “who” writes out one line for each user logged into the system, giving a name, a terminal name, and the time of login. The command “wc” (for “word count”) writes out the number of lines, the number of words, and the number of characters in its input. Thus

```
who | wc
```

tells in the line-count field how many users are logged in. If “who” produced extraneous verbiage, the count would be off. Worse, if “wc” insisted on determining from its input whether lines, words, or characters were wanted, it could not be used in this pipeline. Certainly, not every command that generates a table should omit headings; nevertheless, we have good reasons to interpret the phrase “extraneous verbiage” rather liberally.

The Environment of a Process

The virtual address space of a process is divided into three regions: a read-only, shared program text region; a writable data area that may grow at one end by explicit request; and a stack that grows at automatically as information is pushed onto it by subroutine calls. The address space contains no “control blocks.”

New processes are created by the “fork” operation, which creates a child process whose code and data are copied from the parent. The child inherits the open files of the parent, and executes asynchronously with it unless the parent explicitly waits for termination of the child. The fork mechanism is essential to the basic operation of the system, because each command executed by the shell runs in its own process. This scheme makes a number of services extremely easy to provide. I/O redirection, in particular, is a basically simple operation; it is performed entirely in the subprocess that executes the command, and thus no memory in the parent command interpreter is required to rescind the change in standard input and output. Background processes likewise require no new mechanism; the shell merely refrains from waiting for the completion of a command specified to be asynchronous. Finally, recursive use of the shell to interpret a sequence of commands stored in a file is in no way a special operation.

Communication by processes with the outside world is restricted to a few paths. Explicit system calls, mostly to do I/O, are the most common. A new program receives a set of character-string arguments from its invoker, and returns a byte of status information when it terminates. It may be sent “signals,” which ordinarily force termination, but may, at the choice of the process, be ignored or cause a simulated hardware interrupt. Interrupts from the terminal, for example, cause a signal to be sent to the processes attached to that terminal; faults such as addressing errors are also turned into signals. Unassigned signals

may be used for communication between cooperating processes. A final, rather specialized, mechanism allows a parent process to trace the actions of its child, receiving notification of faults incurred and accessing the memory of the child. This is used for debugging.

There is thus no general inter-process communication or synchronization scheme. This is a weakness of the system, but it is not felt to be important in most of the uses to which UNIX is put (although, as discussed below, it is very important in other uses). Semaphores, for example, can be implemented by using creation and deletion of a known file to represent the P and V operations. Using a semaphore would certainly be more efficient if the mechanism were made a primitive, but here, as in other aspects of the design, we have preferred to avoid putting into the system new mechanisms that can already be implemented using existing mechanisms. Only when serious and demonstrable inefficiency results is it worth complicating the basic interfaces.

Reliability

The reliability of a system is measured by the absence of unplanned outages, its ability to retain filed information, and the correct functioning of its software.

First, the operating system should not crash. UNIX systems generally have a good, though not impeccable, record for software reliability. The typical period between software crashes (depending somewhat on how much tinkering with the system has been going on recently) is well over a fortnight of continuous operation.

Two events—running out of swap space, and an unrecoverable I/O error during swapping—cause the system to crash “voluntarily,” that is, not as a result of a bug causing a fault. It turns out to be rather inconvenient to arrange a more graceful exit for a process that cannot be swapped. Occurrence of swap-space exhaustion can be made arbitrarily rare by providing enough space, and the current system refuses to create a new process unless there is enough room for it to grow to maximum size. Unrecoverable I/O errors in swapping are usually a signal that the hardware is badly impaired, so in neither of these cases do we feel strongly motivated to alleviate the theoretical problems.

The discussion below points out that overconsumption of resources other than swap space does occur, but generally does not cause a crash, although the system may not be very useful for a period of time. In most such cases a really general remedy is hard to imagine. For example, if one insists on using almost all of the file storage space for storing files, one is certain to run out of file space now and then, and a quota system is unlikely to be of much help, because the space is almost certainly overallocated. An automatically enforced file-space quota would help, however, in the case of the user who accidentally creates a monstrous file, or a monstrous number of small files.

Hardware is by far the most frequent cause of crashes, and in a basically healthy machine, the most frequent difficulty is momentary power dips, which tend to cause disks to go off line and the processor to enter peculiar, undocumented states. Other kinds of failures occur less often. It does seem characteristic of the PDP-11, particularly in large configurations, to develop transient, hard-to-diagnose Unibus maladies. It must be admitted, however, that the system is not very tolerant of malfunctioning hardware, nor does it produce particularly informative diagnostics when trouble occurs.

A reliable system should not lose or corrupt users' files. The operating system does not take any unusual precautions in this regard. Data destined to be written on the disk may remain in an associative memory cache for up to 15 seconds. Nevertheless the author's machine has ruined only three or four files in the past year, not counting files being created at the time of a crash. The rate of destruction of files by the system is negligible compared to that by users who accidentally remove or overwrite them. Nevertheless, the file system is insufficiently redundant to make recovery from a power dip, crash, or momentary hardware malfunction automatic. Frequent dumps guard against disaster (which has occurred—there have been head crashes, and twice a sick disk controller began writing garbage instead of what was asked.)

Security

“Security” means the ability to protect against unwanted accessing or destruction of data and against denial of service to others, for example by causing a crash. UNIX and much of its software were written in a rather open environment, so the continuous, careful effort required to maintain a fully secure system has not always been expended; as a result there are several security problems.

The weakest area is in protecting against crashing, or at least crippling, the operation of the system. Most versions lack checks for overconsumption of certain resources, such as file space, total number of files, and number of processes (which are limited on a per-user basis in more recent versions). Running out of these things does not cause a crash, but will make the system unusable for a period. When resource exhaustion occurs, it is generally evident what happened and who was responsible, so malicious actions are detectable, but the real problem is the accidental program bug.

The theoretical aspects of the situation are brighter in the area of information protection. Each file is marked with its owner and the “group” of users to which the owner belongs. Files also have a set of nine protection bits divided into three sets of three bits specifying permission to read, to write, or execute as a program. The three sets indicate the permissions applicable to the owner of the file, to members of the owner’s group, and to all others.

For directories, the meaning of the access bits are modified: “read” means the ability to read the directory as a file, that is to discover all the names it contains; “execute” means the ability to search a directory for a given name when it appears as part of a qualified name; “write” means ability to create and delete files in that directory, and is unrelated to writing of files in the directory.

This classification is not fine enough to account for the needs of all installations, but is usually adequate. In fact, most installations do not use groups at all (all users are in the same group), and even those that do would be happy to have more possible user IDs and fewer group IDs. (Older versions of the system had only 256 of each; the current system has 65536, however, which should be enough.)

One particular user (the “super-user”) is able to access all files without regard to permissions. This user is also the only one permitted to exercise privileged system entries. It is recognized that the very existence of the notion of a super-user is a theoretical, and often practical, blemish on any protection scheme.

An unusual feature of the protection system is the “set-user-ID” bit. When this bit is on for a file, and the file is executed as a program, the user number used in file permission checking is not that of the person running the program, but that of the owner of the file. In practice, the bit is used to mark the programs that perform the privileged system functions mentioned above (such as creation of directories, changing the owner of a file, and so forth).

In theory, the protection scheme is adequate to maintain security, but in practice breakdowns can easily occur. Most often these come from incorrect protection modes on files. Our software tends to create files that are accessible, even writable, by everyone. This is not an accident, but a reflection of the open environment in which we operate. Nevertheless, people in more hostile situations must adjust modes frequently; it is easy to forget, and in any case there are brief periods when the modes are wrong. It would be better if software created files in a default mode specifiable by each user. The system administrators must be even more careful than the users to apply proper protection. For example, it is easy to write a user program that interprets the contents of a physical disk drive as a file system volume. Unless the special file referring to the disk is protected, the files on it can be accessed in spite of their protection modes. If a set-user-ID file is writable, another user can copy his own program onto it.

It is also possible to take advantage of bugs in privileged set-user-ID programs. For example, the program that sends mail to other users might be given the ability to send to directories that are otherwise protected. If so, this program must be carefully written in order to avoid being fooled into mailing other people’s private files to its invoker.

There are thus a number of practical difficulties in maintaining a fully secure system. Nevertheless, the operating system itself seems capable of maintaining data security. The word “seems” must be used because the system has not been formally verified, yet no security-relevant bugs are known (except the ability to run it out of resources, which was mentioned above). In some ways, in fact, UNIX is inherently safer than many other systems. For example, I/O is always done on open files, which are named by an object

local to a process. Permissions are checked when the file is opened. The I/O calls themselves have as argument only the (local) name of the open file, and the specification of the user's buffer; physical I/O occurs to a system buffer, and the data is copied in or out of the user's address space by a single piece of code in the system. Thus there is no need for complicated, bug-prone verification of device commands and channel programs supplied by the user. Likewise, the absence of user "data control blocks" or other control blocks from the user's address space means that the interface between user processes and the system is rather easily checked, because is conducted by means of explicit arguments.

Use of a High-level Language

UNIX and the preponderance of its software are written in the C language [11]. An introduction to the language appears in this issue [12]. Because UNIX was originally written in assembly language, before C was invented, we are in a better position than most to gauge the effect of using a high-level language on writing systems. Briefly, the effects were remarkably beneficial and the costs minuscule by comparison. The effects cannot be quantized, because we do not measure productivity by lines of code, but it is suggestive to say that UNIX offers a good deal of interesting software, ranging from parser-generators through mathematical equation-formatting packages, that would never have been written at all if their authors had had to write assembly code; many of our most inventive contributors do not know, and do not wish to learn, the instruction set of the machine.

The C versions of programs that were rewritten after C became available are much more easily understood, repaired, and extended than the assembler versions. This applies especially to the operating system itself. The original system was very difficult to modify, especially to add new devices, but also to make even minor changes. The C version is readily modifiable by comparison, and not only by us; more than one university, for example, has completely rewritten the typewriter device driver to suit its own taste. (Paradoxically, the fact that the system is easy to modify causes some annoyance, in the form of variant versions.)

An extremely valuable, though originally unplanned, benefit of writing in C is the portability of the system. The transportation of UNIX from the PDP-11 to the Interdata 8/32 is discussed in another paper [13]. It appears to be possible to produce an operating system and set of software that runs on several machines and whose expression in source code is, except for a few modules, identical on each machine. The payoff from such a system, either to an organization that uses several kinds of hardware or to a manufacturer who produces more than one line of machines, should be evident.

Compared to the benefits, the costs of using a high-level language seem negligible. Certainly the object programs generated by the compiler are somewhat larger than those that would be produced by a careful assembly-language coder. It is hard to estimate the average increase in size, because in rewriting it is difficult to resist the opportunity to redesign somewhat (and usually improve). A typical inflation factor for a well-coded C program would be about 20-40 percent. The decrease in speed is comparable, but can sometimes be larger, mainly because subroutine linkage tends to be more costly in C (just as in other high-level languages) than in assembly programs. However, it is by now a matter of common knowledge that a tiny fraction of the code is likely to consume most of the time, and our experience certainly confirms this belief. A profiling tool for C programs has been useful in making heavily-used programs acceptably fast by directing the programmer's attention to the part of the program where particularly careful coding is worth while.

The above guesses of space and time inflation for C programs are not based on any comprehensive study. Although such a study might be interesting, it would be somewhat irrelevant, in that no matter what the results turned out to be, they would not cause us to start writing assembly language. The operating system and the important programs that run under it are acceptably efficient as they are. This is not to say, of course, that efforts to improve the code generation of the C compiler are in vain. It does mean that we have come to view the operating system itself, as well as other "system programs" such as editors, compilers, and basic utilities, as just as susceptible to expression in a high-level language as are the Fortran codes of numerical mathematics or the Cobol programs of the business world.

In assessing the costs of using C, the cost of the compilations themselves has to be considered. This too we deem acceptable. For example, to compile and link-edit the entire operating system ("sysgen") takes somewhat over nine minutes of clock time (of which seven minutes are CPU time); the system

consists of about 12,500 lines of C code, leading to a rate of about 22 lines per second from source to executable object on a PDP-11/70. The compiler is faster than this figure would indicate; the system source makes heavy use of "include" files, so the actual number of lines processed by the compiler is 38,000 and the rate is 65 lines per second.

These days all the best authorities advocate the use of a high-level language, so UNIX can hardly be accused of starting a revolution with this as its goal. Still, not all of those who actually produce systems have leaped on the bandwagon. Perhaps UNIX can help provide the required nudge. In its largest PDP-11 configurations, it serves 48 simultaneous users (which is about twice the number that the hardware manufacturer's most nearly comparable system handles); in a somewhat cut-down version, still written in C and still recognizable as the same system, it occupies 8K words and supports a single user on the LSI-11 microcomputer.

What UNIX Does Not Do

A number of facilities provided in other systems are not present in UNIX. Many of these things would be useful, or even vital, to some applications—so vital, in fact, that several variant versions of the system, each implementing some subset of the possible facilities mentioned below, are extant. The existence of these variants is in itself a good argument for including the new extensions, perhaps somewhat generalized, in a unified version of the system. At the same time, it is necessary to be convinced that a proposed extension is not merely a too-narrowly conceived, isolated "feature" that will not mesh well with the rest of the system. It is also necessary to realize that the limited address space of the PDP-11, the most common UNIX host, imposes severe constraints on the size of the system.

UNIX is not a "real-time" system in the sense that it is not possible to lock a process in memory so as to guarantee rapid response to events, nor to connect directly to I/O devices. MERT[14], in a sense a generalization of UNIX, does allow these operations and in fact all those mentioned in this section. It is a three-level system, with a kernel, one or more supervisor processes, and user processes. One of the standard supervisor processes is a UNIX emulator, so that all the ordinary UNIX software is available, albeit with somewhat degraded efficiency.

There is no general inter-process message facility, nor even a limited communication scheme such as semaphores. It turns out that the pipe mechanism mentioned above is sufficient to implement whatever communication is needed between closely-related, cooperating processes; "closely-related" means processes with a common ancestor that sets up the communication links. Pipes are not, however, of any use in communicating with daemon processes intended to serve several users. At some of the sites at which UNIX is run a scheme of "named pipes" has been implemented. This involves a named file read by a single process that delays until messages are written into the file by anyone (with permission to do so) who cares to send a message.

Input and output ordinarily appear to be synchronous; programs wait until their I/O is completed. For disk files, read-ahead and write-behind are handled by the operating system. The mechanisms are efficient enough, and the simplification in user-level code large enough, that we have no general doubts about the wisdom of doing things in this way. There remain special applications in which one desires to initiate I/O on several streams and delay until the operation is complete on only one of them. When the number of streams is small, it is possible to simulate this usage with several processes. However, the writers of a UNIX NCP ("network control program") interface to the ARPANET [15] feel that genuinely asynchronous I/O would improve their implementation significantly.

Memory is not shared between processes, except for the (read-only) program text. Partly to alleviate the restrictions on the virtual address space imposed by the PDP-11, and partly to simplify communication among tightly-coupled but asynchronous processes, the controlled sharing of writable data areas would be valuable to some applications. The limited virtual address space available on the PDP-11 turns out to be of particular importance. A number of projects that use UNIX as a base desire better interprocess communication (both by means of messages and by sharing memory) because they are driven to use several processes for a task that logically requires only one. This is true of several Bell System applications and also of INGRES[9].

UNIX does not attempt to assign non-sharable devices to users. Some devices can only be opened by

one process, but there is no mechanism for reserving devices for a particular user for a period of time or over several commands. Few installations with which we have communicated feel this to be a problem. The line printer, for example, is usually dedicated to a spooling program, and its direct use is either forbidden or managed informally. Tapes are always allocated informally. Should the need arise, however, it is worth noting that commands to assign and release devices may be implemented without changing the operating system. Because the same protection mechanism applies to device files as to ordinary files, an "assign" command could operate essentially by changing the owner identification attached to the requested device to that of the invoker for the duration of usage.

Recommendations

The following points are earnestly recommended to designers of operating systems:

- There is really no excuse for not providing a hierarchically-arranged file system. It is very useful for maintaining directories containing related files, it is efficient because the amount of searching for files is bounded, and it is easy to implement.
- The notion of "record" seems to be an obsolete remnant of the days of the 80-column card. A file should consist of a sequence of bytes.
- The greatest care should be taken to ensure that there is only one format for files. This is essential for making programs work smoothly together.
- Systems should be written in a high-level language that encourages portability. Manufacturers who build more than one line of machines and also build more than one operating system and set of utilities are wasting money.

Acknowledgment

Much, even most, of the design and implementation of UNIX is the work of Ken Thompson. My use of the term "we" in this paper is intended to include him; I hope his views have not been misrepresented.

References

1. P. A. Crisman (Ed.) *The Compatible Time-Sharing System*, MIT Press, Cambridge, Mass., 1965
2. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "Unix Time-Sharing System: Document Preparation," Bell System T. J. **57** 6, part 2, July-August 1978.
3. S. C. Johnson, YACC—Another compiler-compiler," Comp Sci Tech Rep #32, Bell Laboratories, (July 1975)
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Bell System T. J. **57** 6, part 2, July-August 1978.
5. K. Thompson, "UNIX Time-Sharing System: Unix Implementation," Bell System T. J. **57** 6, part 2, July-August 1978.
6. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering, (October 13-15, 1976), pp 164-168.
7. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," Bell System T. J. **57** 6, part 2, July-August 1978.
8. R. J. Feiertag and E. I. Organick, "The Multics Input-output System," Proc. Third Symposium on Operating Systems Principles (Oct. 18-20, 1971) pp 18-20, ACM, New York.
9. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES," J. ACM Trans. on Database Systems **1** 3, Sept. 1976 pp. 189-222.
10. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," Bell System T. J. **57** 6, part 2, July-August 1978.
11. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
12. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The

- C Programming Language," Bell System T. J. **57** 6, part 2, July-August 1978.
- 13. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of Programs and the UNIX System," Bell System T. J. **57** 6, part 2, July-August 1978.
 - 14. D. L. Bayer, H. Lycklama, "MERT - A Multi-environment Real-time Operating System," Proc. Fifth Symposium on Operating System Principles, Nov. 19-21 1975, pp 33-42, ACM, New York
 - 15. G. L. Chesson, "The Network UNIX System," Operating Systems Review (1975), pp 60-66. Also in Proc. 5th Symp. on Operating Systems Principles.

SECOND EDITION

THE



PROGRAMMING
LANGUAGE

BRIAN W KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Preface	6
Preface to the first edition	8
Chapter 1 - A Tutorial Introduction	9
1.1 Getting Started	9
1.2 Variables and Arithmetic Expressions	11
1.3 The for statement	15
1.4 Symbolic Constants	17
1.5 Character Input and Output	17
1.5.1 File Copying	18
1.5.2 Character Counting	19
1.5.3 Line Counting	20
1.5.4 Word Counting	21
1.6 Arrays	23
1.7 Functions	25
1.8 Arguments - Call by Value	28
1.9 Character Arrays	29
1.10 External Variables and Scope	31
Chapter 2 - Types, Operators and Expressions	35
2.1 Variable Names	35
2.2 Data Types and Sizes	35
2.3 Constants	36
2.4 Declarations	38
2.5 Arithmetic Operators	39
2.6 Relational and Logical Operators	39
2.7 Type Conversions	40
2.8 Increment and Decrement Operators	43
2.9 Bitwise Operators	45
2.10 Assignment Operators and Expressions	46
2.11 Conditional Expressions	47
2.12 Precedence and Order of Evaluation	48
Chapter 3 - Control Flow	50
3.1 Statements and Blocks	50
3.2 If-Else	50
3.3 Else-If	51
3.4 Switch	52
3.5 Loops - While and For	53
3.6 Loops - Do-While	56
3.7 Break and Continue	57
3.8 Goto and labels	57
Chapter 4 - Functions and Program Structure	59
4.1 Basics of Functions	59
4.2 Functions Returning Non-integers	61
4.3 External Variables	63
4.4 Scope Rules	68
4.5 Header Files	69
4.6 Static Variables	70
4.7 Register Variables	71
4.8 Block Structure	71
4.9 Initialization	72
4.10 Recursion	73
4.11 The C Preprocessor	74
4.11.1 File Inclusion	75
4.11.2 Macro Substitution	75

4.11.3 Conditional Inclusion	77
Chapter 5 - Pointers and Arrays	78
5.1 Pointers and Addresses	78
5.2 Pointers and Function Arguments	79
5.3 Pointers and Arrays	81
5.4 Address Arithmetic	84
5.5 Character Pointers and Functions	87
5.6 Pointer Arrays; Pointers to Pointers	89
5.7 Multi-dimensional Arrays	92
5.8 Initialization of Pointer Arrays	93
5.9 Pointers vs. Multi-dimensional Arrays	94
5.10 Command-line Arguments	95
5.11 Pointers to Functions	98
5.12 Complicated Declarations	100
Chapter 6 - Structures	105
6.1 Basics of Structures	105
6.2 Structures and Functions	107
6.3 Arrays of Structures	109
6.4 Pointers to Structures	112
6.5 Self-referential Structures	113
6.6 Table Lookup	117
6.7 Typedef	119
6.8 Unions	120
6.9 Bit-fields	121
Chapter 7 - Input and Output	124
7.1 Standard Input and Output	124
7.2 Formatted Output - printf	125
7.3 Variable-length Argument Lists	127
7.4 Formatted Input - Scanf	128
7.5 File Access	130
7.6 Error Handling - Stderr and Exit	132
7.7 Line Input and Output	134
7.8 Miscellaneous Functions	135
7.8.1 String Operations	135
7.8.2 Character Class Testing and Conversion	135
7.8.3 Ungetc	135
7.8.4 Command Execution	135
7.8.5 Storage Management	136
7.8.6 Mathematical Functions	136
7.8.7 Random Number generation	136
Chapter 8 - The UNIX System Interface	138
8.1 File Descriptors	138
8.2 Low Level I/O - Read and Write	139
8.3 Open, Creat, Close, Unlink	140
8.4 Random Access - Lseek	142
8.5 Example - An implementation of Fopen and Getc	142
8.6 Example - Listing Directories	145
8.7 Example - A Storage Allocator	149
Appendix A - Reference Manual	154
A.1 Introduction	154
A.2 Lexical Conventions	154
A.2.1 Tokens	154
A.2.2 Comments	154

A.2.3 Identifiers	154
A.2.4 Keywords	154
A.2.5 Constants	155
A.2.6 String Literals	156
A.3 Syntax Notation	156
A.4 Meaning of Identifiers	157
A.4.1 Storage Class	157
A.4.2 Basic Types	157
A.4.3 Derived types	158
A.4.4 Type Qualifiers	158
A.5 Objects and Lvalues	158
A.6 Conversions	159
A.6.1 Integral Promotion	159
A.6.2 Integral Conversions	159
A.6.3 Integer and Floating	159
A.6.4 Floating Types	159
A.6.5 Arithmetic Conversions	159
A.6.6 Pointers and Integers	160
A.6.7 Void	160
A.6.8 Pointers to Void	161
A.7 Expressions	161
A.7.1 Pointer Conversion	161
A.7.2 Primary Expressions	161
A.7.3 Postfix Expressions	162
A.7.4 Unary Operators	164
A.7.5 Casts	165
A.7.6 Multiplicative Operators	165
A.7.7 Additive Operators	166
A.7.8 Shift Operators	166
A.7.9 Relational Operators	167
A.7.10 Equality Operators	167
A.7.11 Bitwise AND Operator	167
A.7.12 Bitwise Exclusive OR Operator	167
A.7.13 Bitwise Inclusive OR Operator	168
A.7.14 Logical AND Operator	168
A.7.15 Logical OR Operator	168
A.7.16 Conditional Operator	168
A.7.17 Assignment Expressions	169
A.7.18 Comma Operator	169
A.7.19 Constant Expressions	169
A.8 Declarations	170
A.8.1 Storage Class Specifiers	170
A.8.2 Type Specifiers	171
A.8.3 Structure and Union Declarations	172
A.8.4 Enumerations	174
A.8.5 Declarators	175
A.8.6 Meaning of Declarators	176
A.8.7 Initialization	178
A.8.8 Type names	180
A.8.9 Typedef	181
A.8.10 Type Equivalence	181
A.9 Statements	181
A.9.1 Labeled Statements	182

A.9.2 Expression Statement	182
A.9.3 Compound Statement	182
A.9.4 Selection Statements	183
A.9.5 Iteration Statements	183
A.9.6 Jump statements	184
A.10 External Declarations	184
A.10.1 Function Definitions	185
A.10.2 External Declarations	186
A.11 Scope and Linkage	186
A.11.1 Lexical Scope	187
A.11.2 Linkage	187
A.12 Preprocessing	187
A.12.1 Trigraph Sequences	188
A.12.2 Line Splicing	188
A.12.3 Macro Definition and Expansion	188
A.12.4 File Inclusion	190
A.12.5 Conditional Compilation	191
A.12.6 Line Control	192
A.12.7 Error Generation	192
A.12.8 Pragmas	192
A.12.9 Null directive	192
A.12.10 Predefined names	192
A.13 Grammar	193
Appendix B - Standard Library	199
B.1 Input and Output: <stdio.h>	199
B.1.1 File Operations	199
B.1.2 Formatted Output	200
B.1.3 Formatted Input	202
B.1.4 Character Input and Output Functions	203
B.1.5 Direct Input and Output Functions	204
B.1.6 File Positioning Functions	204
B.1.7 Error Functions	205
B.2 Character Class Tests: <ctype.h>	205
B.3 String Functions: <string.h>	205
B.4 Mathematical Functions: <math.h>	206
B.5 Utility Functions: <stdlib.h>	207
B.6 Diagnostics: <assert.h>	209
B.7 Variable Argument Lists: <stdarg.h>	209
B.8 Non-local Jumps: <setjmp.h>	210
B.9 Signals: <signal.h>	210
B.10 Date and Time Functions: <time.h>	210
B.11 Implementation-defined Limits: <limits.h> and <float.h>	212
Appendix C - Summary of Changes	214

Preface

The computing world has undergone a revolution since the publication of *The C Programming Language* in 1978. Big computers are much bigger, and personal computers have capabilities that rival mainframes of a decade ago. During this time, C has changed too, although only modestly, and it has spread far beyond its origins as the language of the UNIX operating system.

The growing popularity of C, the changes in the language over the years, and the creation of compilers by groups not involved in its design, combined to demonstrate a need for a more precise and more contemporary definition of the language than the first edition of this book provided. In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce ``an unambiguous and machine-independent definition of the language C'', while still retaining its spirit. The result is the ANSI standard for C.

The standard formalizes constructions that were hinted but not described in the first edition, particularly structure assignment and enumerations. It provides a new form of function declaration that permits cross-checking of definition with use. It specifies a standard library, with an extensive set of functions for performing input and output, memory management, string manipulation, and similar tasks. It makes precise the behavior of features that were not spelled out in the original definition, and at the same time states explicitly which aspects of the language remain machine-dependent.

This Second Edition of *The C Programming Language* describes C as defined by the ANSI standard. Although we have noted the places where the language has evolved, we have chosen to write exclusively in the new form. For the most part, this makes no significant difference; the most visible change is the new form of function declaration and definition. Modern compilers already support most features of the standard.

We have tried to retain the brevity of the first edition. C is not a big language, and it is not well served by a big book. We have improved the exposition of critical features, such as pointers, that are central to C programming. We have refined the original examples, and have added new examples in several chapters. For instance, the treatment of complicated declarations is augmented by programs that convert declarations into words and vice versa. As before, all examples have been tested directly from the text, which is in machine-readable form.

Appendix A, the reference manual, is not the standard, but our attempt to convey the essentials of the standard in a smaller space. It is meant for easy comprehension by programmers, but not as a definition for compiler writers -- that role properly belongs to the standard itself. Appendix B is a summary of the facilities of the standard library. It too is meant for reference by programmers, not implementers. Appendix C is a concise summary of the changes from the original version.

As we said in the preface to the first edition, C ``wears well as one's experience with it grows''. With a decade more experience, we still feel that way. We hope that this book will help you learn C and use it well.

We are deeply indebted to friends who helped us to produce this second edition. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson, and Rob Pike gave us perceptive comments on almost every page of draft manuscripts. We are grateful for careful reading by Al Aho, Dennis Allison, Joe Campbell, G.R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford, and Chris van Wyk. We also received helpful suggestions from Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom

London, Jim Reeds, Clovis Tondo, and Peter Weinberger. Dave Prosser answered many detailed questions about the ANSI standard. We used Bjarne Stroustrup's C++ translator extensively for local testing of our programs, and Dave Kristol provided us with an ANSI C compiler for final testing. Rich Drechsler helped greatly with typesetting.

Our sincere thanks to all.

Brian W. Kernighan
Dennis M. Ritchie

Preface to the first edition

C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C is not a ``very high level'' language, nor a ``big'' one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. Production compilers also exist for several other machines, including the IBM System/370, the Honeywell 6000, and the Interdata 8/32. C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.

This book is meant to help the reader learn how to program in C. It contains a tutorial introduction to get new users started as soon as possible, separate chapters on each major feature, and a reference manual. Most of the treatment is based on reading, writing and revising examples, rather than on mere statements of rules. For the most part, the examples are complete, real programs rather than isolated fragments. All examples have been tested directly from the text, which is in machine-readable form. Besides showing how to make effective use of the language, we have also tried where possible to illustrate useful algorithms and principles of good style and sound design.

The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. Nonetheless, a novice programmer should be able to read along and pick up the language, although access to more knowledgeable colleague will help.

In our experience, C has proven to be a pleasant, expressive and versatile language for a wide variety of programs. It is easy to learn, and it wears well as on's experience with it grows. We hope that this book will help you to use it well.

The thoughtful criticisms and suggestions of many friends and colleagues have added greatly to this book and to our pleasure in writing it. In particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy Bill Roome, Bob Rosin and Larry Rosler all read multiple volumes with care. We are also indebted to Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson, and Peter Weinberger for helpful comments at various stages, and to Mile Lesk and Joe Ossanna for invaluable assistance with typesetting.

Brian W. Kernighan
Dennis M. Ritchie

Chapter 1 - A Tutorial Introduction

Let us begin with a quick introduction in C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, rules, and exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). We want to get you as quickly as possible to the point where you can write useful programs, and to do that we have to concentrate on the basics: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output. We are intentionally leaving out of this chapter features of C that are important for writing bigger programs. These include pointers, structures, most of C's rich set of operators, several control-flow statements, and the standard library.

This approach and its drawbacks. Most notable is that the complete story on any particular feature is not found here, and the tutorial, by being brief, may also be misleading. And because the examples do not use the full power of C, they are not as concise and elegant as they might be. We have tried to minimize these effects, but be warned. Another drawback is that later chapters will necessarily repeat some of this chapter. We hope that the repetition will help you more than it annoys.

In any case, experienced programmers should be able to extrapolate from the material in this chapter to their own programming needs. Beginners should supplement it by writing small, similar programs of their own. Both groups can use it as a framework on which to hang the more detailed descriptions that begin in [Chapter 2](#).

1.1 Getting Started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

Print the words

hello, world

This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print ``hello, world" is

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Just how to run this program depends on the system you are using. As a specific example, on the UNIX operating system you must create the program in a file whose name ends in ``.c", such as `hello.c`, then compile it with the command

`cc hello.c`

If you haven't botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called `a.out`. If you run `a.out` by typing the command

`a.out`

it will print

```
hello, world
```

On other systems, the rules will be different; check with a local expert.

Now, for some explanations about the program itself. A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions in Fortran or the procedures and functions of Pascal. Our example is a function named `main`. Normally you are at liberty to give functions whatever names you like, but ```main`'' is special - your program begins executing at the beginning of `main`. This means that every program must have a `main` somewhere.

`main` will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program,

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

tells the compiler to include information about the standard input/output library; the line appears at the beginning of many C source files. The standard library is described in [Chapter 7](#) and [Appendix B](#).

One method of communicating data between functions is for the calling function to provide a list of values, called *arguments*, to the function it calls. The parentheses after the function name surround the argument list. In this example, `main` is defined to be a function that expects no arguments, which is indicated by the empty list ().

```
#include <stdio.h>           include information about standard
library                      define a function called main
main()                        that received no argument values
{                            statements of main are enclosed in braces
    printf("hello, world\n");  main calls library function printf
}                            to print this sequence of characters
                           \n represents the newline character
```

The first C program

The statements of a function are enclosed in braces { }. The function `main` contains only one statement,

```
printf("hello, world\n");
```

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function `printf` with the argument "hello, world\n". `printf` is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like "hello, world\n", is called a *character string* or *string constant*. For the moment our only use of character strings will be as arguments for `printf` and other functions.

The sequence `\n` in the string is C notation for the *newline character*, which when printed advances the output to the left margin on the next line. If you leave out the `\n` (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use `\n` to include a newline character in the `printf` argument; if you try something like

```
printf("hello, world
      ");
```

the C compiler will produce an error message.

`printf` never supplies a newline character automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written

```
#include <stdio.h>

main()
{
    printf("hello, ");
    printf("world");
    printf("\n");
}
```

to produce identical output.

Notice that `\n` represents only a single character. An *escape sequence* like `\n` provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the others that C provides are `\t` for tab, `\b` for backspace, `\"` for the double quote and `\\"` for the backslash itself. There is a complete list in [Section 2.3](#).

Exercise 1-1. Run the ``hello, world'' program on your system. Experiment with leaving out parts of the program, to see what error messages you get.

Exercise 1-2. Experiment to find out what happens when `printf`'s argument string contains `\c`, where *c* is some character not listed above.

1.2 Variables and Arithmetic Expressions

The next program uses the formula ${}^{\circ}\text{C} = (5/9)({}^{\circ}\text{F}-32)$ to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

```

1      -17
20     -6
40      4
60     15
80     26
100    37
120    48
140    60
160    71
180    82
200    93
220   104
240   115
260   126
280   137
300   148

```

The program itself still consists of the definition of a single function named `main`. It is longer than the one that printed ``hello, world'', but not complicated. It introduces several new ideas, including comments, declarations, variables, arithmetic expressions, loops , and formatted output.

```

#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;          /* lower limit of temperature scale */
    upper = 300;         /* upper limit */
    step = 20;           /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}

```

The two lines

```

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

```

are a *comment*, which in this case explains briefly what the program does. Any characters between `/*` and `*/` are ignored by the compiler; they may be used freely to make a program easier to understand. Comments may appear anywhere where a blank, tab or newline can.

In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements. A *declaration* announces the properties of variables; it consists of a name and a list of variables, such as

```

int fahr, celsius;
int lower, upper, step;

```

The type `int` means that the variables listed are integers; by contrast with `float`, which means floating point, i.e., numbers that may have a fractional part. The range of both `int` and `float` depends on the machine you are using; 16-bit `ints`, which lie between -32768 and +32767, are common, as are 32-bit `ints`. A `float` number is typically a 32-bit quantity, with at least six significant digits and magnitude generally between about 10^{-38} and 10^{38} .

C provides several other data types besides `int` and `float`, including:

char	character - a single byte
short	short integer
long	long integer
double	double-precision floating point

The size of these objects is also machine-dependent. There are also *arrays*, *structures* and *unions* of these basic types, *pointers* to them, and *functions* that return them, all of which we will meet in due course.

Computation in the temperature conversion program begins with the *assignment statements*

```
lower = 0;
upper = 300;
step = 20;
```

which set the variables to their initial values. Individual statements are terminated by semicolons.

Each line of the table is computed the same way, so we use a loop that repeats once per output line; this is the purpose of the `while` loop

```
while (fahr <= upper) {
    ...
}
```

The `while` loop operates as follows: The condition in parentheses is tested. If it is true (`fahr` is less than or equal to `upper`), the body of the loop (the three statements enclosed in braces) is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false (`fahr` exceeds `upper`) the loop ends, and execution continues at the statement that follows the loop. There are no further statements in this program, so it terminates.

The body of a `while` can be one or more statements enclosed in braces, as in the temperature converter, or a single statement without braces, as in

```
while (i < j)
    i = 2 * i;
```

In either case, we will always indent the statements controlled by the `while` by one tab stop (which we have shown as four spaces) so you can see at a glance which statements are inside the loop. The indentation emphasizes the logical structure of the program. Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read. We recommend writing only one statement per line, and using blanks around operators to clarify grouping. The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently.

Most of the work gets done in the body of the loop. The Celsius temperature is computed and assigned to the variable `celsius` by the statement

```
celsius = 5 * (fahr-32) / 9;
```

The reason for multiplying by 5 and dividing by 9 instead of just multiplying by $5/9$ is that in C, as in many other languages, integer division *truncates*: any fractional part is discarded. Since 5 and 9 are integers, $5/9$ would be truncated to zero and so all the Celsius temperatures would be reported as zero.

This example also shows a bit more of how `printf` works. `printf` is a general-purpose output formatting function, which we will describe in detail in [Chapter 7](#). Its first argument is a string of characters to be printed, with each % indicating where one of the other (second, third,

...) arguments is to be substituted, and in what form it is to be printed. For instance, %d specifies an integer argument, so the statement

```
printf("%d\t%d\n", fahr, celsius);
```

causes the values of the two integers fahr and celsius to be printed, with a tab (\t) between them.

Each % construction in the first argument of printf is paired with the corresponding second argument, third argument, etc.; they must match up properly by number and type, or you will get wrong answers.

By the way, printf is not part of the C language; there is no input or output defined in C itself. printf is just a useful function from the standard library of functions that are normally accessible to C programs. The behaviour of printf is defined in the ANSI standard, however, so its properties should be the same with any compiler and library that conforms to the standard.

In order to concentrate on C itself, we don't talk much about input and output until [chapter 7](#). In particular, we will defer formatted input until then. If you have to input numbers, read the discussion of the function scanf in [Section 7.4](#). scanf is like printf, except that it reads input instead of writing output.

There are a couple of problems with the temperature conversion program. The simpler one is that the output isn't very pretty because the numbers are not right-justified. That's easy to fix; if we augment each %d in the printf statement with a width, the numbers printed will be right-justified in their fields. For instance, we might say

```
printf("%3d %6d\n", fahr, celsius);
```

to print the first number of each line in a field three digits wide, and the second in a field six digits wide, like this:

0	-17
20	-6
40	4
60	15
80	26
100	37
...	

The more serious problem is that because we have used integer arithmetic, the Celsius temperatures are not very accurate; for instance, 0°F is actually about -17.8°C, not -17. To get more accurate answers, we should use floating-point arithmetic instead of integer. This requires some changes in the program. Here is the second version:

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300; floating-point version */
main()
{
    float fahr, celsius;
    float lower, upper, step;

    lower = 0;      /* lower limit of temperatuire scale */
    upper = 300;    /* upper limit */
    step = 20;      /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
    }
}
```

```

        fahr = fahr + step;
    }
}

```

This is much the same as before, except that `fahr` and `celsius` are declared to be `float` and the formula for conversion is written in a more natural way. We were unable to use `5/9` in the previous version because integer division would truncate it to zero. A decimal point in a constant indicates that it is floating point, however, so `5.0/9.0` is not truncated because it is the ratio of two floating-point values.

If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done. If we had written `(fahr - 32)`, the `32` would be automatically converted to floating point. Nevertheless, writing floating-point constants with explicit decimal points even when they have integral values emphasizes their floating-point nature for human readers.

The detailed rules for when integers are converted to floating point are in [Chapter 2](#). For now, notice that the assignment

```

fahr = lower;
and the test

```

```
while (fahr <= upper)
```

also work in the natural way - the `int` is converted to `float` before the operation is done.

The `printf` conversion specification `%3.0f` says that a floating-point number (here `fahr`) is to be printed at least three characters wide, with no decimal point and no fraction digits. `%6.1f` describes another number (`celsius`) that is to be printed at least six characters wide, with 1 digit after the decimal point. The output looks like this:

```

0      -17.8
20     -6.7
40      4.4
...

```

Width and precision may be omitted from a specification: `%6f` says that the number is to be at least six characters wide; `%.2f` specifies two characters after the decimal point, but the width is not constrained; and `%f` merely says to print the number as floating point.

<code>%d</code>	print as decimal integer
<code>%6d</code>	print as decimal integer, at least 6 characters wide
<code>%f</code>	print as floating point
<code>%6f</code>	print as floating point, at least 6 characters wide
<code>%.2f</code>	print as floating point, 2 characters after decimal point
<code>%6.2f</code>	print as floating point, at least 6 wide and 2 after decimal point

Among others, `printf` also recognizes `%o` for octal, `%x` for hexadecimal, `%c` for character, `%s` for character string and `%%` for itself.

Exercise 1-3. Modify the temperature conversion program to print a heading above the table.

Exercise 1-4. Write a program to print the corresponding Celsius to Fahrenheit table.

1.3 The for statement

There are plenty of different ways to write a program for a particular task. Let's try a variation on the temperature converter.

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

```

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}

```

This produces the same answers, but it certainly looks different. One major change is the elimination of most of the variables; only `fahr` remains, and we have made it an `int`. The lower and upper limits and the step size appear only as constants in the `for` statement, itself a new construction, and the expression that computes the Celsius temperature now appears as the third argument of `printf` instead of a separate assignment statement.

This last change is an instance of a general rule - in any context where it is permissible to use the value of some type, you can use a more complicated expression of that type. Since the third argument of `printf` must be a floating-point value to match the `%6.1f`, any floating-point expression can occur here.

The `for` statement is a loop, a generalization of the `while`. If you compare it to the earlier `while`, its operation should be clear. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialization

```
fahr = 0
```

is done once, before the loop proper is entered. The second part is the test or condition that controls the loop:

```
fahr <= 300
```

This condition is evaluated; if it is true, the body of the loop (here a single `printf`) is executed. Then the increment step

```
fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become false. As with the `while`, the body of the loop can be a single statement or a group of statements enclosed in braces. The initialization, condition and increment can be any expressions.

The choice between `while` and `for` is arbitrary, based on which seems clearer. The `for` is usually appropriate for loops in which the initialization and increment are single statements and logically related, since it is more compact than `while` and it keeps the loop control statements together in one place.

Exercise 1-5. Modify the temperature conversion program to print the table in reverse order, that is, from 300 degrees to 0.

1.4 Symbolic Constants

A final observation before we leave temperature conversion forever. It's bad practice to bury "magic numbers" like 300 and 20 in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way. One way to deal with magic numbers is to give them meaningful names. A `#define` line defines a *symbolic name or symbolic constant* to be a particular string of characters:

```
#define name replacement list
```

Thereafter, any occurrence of *name* (not in quotes and not part of another name) will be replaced by the corresponding *replacement text*. The *name* has the same form as a variable name: a sequence of letters and digits that begins with a letter. The *replacement text* can be any sequence of characters; it is not limited to numbers.

```
#include <stdio.h>

#define LOWER 0      /* lower limit of table */
#define UPPER 300    /* upper limit */
#define STEP 20     /* step size */

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

The quantities `LOWER`, `UPPER` and `STEP` are symbolic constants, not variables, so they do not appear in declarations. Symbolic constant names are conventionally written in upper case so they can be readily distinguished from lower case variable names. Notice that there is no semicolon at the end of a `#define` line.

1.5 Character Input and Output

We are going to consider a family of related programs for processing character data. You will find that many programs are just expanded versions of the prototypes that we discuss here.

The model of input and output supported by the standard library is very simple. Text input or output, regardless of where it originates or where it goes to, is dealt with as streams of characters. A *text stream* is a sequence of characters divided into lines; each line consists of zero or more characters followed by a newline character. It is the responsibility of the library to make each input or output stream conform this model; the C programmer using the library need not worry about how lines are represented outside the program.

The standard library provides several functions for reading or writing one character at a time, of which `getchar` and `putchar` are the simplest. Each time it is called, `getchar` reads the *next input character* from a text stream and returns that as its value. That is, after

```
c = getchar();
```

the variable `c` contains the next character of input. The characters normally come from the keyboard; input from files is discussed in [Chapter 7](#).

The function `putchar` prints a character each time it is called:

```
putchar(c);
```

prints the contents of the integer variable `c` as a character, usually on the screen. Calls to `putchar` and `printf` may be interleaved; the output will appear in the order in which the calls are made.

1.5.1 File Copying

Given `getchar` and `putchar`, you can write a surprising amount of useful code without knowing anything more about input and output. The simplest example is a program that copies its input to its output one character at a time:

```
read a character
  while (character is not end-of-file indicator)
    output the character just read
    read a character
```

Converting this into C gives:

```
#include <stdio.h>

/* copy input to output; 1st version */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

The relational operator `!=` means ``not equal to''.

What appears to be a character on the keyboard or screen is of course, like everything else, stored internally just as a bit pattern. The type `char` is specifically meant for storing such character data, but any integer type can be used. We used `int` for a subtle but important reason.

The problem is distinguishing the end of input from valid data. The solution is that `getchar` returns a distinctive value when there is no more input, a value that cannot be confused with any real character. This value is called `EOF`, for ``end of file''. We must declare `c` to be a type big enough to hold any value that `getchar` returns. We can't use `char` since `c` must be big enough to hold `EOF` in addition to any possible `char`. Therefore we use `int`.

`EOF` is an integer defined in `<stdio.h>`, but the specific numeric value doesn't matter as long as it is not the same as any `char` value. By using the symbolic constant, we are assured that nothing in the program depends on the specific numeric value.

The program for copying would be written more concisely by experienced C programmers. In C, any assignment, such as

```
c = getchar();
```

is an expression and has a value, which is the value of the left hand side after the assignment. This means that a assignment can appear as part of a larger expression. If the assignment of a character to `c` is put inside the test part of a `while` loop, the copy program can be written this way:

```
#include <stdio.h>

/* copy input to output; 2nd version */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

The `while` gets a character, assigns it to `c`, and then tests whether the character was the end-of-file signal. If it was not, the body of the `while` is executed, printing the character. The `while` then repeats. When the end of the input is finally reached, the `while` terminates and so does `main`.

This version centralizes the input - there is now only one reference to `getchar` - and shrinks the program. The resulting program is more compact, and, once the idiom is mastered, easier to read. You'll see this style often. (It's possible to get carried away and create impenetrable code, however, a tendency that we will try to curb.)

The parentheses around the assignment, within the condition are necessary. The *precedence* of `!=` is higher than that of `=`, which means that in the absence of parentheses the relational test `!=` would be done before the assignment `=`. So the statement

```
c = getchar() != EOF
```

is equivalent to

```
c = (getchar() != EOF)
```

This has the undesired effect of setting `c` to 0 or 1, depending on whether or not the call of `getchar` returned end of file. (More on this in [Chapter 2](#).)

Exercise 1-6. Verify that the expression `getchar() != EOF` is 0 or 1.

Exercise 1-7. Write a program to print the value of `EOF`.

1.5.2 Character Counting

The next program counts characters; it is similar to the copy program.

```
#include <stdio.h>

/* count characters in input; 1st version */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
```

```

    ++nc;
    printf("%ld\n", nc);
}

```

The statement

```
++nc;
```

presents a new operator, `++`, which means *increment by one*. You could instead write `nc = nc + 1` but `++nc` is more concise and often more efficient. There is a corresponding operator `--` to decrement by 1. The operators `++` and `--` can be either prefix operators (`++nc`) or postfix operators (`nc++`); these two forms have different values in expressions, as will be shown in [Chapter 2](#), but `++nc` and `nc++` both increment `nc`. For the moment we will stick to the prefix form.

The character counting program accumulates its count in a `long` variable instead of an `int`. `long` integers are at least 32 bits. Although on some machines, `int` and `long` are the same size, on others an `int` is 16 bits, with a maximum value of 32767, and it would take relatively little input to overflow an `int` counter. The conversion specification `%ld` tells `printf` that the corresponding argument is a `long` integer.

It may be possible to cope with even bigger numbers by using a `double` (double precision `float`). We will also use a `for` statement instead of a `while`, to illustrate another way to write the loop.

```
#include <stdio.h>

/* count characters in input; 2nd version */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

`printf` uses `%f` for both `float` and `double`; `%.0f` suppresses the printing of the decimal point and the fraction part, which is zero.

The body of this `for` loop is empty, because all the work is done in the test and increment parts. But the grammatical rules of C require that a `for` statement have a body. The isolated semicolon, called a *null statement*, is there to satisfy that requirement. We put it on a separate line to make it visible.

Before we leave the character counting program, observe that if the input contains no characters, the `while` or `for` test fails on the very first call to `getchar`, and the program produces zero, the right answer. This is important. One of the nice things about `while` and `for` is that they test at the top of the loop, before proceeding with the body. If there is nothing to do, nothing is done, even if that means never going through the loop body. Programs should act intelligently when given zero-length input. The `while` and `for` statements help ensure that programs do reasonable things with boundary conditions.

1.5.3 Line Counting

The next program counts input lines. As we mentioned above, the standard library ensures that an input text stream appears as a sequence of lines, each terminated by a newline. Hence, counting lines is just counting newlines:

```
#include <stdio.h>

/* count lines in input */
main()
```

```

{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}

```

The body of the `while` now consists of an `if`, which in turn controls the increment `++nl`. The `if` statement tests the parenthesized condition, and if the condition is true, executes the statement (or group of statements in braces) that follows. We have again indented to show what is controlled by what.

The double equals sign `==` is the C notation for ``is equal to'' (like Pascal's single `=` or Fortran's `.EQ.`). This symbol is used to distinguish the equality test from the single `=` that C uses for assignment. A word of caution: newcomers to C occasionally write `=` when they mean `==`. As we will see in [Chapter 2](#), the result is usually a legal expression, so you will get no warning.

A character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a *character constant*, although it is just another way to write a small integer. So, for example, `'A'` is a character constant; in the ASCII character set its value is 65, the internal representation of the character `A`. Of course, `'A'` is to be preferred over 65: its meaning is obvious, and it is independent of a particular character set.

The escape sequences used in string constants are also legal in character constants, so `'\n'` stands for the value of the newline character, which is 10 in ASCII. You should note carefully that `'\n'` is a single character, and in expressions is just an integer; on the other hand, `'\n'` is a string constant that happens to contain only one character. The topic of strings versus characters is discussed further in [Chapter 2](#).

Exercise 1-8. Write a program to count blanks, tabs, and newlines.

Exercise 1-9. Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank.

Exercise 1-10. Write a program to copy its input to its output, replacing each tab by `\t`, each backspace by `\b`, and each backslash by `\\"`. This makes tabs and backspaces visible in an unambiguous way.

1.5.4 Word Counting

The fourth in our series of useful programs counts lines, words, and characters, with the loose definition that a word is any sequence of characters that does not contain a blank, tab or newline. This is a bare-bones version of the UNIX program `wc`.

```

#include <stdio.h>

#define IN   1 /* inside a word */
#define OUT  0 /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {

```

```

++nc;
if (c == '\n')
    ++nl;
if (c == ' ' || c == '\n' || c == '\t')
    state = OUT;
else if (state == OUT) {
    state = IN;
    ++nw;
}
printf("%d %d %d\n", nl, nw, nc);
}

```

Every time the program encounters the first character of a word, it counts one more word. The variable `state` records whether the program is currently in a word or not; initially it is ``not in a word'', which is assigned the value `OUT`. We prefer the symbolic constants `IN` and `OUT` to the literal values 1 and 0 because they make the program more readable. In a program as tiny as this, it makes little difference, but in larger programs, the increase in clarity is well worth the modest extra effort to write it this way from the beginning. You'll also find that it's easier to make extensive changes in programs where magic numbers appear only as symbolic constants.

The line

```
nl = nw = nc = 0;
```

sets all three variables to zero. This is not a special case, but a consequence of the fact that an assignment is an expression with the value and assignments associated from right to left. It's as if we had written

```
nl = (nw = (nc = 0));
```

The operator `||` means OR, so the line

```
if (c == ' ' || c == '\n' || c == '\t')
```

says "if `c` is a blank *or* `c` is a newline *or* `c` is a tab". (Recall that the escape sequence `\t` is a visible representation of the tab character.) There is a corresponding operator `&&` for AND; its precedence is just higher than `||`. Expressions connected by `&&` or `||` are evaluated left to right, and it is guaranteed that evaluation will stop as soon as the truth or falsehood is known. If `c` is a blank, there is no need to test whether it is a newline or tab, so these tests are not made. This isn't particularly important here, but is significant in more complicated situations, as we will soon see.

The example also shows an `else`, which specifies an alternative action if the condition part of an `if` statement is false. The general form is

```
if (expression)
    statement1
else
    statement2
```

One and only one of the two statements associated with an `if-else` is performed. If the *expression* is true, *statement₁* is executed; if not, *statement₂* is executed. Each *statement* can be a single statement or several in braces. In the word count program, the one after the `else` is an `if` that controls two statements in braces.

Exercise 1-11. How would you test the word count program? What kinds of input are most likely to uncover bugs if there are any?

Exercise 1-12. Write a program that prints its input one word per line.

1.6 Arrays

Let us write a program to count the number of occurrences of each digit, of white space characters (blank, tab, newline), and of all other characters. This is artificial, but it permits us to illustrate several aspects of C in one program.

There are twelve categories of input, so it is convenient to use an array to hold the number of occurrences of each digit, rather than ten individual variables. Here is one version of the program:

```

#include <stdio.h>

/* count digits, white space, others */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
           nwhite, nother);
}

```

The output of this program on itself is

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

The declaration

```
int ndigit[10];
```

declares `ndigit` to be an array of 10 integers. Array subscripts always start at zero in C, so the elements are `ndigit[0]`, `ndigit[1]`, ..., `ndigit[9]`. This is reflected in the `for` loops that initialize and print the array.

A subscript can be any integer expression, which includes integer variables like `i`, and integer constants.

This particular program relies on the properties of the character representation of the digits. For example, the test

```
if (c >= '0' && c <= '9')
```

determines whether the character in `c` is a digit. If it is, the numeric value of that digit is

```
c - '0'
```

This works only if '`'0'`', '`'1'`', ..., '`'9'`' have consecutive increasing values. Fortunately, this is true for all character sets.

By definition, `char`s are just small integers, so `char` variables and constants are identical to `int`s in arithmetic expressions. This is natural and convenient; for example `c - '0'` is an integer expression with a value between 0 and 9 corresponding to the character '`'0'`' to '`'9'`' stored in `c`, and thus a valid subscript for the array `ndigit`.

The decision as to whether a character is a digit, white space, or something else is made with the sequence

```

if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;

```

```
else
    ++nother;
```

The pattern

```
if (condition1)
    statement1
else if (condition2)
    statement2
...
...
else
    statementn
```

occurs frequently in programs as a way to express a multi-way decision. The *conditions* are evaluated in order from the top until some *condition* is satisfied; at that point the corresponding *statement* part is executed, and the entire construction is finished. (Any *statement* can be several statements enclosed in braces.) If none of the conditions is satisfied, the *statement* after the final *else* is executed if it is present. If the final *else* and *statement* are omitted, as in the word count program, no action takes place. There can be any number of

```
else if(condition)
    statement
```

groups between the initial *if* and the final *else*.

As a matter of style, it is advisable to format this construction as we have shown; if each *if* were indented past the previous *else*, a long sequence of decisions would march off the right side of the page.

The *switch* statement, to be discussed in [Chapter 4](#), provides another way to write a multi-way branch that is particularly suitable when the condition is whether some integer or character expression matches one of a set of constants. For contrast, we will present a *switch* version of this program in [Section 3.4](#).

Exercise 1-13. Write a program to print a histogram of the lengths of words in its input. It is easy to draw the histogram with the bars horizontal; a vertical orientation is more challenging.

Exercise 1-14. Write a program to print a histogram of the frequencies of different characters in its input.

1.7 Functions

In C, a function is equivalent to a subroutine or function in Fortran, or a procedure or function in Pascal. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore *how* a job is done; knowing *what* is done is sufficient. C makes the use of functions easy, convenient and efficient; you will often see a short function defined and called only once, just because it clarifies some piece of code.

So far we have used only functions like *printf*, *getchar* and *putchar* that have been provided for us; now it's time to write a few of our own. Since C has no exponentiation operator like the **** of Fortran, let us illustrate the mechanics of function definition by writing a function *power(m,n)* to raise an integer *m* to a positive integer power *n*. That is, the value of *power(2,5)* is 32. This function is not a practical exponentiation routine, since it handles only positive powers of small integers, but it's good enough for illustration.(The standard library contains a function *pow(x,y)* that computes x^y .)

Here is the function *power* and a main program to exercise it, so you can see the whole structure at once.

```
#include <stdio.h>

int power(int m, int n);

/* test power function */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

A function definition has this form:

```
return-type function-name(parameter declarations, if any)
{
    declarations
    statements
}
```

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. If the source program appears in several files, you may have to say more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute. For the moment, we will assume that both functions are in the same file, so whatever you have learned about running C programs will still work.

The function `power` is called twice by `main`, in the line

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Each call passes two arguments to `power`, which each time returns an integer to be formatted and printed. In an expression, `power(2,i)` is an integer just as `2` and `i` are. (Not all functions produce an integer value; we will take this up in [Chapter 4](#).)

The first line of `power` itself,

```
int power(int base, int n)
```

declares the parameter types and names, and the type of the result that the function returns. The names used by `power` for its parameters are local to `power`, and are not visible to any other function: other routines can use the same names without conflict. This is also true of the variables `i` and `p`: the `i` in `power` is unrelated to the `i` in `main`.

We will generally use *parameter* for a variable named in the parenthesized list in a function. The terms *formal argument* and *actual argument* are sometimes used for the same distinction.

The value that `power` computes is returned to `main` by the `return`: statement. Any expression may follow `return`:

```
return expression;
```

A function need not return a value; a `return` statement with no expression causes control, but no useful value, to be returned to the caller, as does "falling off the end" of a function by

reaching the terminating right brace. And the calling function can ignore a value returned by a function.

You may have noticed that there is a `return` statement at the end of `main`. Since `main` is a function like any other, it may return a value to its caller, which is in effect the environment in which the program was executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions. In the interests of simplicity, we have omitted `return` statements from our `main` functions up to this point, but we will include them hereafter, as a reminder that programs should return status to their environment.

The declaration

```
int power(int base, int n);
```

just before `main` says that `power` is a function that expects two `int` arguments and returns an `int`. This declaration, which is called a *function prototype*, has to agree with the definition and uses of `power`. It is an error if the definition of a function or any uses of it do not agree with its prototype.

parameter names need not agree. Indeed, parameter names are optional in a function prototype, so for the prototype we could have written

```
int power(int, int);
```

Well-chosen names are good documentation however, so we will often use them.

A note of history: the biggest change between ANSI C and earlier versions is how functions are declared and defined. In the original definition of C, the `power` function would have been written like this:

```

/* power: raise base to n-th power; n >= 0 */
/*          (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}

```

The parameters are named between the parentheses, and their types are declared before opening the left brace; undeclared parameters are taken as `int`. (The body of the function is the same as before.)

The declaration of `power` at the beginning of the program would have looked like this:

```
int power();
```

No parameter list was permitted, so the compiler could not readily check that `power` was being called correctly. Indeed, since by default `power` would have been assumed to return an `int`, the entire declaration might well have been omitted.

The new syntax of function prototypes makes it much easier for a compiler to detect errors in the number of arguments or their types. The old style of declaration and definition still works in ANSI C, at least for a transition period, but we strongly recommend that you use the new form when you have a compiler that supports it.

Exercise 1.15. Rewrite the temperature conversion program of [Section 1.2](#) to use a function for conversion.

1.8 Arguments - Call by Value

One aspect of C functions may be unfamiliar to programmers who are used to some other languages, particularly Fortran. In C, all function arguments are passed ``by value.'' This means that the called function is given the values of its arguments in temporary variables rather than the originals. This leads to some different properties than are seen with ``call by reference'' languages like Fortran or with `var` parameters in Pascal, in which the called routine has access to the original argument, not a local copy.

Call by value is an asset, however, not a liability. It usually leads to more compact programs with fewer extraneous variables, because parameters can be treated as conveniently initialized local variables in the called routine. For example, here is a version of `power` that makes use of this property.

```

/* power: raise base to n-th power; n >= 0; version 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}

```

The parameter `n` is used as a temporary variable, and is counted down (a `for` loop that runs backwards) until it becomes zero; there is no longer a need for the variable `i`. Whatever is done to `n` inside `power` has no effect on the argument that `power` was originally called with.

When necessary, it is possible to arrange for a function to modify a variable in a calling routine. The caller must provide the *address* of the variable to be set (technically a *pointer* to the

variable), and the called function must declare the parameter to be a pointer and access the variable indirectly through it. We will cover pointers in [Chapter 5](#).

The story is different for arrays. When the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array - there is no copying of array elements. By subscripting this value, the function can access and alter any argument of the array. This is the topic of the next section.

1.9 Character Arrays

The most common type of array in C is the array of characters. To illustrate the use of character arrays and functions to manipulate them, let's write a program that reads a set of text lines and prints the longest. The outline is simple enough:

```
while (there's another line)
    if (it's longer than the previous longest)
        (save it)
        (save its length)
print longest line
```

This outline makes it clear that the program divides naturally into pieces. One piece gets a new line, another saves it, and the rest controls the process.

Since things divide so nicely, it would be well to write them that way too. Accordingly, let us first write a separate function `getline` to fetch the next line of input. We will try to make the function useful in other contexts. At the minimum, `getline` has to return a signal about possible end of file; a more useful design would be to return the length of the line, or zero if end of file is encountered. Zero is an acceptable end-of-file return because it is never a valid line length. Every text line has at least one character; even a line containing only a newline has length 1.

When we find a line that is longer than the previous longest line, it must be saved somewhere. This suggests a second function, `copy`, to copy the new line to a safe place.

Finally, we need a main program to control `getline` and `copy`. Here is the result.

```

#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print the longest input line */
main()
{
    int len; /* current line length */
    int max; /* maximum length seen so far */
    char line[MAXLINE]; /* current input line */
    char longest[MAXLINE]; /* longest line saved here */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0) /* there was a line */
        printf("%s", longest);
    return 0;
}

/* getline: read a line into s, return length */
int getline(char s[], int lim)
{
    int c, i;

    for (i=0; i < lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: copy 'from' into 'to'; assume to is big enough */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}

```

The functions `getline` and `copy` are declared at the beginning of the program, which we assume is contained in one file.

`main` and `getline` communicate through a pair of arguments and a returned value. In `getline`, the arguments are declared by the line

```
int getline(char s[], int lim);
```

which specifies that the first argument, `s`, is an array, and the second, `lim`, is an integer. The purpose of supplying the size of an array in a declaration is to set aside storage. The length of an array `s` is not necessary in `getline` since its size is set in `main`. `getline` uses `return` to send a value back to the caller, just as the function `power` did. This line also declares that `getline` returns an `int`; since `int` is the default return type, it could be omitted.

Some functions return a useful value; others, like `copy`, are used only for their effect and return no value. The return type of `copy` is `void`, which states explicitly that no value is returned.

`getline` puts the character '`\0`' (the *null character*, whose value is zero) at the end of the array it is creating, to mark the end of the string of characters. This conversion is also used by the C language: when a string constant like

```
"hello\n"
```

appears in a C program, it is stored as an array of characters containing the characters in the string and terminated with a '`\0`' to mark the end.

<code>h</code>	<code>e</code>	<code>l</code>	<code>l</code>	<code>o</code>	<code>\n</code>	<code>\0</code>
----------------	----------------	----------------	----------------	----------------	-----------------	-----------------

The `%s` format specification in `printf` expects the corresponding argument to be a string represented in this form. `copy` also relies on the fact that its input argument is terminated with a '`\0`', and copies this character into the output.

It is worth mentioning in passing that even a program as small as this one presents some sticky design problems. For example, what should `main` do if it encounters a line which is bigger than its limit? `getline` works safely, in that it stops collecting when the array is full, even if no newline has been seen. By testing the length and the last character returned, `main` can determine whether the line was too long, and then cope as it wishes. In the interests of brevity, we have ignored this issue.

There is no way for a user of `getline` to know in advance how long an input line might be, so `getline` checks for overflow. On the other hand, the user of `copy` already knows (or can find out) how big the strings are, so we have chosen not to add error checking to it.

Exercise 1-16. Revise the main routine of the longest-line program so it will correctly print the length of arbitrary long input lines, and as much as possible of the text.

Exercise 1-17. Write a program to print all input lines that are longer than 80 characters.

Exercise 1-18. Write a program to remove trailing blanks and tabs from each line of input, and to delete entirely blank lines.

Exercise 1-19. Write a function `reverse(s)` that reverses the character string `s`. Use it to write a program that reverses its input a line at a time.

1.10 External Variables and Scope

The variables in `main`, such as `line`, `longest`, etc., are private or local to `main`. Because they are declared within `main`, no other function can have direct access to them. The same is true of the variables in other functions; for example, the variable `i` in `getline` is unrelated to the `i` in `copy`. Each local variable in a function comes into existence only when the function is called, and disappears when the function is exited. This is why such variables are usually known as *automatic variables*, following terminology in other languages. We will use the term *automatic* henceforth to refer to these local variables. ([Chapter 4](#) discusses the `static` storage class, in which local variables do retain their values between calls.)

Because automatic variables come and go with function invocation, they do not retain their values from one call to the next, and must be explicitly set upon each entry. If they are not set, they will contain garbage.

As an alternative to automatic variables, it is possible to define variables that are *external* to all functions, that is, variables that can be accessed by name by any function. (This mechanism is rather like Fortran COMMON or Pascal variables declared in the outermost block.) Because

external variables are globally accessible, they can be used instead of argument lists to communicate data between functions. Furthermore, because external variables remain in existence permanently, rather than appearing and disappearing as functions are called and exited, they retain their values even after the functions that set them have returned.

An external variable must be *defined*, exactly once, outside of any function; this sets aside storage for it. The variable must also be *declared* in each function that wants to access it; this states the type of the variable. The declaration may be an explicit `extern` statement or may be implicit from context. To make the discussion concrete, let us rewrite the longest-line program with `line`, `longest`, and `max` as external variables. This requires changing the calls, declarations, and bodies of all three functions.

```
#include <stdio.h>

#define MAXLINE 1000      /* maximum input line size */

int max;                  /* maximum length seen so far */
char line[MAXLINE];       /* current input line */
char longest[MAXLINE];    /* longest line saved here */

int getline(void);
void copy(void);

/* print longest input line; specialized version */
main()
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0) /* there was a line */
        printf("%s", longest);
    return 0;
}
```

```

/* getline: specialized version */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE - 1
        && (c=getchar) != EOF && c != '\n'; ++i)
        line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: specialized version */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}

```

The external variables in `main`, `getline` and `copy` are defined by the first lines of the example above, which state their type and cause storage to be allocated for them. Syntactically, external definitions are just like definitions of local variables, but since they occur outside of functions, the variables are external. Before a function can use an external variable, the name of the variable must be made known to the function; the declaration is the same as before except for the added keyword `extern`.

In certain circumstances, the `extern` declaration can be omitted. If the definition of the external variable occurs in the source file before its use in a particular function, then there is no need for an `extern` declaration in the function. The `extern` declarations in `main`, `getline` and `copy` are thus redundant. In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all `extern` declarations.

If the program is in several source files, and a variable is defined in `file1` and used in `file2` and `file3`, then `extern` declarations are needed in `file2` and `file3` to connect the occurrences of the variable. The usual practice is to collect `extern` declarations of variables and functions in a separate file, historically called a *header*, that is included by `#include` at the front of each source file. The suffix `.h` is conventional for header names. The functions of the standard library, for example, are declared in headers like `<stdio.h>`. This topic is discussed at length in [Chapter 4](#), and the library itself in [Chapter 7](#) and [Appendix B](#).

Since the specialized versions of `getline` and `copy` have no arguments, logic would suggest that their prototypes at the beginning of the file should be `getline()` and `copy()`. But for compatibility with older C programs the standard takes an empty list as an old-style declaration, and turns off all argument list checking; the word `void` must be used for an explicitly empty list. We will discuss this further in [Chapter 4](#).

You should note that we are using the words *definition* and *declaration* carefully when we refer to external variables in this section. ``Definition'' refers to the place where the variable is created or assigned storage; ``declaration'' refers to places where the nature of the variable is stated but no storage is allocated.

By the way, there is a tendency to make everything in sight an `extern` variable because it appears to simplify communications - argument lists are short and variables are always there when you want them. But external variables are always there even when you don't want them. Relying too heavily on external variables is fraught with peril since it leads to programs whose data connections are not all obvious - variables can be changed in unexpected and even inadvertent ways, and the program is hard to modify. The second version of the longest-line program is inferior to the first, partly for these reasons, and partly because it destroys the generality of two useful functions by writing into them the names of the variables they manipulate.

At this point we have covered what might be called the conventional core of C. With this handful of building blocks, it's possible to write useful programs of considerable size, and it would probably be a good idea if you paused long enough to do so. These exercises suggest programs of somewhat greater complexity than the ones earlier in this chapter.

Exercise 1-20. Write a program `detab` that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every n columns. Should n be a variable or a symbolic parameter?

Exercise 1-21. Write a program `entab` that replaces strings of blanks by the minimum number of tabs and blanks to achieve the same spacing. Use the same tab stops as for `detab`. When either a tab or a single blank would suffice to reach a tab stop, which should be given preference?

Exercise 1-22. Write a program to ``fold'' long input lines into two or more shorter lines after the last non-blank character that occurs before the n -th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.

Exercise 1-23. Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. C comments don't nest.

Exercise 1-24. Write a program to check a C program for rudimentary syntax errors like unmatched parentheses, brackets and braces. Don't forget about quotes, both single and double, escape sequences, and comments. (This program is hard if you do it in full generality.)

Chapter 2 - Types, Operators and Expressions

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it. These building blocks are the topics of this chapter.

The ANSI standard has made many small changes and additions to basic types and expressions. There are now signed and unsigned forms of all integer types, and notations for unsigned constants and hexadecimal character constants. Floating-point operations may be done in single precision; there is also a long double type for extended precision. String constants may be concatenated at compile time. Enumerations have become part of the language, formalizing a feature of long standing. Objects may be declared const, which prevents them from being changed. The rules for automatic coercions among arithmetic types have been augmented to handle the richer set of types.

2.1 Variable Names

Although we didn't say so in [Chapter 1](#), there are some restrictions on the names of variables and symbolic constants. Names are made up of letters and digits; the first character must be a letter. The underscore ``_'' counts as a letter; it is sometimes useful for improving the readability of long variable names. Don't begin variable names with underscore, however, since library routines often use such names. Upper and lower case letters are distinct, so x and X are two different names. Traditional C practice is to use lower case for variable names, and all upper case for symbolic constants.

At least the first 31 characters of an internal name are significant. For function names and external variables, the number may be less than 31, because external names may be used by assemblers and loaders over which the language has no control. For external names, the standard guarantees uniqueness only for 6 characters and a single case. Keywords like if, else, int, float, etc., are reserved: you can't use them as variable names. They must be in lower case.

It's wise to choose variable names that are related to the purpose of the variable, and that are unlikely to get mixed up typographically. We tend to use short names for local variables, especially loop indices, and longer names for external variables.

2.2 Data Types and Sizes

There are only a few basic data types in C:

char	a single byte, capable of holding one character in the local character set
int	an integer, typically reflecting the natural size of integers on the host machine
float	single-precision floating point
double	double-precision floating point

In addition, there are a number of qualifiers that can be applied to these basic types. short and long apply to integers:

```
short int sh;
long int counter;
```

The word int can be omitted in such declarations, and typically it is.

The intent is that `short` and `long` should provide different lengths of integers where practical; `int` will normally be the natural size for a particular machine. `short` is often 16 bits long, and `int` either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the restriction that `shorts` and `ints` are at least 16 bits, `longs` are at least 32 bits, and `short` is no longer than `int`, which is no longer than `long`.

The qualifier `signed` or `unsigned` may be applied to `char` or any integer. `unsigned` numbers are always positive or zero, and obey the laws of arithmetic modulo 2^n , where n is the number of bits in the type. So, for instance, if `chars` are 8 bits, `unsigned char` variables have values between 0 and 255, while `signed char` have values between -128 and 127 (in a two's complement machine.) Whether plain `chars` are signed or unsigned is machine-dependent, but printable characters are always positive.

The type `long double` specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; `float`, `double` and `long double` could represent one, two or three distinct sizes.

The standard headers `<limits.h>` and `<float.h>` contain symbolic constants for all of these sizes, along with other properties of the machine and compiler. These are discussed in [Appendix B](#).

Exercise 2-1. Write a program to determine the ranges of `char`, `short`, `int`, and `long` variables, both `signed` and `unsigned`, by printing appropriate values from standard headers and by direct computation. Harder if you compute them: determine the ranges of the various floating-point types.

2.3 Constants

An integer constant like 1234 is an `int`. A `long` constant is written with a terminal `l` (ell) or `L`, as in 123456789`L`; an integer constant too big to fit into an `int` will also be taken as a `long`. Unsigned constants are written with a terminal `u` or `U`, and the suffix `ul` or `UL` indicates `unsigned long`.

Floating-point constants contain a decimal point (123.4) or an exponent (1e-2) or both; their type is `double`, unless suffixed. The suffixes `f` or `F` indicate a `float` constant; `l` or `L` indicate a `long double`.

The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading `0` (zero) on an integer constant means octal; a leading `0x` or `0X` means hexadecimal. For example, decimal 31 can be written as `037` in octal and `0x1f` or `0XF` in hex. Octal and hexadecimal constants may also be followed by `L` to make them `long` and `U` to make them `unsigned`: `0XFUL` is an `unsigned long` constant with value 15 decimal.

A character constant is an integer, written as one character within single quotes, such as '`x`'. The value of a character constant is the numeric value of the character in the machine's character set. For example, in the ASCII character set the character constant '`0`' has the value 48, which is unrelated to the numeric value 0. If we write '`0`' instead of a numeric value like 48 that depends on the character set, the program is independent of the particular value and easier to read. Character constants participate in numeric operations just as any other integers, although they are most often used in comparisons with other characters.

Certain characters can be represented in character and string constants by escape sequences like `\n` (newline); these sequences look like two characters, but represent only one. In addition, an arbitrary byte-sized bit pattern can be specified by

'\ooo'

where *ooo* is one to three octal digits (0...7) or by

'\xhh'

where *hh* is one or more hexadecimal digits (0...9, a...f, A...F). So we might write

```
#define VTAB '\013' /* ASCII vertical tab */
#define BELL '\007' /* ASCII bell character */
```

or, in hexadecimal,

```
#define VTAB '\xb' /* ASCII vertical tab */
#define BELL '\x7' /* ASCII bell character */
```

The complete set of escape sequences is

\a	alert (bell) character	\\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	\ooo	octal number
\t	horizontal tab	\xhh	hexadecimal number
\v	vertical tab		

The character constant '\0' represents the character with value zero, the null character. '\0' is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

A *constant expression* is an expression that involves only constants. Such expressions may be evaluated at during compilation rather than run-time, and accordingly may be used in any place that a constant can occur, as in

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

or

```
#define LEAP 1 /* in leap years */
int days[31+28+LEAP+31+30+31+30+31+31+31+30+31+30+31];
```

A *string constant*, or *string literal*, is a sequence of zero or more characters surrounded by double quotes, as in

"I am a string"

or

"" /* the empty string */

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; \" represents the double-quote character. String constants can be concatenated at compile time:

"hello, " "world"

is equivalent to

"hello, world"

This is useful for splitting up long strings across several source lines.

Technically, a string constant is an array of characters. The internal representation of a string has a null character '\0' at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be, but programs must scan a string completely to determine its length. The standard library function `strlen(s)` returns the length of its character string argument *s*, excluding the terminal '\0'. Here is our version:

```
/* strlen: return length of s */
int strlen(char s[])
{
    int i;

    while (s[i] != '\0')
        ++i;
    return i;
}
```

`strlen` and other string functions are declared in the standard header `<string.h>`.

Be careful to distinguish between a character constant and a string that contains a single character: '`x`' is not the same as "`x`". The former is an integer, used to produce the numeric value of the letter *x* in the machine's character set. The latter is an array of characters that contains one character (the letter *x*) and a '`\0`'.

There is one other kind of constant, the *enumeration constant*. An enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in an `enum` has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
               JUL, AUG, SEP, OCT, NOV, DEC };
               /* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to `#define` with the advantage that the values can be generated for you. Although variables of `enum` types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, enumeration variables offer the chance of checking and so are often better than `#defines`. In addition, a debugger may be able to print values of enumeration variables in their symbolic form.

2.4 Declarations

All variables must be declared before use, although certain declarations can be made implicitly by context. A declaration specifies a type, and contains a list of one or more variables of that type, as in

```
int lower, upper, step;
char c, line[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could well be written as

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

The latter form takes more space, but is convenient for adding a comment to each declaration for subsequent modifications.

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer, as in

```
char esc = '\\';
int i = 0;
int limit = MAXLINE+1;
float eps = 1.0e-5;
```

If the variable in question is not automatic, the initialization is done once only, conceptionally before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression. External and static variables are initialized to zero by default. Automatic variables for which is no explicit initializer have undefined (i.e., garbage) values.

The qualifier `const` can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the `const` qualifier says that the elements will not be altered.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

The `const` declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

The result is implementation-defined if an attempt is made to change a `const`.

2.5 Arithmetic Operators

The binary arithmetic operators are `+`, `-`, `*`, `/`, and the modulus operator `%`. Integer division truncates any fractional part. The expression

```
x % y
```

produces the remainder when `x` is divided by `y`, and thus is zero when `y` divides `x` exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 *are* leap years. Therefore

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d is a leap year\n", year);
else
    printf("%d is not a leap year\n", year);
```

The `%` operator cannot be applied to a `float` or `double`. The direction of truncation for `/` and the sign of the result for `%` are machine-dependent for negative operands, as is the action taken on overflow or underflow.

The binary `+` and `-` operators have the same precedence, which is lower than the precedence of `*`, `/` and `%`, which is in turn lower than unary `+` and `-`. Arithmetic operators associate left to right.

Table 2.1 at the end of this chapter summarizes precedence and associativity for all operators.

2.6 Relational and Logical Operators

The relational operators are

```
>     >=    <     <=
```

They all have the same precedence. Just below them in precedence are the equality operators:

```
==     !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like `i < lim-1` is taken as `i < (lim-1)`, as would be expected.

More interesting are the logical operators `&&` and `||`. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. Most C programs rely on these properties. For example, here is a loop from the input function `getline` that we wrote in [Chapter 1](#):

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Before reading a new character it is necessary to check that there is room to store it in the array `s`, so the test `i < lim-1` *must* be made first. Moreover, if this test fails, we must not go on and read another character.

Similarly, it would be unfortunate if `c` were tested against `EOF` before `getchar` is called; therefore the call and assignment must occur before the character in `c` is tested.

The precedence of `&&` is higher than that of `||`, and both are lower than relational and equality operators, so expressions like

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

need no extra parentheses. But since the precedence of `!=` is higher than assignment, parentheses are needed in

```
(c=getchar()) != '\n'
```

to achieve the desired result of assignment to `c` and then comparison with `'\n'`.

By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false.

The unary negation operator `!` converts a non-zero operand into 0, and a zero operand in 1. A common use of `!` is in constructions like

```
if (!valid)
```

rather than

```
if (valid == 0)
```

It's hard to generalize about which form is better. Constructions like `!valid` read nicely ("if not valid"), but more complicated ones can be hard to understand.

Exercise 2-2. Write a loop equivalent to the `for` loop above without using `&&` or `||`.

2.7 Type Conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a ``narrower'' operand into a ``wider'' one without losing information, such as converting an integer into floating point in an expression like `f + i`. Expressions that don't make sense, like using a `float` as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

A `char` is just a small integer, so `chars` may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function `atoi`, which converts a string of digits into its numeric equivalent.

```
/* atoi: convert s to integer */
int atoi(char s[])
{
    int i, n;
```

```

n = 0;
for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
    n = 10 * n + (s[i] - '0');
return n;
}

```

As we discussed in [Chapter 1](#), the expression

```
s[i] - '0'
```

gives the numeric value of the character stored in `s[i]`, because the values of '0', '1', etc., form a contiguous increasing sequence.

Another example of `char` to `int` conversion is the function `lower`, which maps a single character to lower case *for the ASCII character set*. If the character is not an upper case letter, `lower` returns it unchanged.

```

/* lower: convert c to lower case; ASCII only */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}

```

This works for ASCII because corresponding upper case and lower case letters are a fixed distance apart as numeric values and each alphabet is contiguous -- there is nothing but letters between A and z. This latter observation is not true of the EBCDIC character set, however, so this code would convert more than just letters in EBCDIC.

The standard header `<ctype.h>`, described in [Appendix B](#), defines a family of functions that provide tests and conversions that are independent of character set. For example, the function `tolower` is a portable replacement for the function `lower` shown above. Similarly, the test

```
c >= '0' && c <= '9'
```

can be replaced by

```
isdigit(c)
```

We will use the `<ctype.h>` functions from now on.

There is one subtle point about the conversion of characters to integers. The language does not specify whether variables of type `char` are signed or unsigned quantities. When a `char` is converted to an `int`, can it ever produce a negative integer? The answer varies from machine to machine, reflecting differences in architecture. On some machines a `char` whose leftmost bit is 1 will be converted to a negative integer ("sign extension"). On others, a `char` is promoted to an `int` by adding zeros at the left end, and thus is always positive.

The definition of C guarantees that any character in the machine's standard printing character set will never be negative, so these characters will always be positive quantities in expressions. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others. For portability, specify `signed` or `unsigned` if non-character data is to be stored in `char` variables.

Relational expressions like `i > j` and logical expressions connected by `&&` and `||` are defined to have value 1 if true, and 0 if false. Thus the assignment

```
d = c >= '0' && c <= '9'
```

sets `d` to 1 if `c` is a digit, and 0 if not. However, functions like `isdigit` may return any non-zero value for true. In the test part of `if`, `while`, `for`, etc., "true" just means "non-zero", so this makes no difference.

Implicit arithmetic conversions work much as expected. In general, if an operator like `+` or `*` that takes two operands (a binary operator) has operands of different types, the ``lower'' type is *promoted* to the ``higher'' type before the operation proceeds. The result is of the integer type. [Section 6 of Appendix A states the conversion rules precisely. If there are no `unsigned` operands, however, the following informal set of rules will suffice:](#)

- If either operand is `long double`, convert the other to `long double`.
- Otherwise, if either operand is `double`, convert the other to `double`.
- Otherwise, if either operand is `float`, convert the other to `float`.
- Otherwise, convert `char` and `short` to `int`.
- Then, if either operand is `long`, convert the other to `long`.

Notice that `floats` in an expression are not automatically converted to `double`; this is a change from the original definition. In general, mathematical functions like those in `<math.h>` will use double precision. The main reason for using `float` is to save storage in large arrays, or, less often, to save time on machines where double-precision arithmetic is particularly expensive.

Conversion rules are more complicated when `unsigned` operands are involved. The problem is that comparisons between signed and `unsigned` values are machine-dependent, because they depend on the sizes of the various integer types. For example, suppose that `int` is 16 bits and `long` is 32 bits. Then `-1L < 1U`, because `1U`, which is an `unsigned int`, is promoted to a signed `long`. But `-1L > 1UL` because `-1L` is promoted to `unsigned long` and thus appears to be a large positive number.

Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

A character is converted to an integer, either by sign extension or not, as described above.

Longer integers are converted to shorter ones or to `chars` by dropping the excess high-order bits. Thus in

```
int i;
char c;

i = c;
c = i;
```

the value of `c` is unchanged. This is true whether or not sign extension is involved. Reversing the order of assignments might lose information, however.

If `x` is `float` and `i` is `int`, then `x = i` and `i = x` both cause conversions; `float` to `int` causes truncation of any fractional part. When a `double` is converted to `float`, whether the value is rounded or truncated is implementation dependent.

Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions. In the absence of a function prototype, `char` and `short` become `int`, and `float` becomes `double`. This is why we have declared function arguments to be `int` and `double` even when the function is called with `char` and `float`.

Finally, explicit type conversions can be forced (``coerced'') in any expression, with a unary operator called a `cast`. In the construction

`(type name) expression`

the *expression* is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the *expression* were assigned to a variable of the specified type, which is then used in place of the whole construction. For example, the library routine `sqrt` expects a `double` argument, and will produce nonsense if inadvertently handled something else. (`sqrt` is declared in `<math.h>`.) So if `n` is an integer, we can use

```
sqrt((double) n)
```

to convert the value of `n` to `double` before passing it to `sqrt`. Note that the cast produces the *value* of `n` in the proper type; `n` itself is not altered. The cast operator has the same high precedence as other unary operators, as summarized in the table at the end of this chapter.

If arguments are declared by a function prototype, as they normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for `sqrt`:

```
double sqrt(double)
```

the call

```
root2 = sqrt(2)
```

coerces the integer 2 into the `double` value 2.0 without any need for a cast.

The standard library includes a portable implementation of a pseudo-random number generator and a function for initializing the seed; the former illustrates a cast:

```
unsigned long int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Exercise 2-3. Write a function `htoi(s)`, which converts a string of hexadecimal digits (including an optional `0x` or `0X`) into its equivalent integer value. The allowable digits are 0 through 9, a through f, and A through F.

2.8 Increment and Decrement Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator `++` adds 1 to its operand, while the decrement operator `--` subtracts 1. We have frequently used `++` to increment variables, as in

```
if (c == '\n')
    ++nl;
```

The unusual aspect is that `++` and `--` may be used either as prefix operators (before the variable, as in `++n`), or postfix operators (after the variable: `n++`). In both cases, the effect is to increment `n`. But the expression `++n` increments `n` *before* its value is used, while `n++` increments `n` *after* its value has been used. This means that in a context where the value is being used, not just the effect, `++n` and `n++` are different. If `n` is 5, then

```
x = n++;
```

sets `x` to 5, but

```
x = ++n;
```

sets `x` to 6. In both cases, `n` becomes 6. The increment and decrement operators can only be applied to variables; an expression like `(i+j)++` is illegal.

In a context where no value is wanted, just the incrementing effect, as in

```
if (c == '\n')
    nl++;
```

prefix and postfix are the same. But there are situations where one or the other is specifically called for. For instance, consider the function `squeeze(s,c)`, which removes all occurrences of the character `c` from the string `s`.

```
/* squeeze: delete all c from s */
void squeeze(char s[], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Each time a non-`c` occurs, it is copied into the current `j` position, and only then is `j` incremented to be ready for the next character. This is exactly equivalent to

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Another example of a similar construction comes from the `getline` function that we wrote in [Chapter 1](#), where we can replace

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

by the more compact

```
if (c == '\n')
    s[i++] = c;
```

As a third example, consider the standard function `strcat(s,t)`, which concatenates the string `t` to the end of string `s`. `strcat` assumes that there is enough space in `s` to hold the combination. As we have written it, `strcat` returns no value; the standard library version returns a pointer to the resulting string.

```
/* strcat: concatenate t to end of s; s must be big enough */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0') /* find end of s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copy t */
        ;
}
```

As each member is copied from `t` to `s`, the postfix `++` is applied to both `i` and `j` to make sure that they are in position for the next pass through the loop.

Exercise 2-4. Write an alternative version of `squeeze(s1,s2)` that deletes each character in `s1` that matches any character in the *string* `s2`.

Exercise 2-5. Write the function `any(s1,s2)`, which returns the first location in a string `s1` where any character from the string `s2` occurs, or `-1` if `s1` contains no characters from `s2`. (The standard library function `strpbrk` does the same job but returns a pointer to the location.)

2.9 Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, `char`, `short`, `int`, and `long`, whether signed or unsigned.

&	bitwise AND
	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

The bitwise AND operator `&` is often used to mask off some set of bits, for example

```
n = n & 0177;
```

sets to zero all but the low-order 7 bits of `n`.

The bitwise OR operator `|` is used to turn bits on:

```
x = x | SET_ON;
```

sets to one in `x` the bits that are set to one in `SET_ON`.

The bitwise exclusive OR operator `^` sets a one in each bit position where its operands have different bits, and zero where they are the same.

One must distinguish the bitwise operators `&` and `|` from the logical operators `&&` and `||`, which imply left-to-right evaluation of a truth value. For example, if `x` is 1 and `y` is 2, then `x & y` is zero while `x && y` is one.

The shift operators `<<` and `>>` perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be non-negative. Thus `x << 2` shifts the value of `x` by two positions, filling vacated bits with zero; this is equivalent to multiplication by 4. Right shifting an unsigned quantity always fits the vacated bits with zero. Right shifting a signed quantity will fill with bit signs ("arithmetic shift") on some machines and with 0-bits ("logical shift") on others.

The unary operator `~` yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example

```
x = x & ~077
```

sets the last six bits of `x` to zero. Note that `x & ~077` is independent of word length, and is thus preferable to, for example, `x & 0177700`, which assumes that `x` is a 16-bit quantity. The portable form involves no extra cost, since `~077` is a constant expression that can be evaluated at compile time.

As an illustration of some of the bit operators, consider the function `getbits(x,p,n)` that returns the (right adjusted) `n`-bit field of `x` that begins at position `p`. We assume that bit position 0 is at the right end and that `n` and `p` are sensible positive values. For example, `getbits(x,4,3)` returns the three bits in positions 4, 3 and 2, right-adjusted.

```
/* getbits: get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
```

```

        return (x >> (p+1-n)) & ~(~0 << n);
    }
}

```

The expression `x >> (p+1-n)` moves the desired field to the right end of the word. `~0` is all 1-bits; shifting it left `n` positions with `~0<<n` places zeros in the rightmost `n` bits; complementing that with `~` makes a mask with ones in the rightmost `n` bits.

Exercise 2-6. Write a function `setbits(x,p,n,y)` that returns `x` with the `n` bits that begin at position `p` set to the rightmost `n` bits of `y`, leaving the other bits unchanged.

Exercise 2-7. Write a function `invert(x,p,n)` that returns `x` with the `n` bits that begin at position `p` inverted (i.e., 1 changed into 0 and vice versa), leaving the others unchanged.

Exercise 2-8. Write a function `righthrot(x,n)` that returns the value of the integer `x` rotated to the right by `n` positions.

2.10 Assignment Operators and Expressions

An expression such as

```
i = i + 2
```

in which the variable on the left side is repeated immediately on the right, can be written in the compressed form

```
i += 2
```

The operator `+=` is called an *assignment operator*.

Most binary operators (operators like `+` that have a left and right operand) have a corresponding assignment operator `op=`, where `op` is one of

```
+ - * / % << >> & ^ |
```

If `expr1` and `expr2` are expressions, then

```
expr1 op= expr2
```

is equivalent to

```
expr1 = (expr1) op (expr2)
```

except that `expr1` is computed only once. Notice the parentheses around `expr2`:

```
x *= y + 1
```

means

```
x = x * (y + 1)
```

rather than

```
x = x * y + 1
```

As an example, the function `bitcount` counts the number of 1-bits in its integer argument.

```

/* bitcount: count 1 bits in x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}

```

Declaring the argument `x` to be an `unsigned` ensures that when it is right-shifted, vacated bits will be filled with zeros, not sign bits, regardless of the machine the program is run on.

Quite apart from conciseness, assignment operators have the advantage that they correspond better to the way people think. We say ``add 2 to *i*'' or ``increment *i* by 2'', not ``take *i*, add 2, then put the result back in *i*''. Thus the expression *i* += 2 is preferable to *i* = *i*+2. In addition, for a complicated expression like

```
yyval[yypv[p3+p4] + yypv[p1]] += 2
```

the assignment operator makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are indeed the same, or to wonder why they're not. And an assignment operator may even help a compiler to produce efficient code.

We have already seen that the assignment statement has a value and can occur in expressions; the most common example is

```
while ((c = getchar()) != EOF)
    ...
```

The other assignment operators (+=, -=, etc.) can also occur in expressions, although this is less frequent.

In all such expressions, the type of an assignment expression is the type of its left operand, and the value is the value after the assignment.

Exercise 2-9. In a two's complement number system, *x* &= (*x*-1) deletes the rightmost 1-bit in *x*. Explain why. Use this observation to write a faster version of *bitcount*.

2.11 Conditional Expressions

The statements

```
if (a > b)
    z = a;
else
    z = b;
```

compute in *z* the maximum of *a* and *b*. The *conditional expression*, written with the ternary operator ``?:'', provides an alternate way to write this and similar constructions. In the expression

```
expr1 ? expr2 : expr3
```

the expression *expr*₁ is evaluated first. If it is non-zero (true), then the expression *expr*₂ is evaluated, and that is the value of the conditional expression. Otherwise *expr*₃ is evaluated, and that is the value. Only one of *expr*₂ and *expr*₃ is evaluated. Thus to set *z* to the maximum of *a* and *b*,

```
z = (a > b) ? a : b; /* z = max(a, b) */
```

It should be noted that the conditional expression is indeed an expression, and it can be used wherever any other expression can be. If *expr*₂ and *expr*₃ are of different types, the type of the result is determined by the conversion rules discussed earlier in this chapter. For example, if *f* is a *float* and *n* an *int*, then the expression

```
(n > 0) ? f : n
```

is of type *float* regardless of whether *n* is positive.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ?: is very low, just above assignment. They are advisable anyway, however, since they make the condition part of the expression easier to see.

The conditional expression often leads to succinct code. For example, this loop prints *n* elements of an array, 10 per line, with each column separated by one blank, and with each line (including the last) terminated by a newline.

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

A newline is printed after every tenth element, and after the n -th. All other elements are followed by one blank. This might look tricky, but it's more compact than the equivalent `if-else`. Another good example is

```
printf("You have %d items%s.\n", n, n==1 ? "" : "s");
```

Exercise 2-10. Rewrite the function `lower`, which converts upper case letters to lower case, with a conditional expression instead of `if-else`.

2.12 Precedence and Order of Evaluation

Table 2.1 summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence, which is higher than that of binary `+` and `-`. The ``operator'' `()` refers to function call. The operators `->` and `.` are used to access members of structures; they will be covered in [Chapter 6](#), along with `sizeof` (size of an object). [Chapter 5](#) discusses `*` (indirection through a pointer) and `&` (address of an object), and [Chapter 3](#) discusses the comma operator.

Operators	Associativity
<code>() [] -> .</code>	left to right
<code>! ~ ++ -- + - * (type) sizeof</code>	right to left
<code>* / %</code>	left to right
<code>+ -</code>	left to right
<code><< >></code>	left to right
<code>< <= > >=</code>	left to right
<code>== !=</code>	left to right
<code>&</code>	left to right
<code>^</code>	left to right
<code> </code>	left to right
<code>&&</code>	left to right
<code> </code>	left to right
<code>? :</code>	right to left
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	right to left
<code>,</code>	left to right

Unary `&`, `-`, and `*` have higher precedence than the binary forms.

Table 2.1: Precedence and Associativity of Operators

Note that the precedence of the bitwise operators `&`, `^`, and `|` falls below `==` and `!=`. This implies that bit-testing expressions like

```
if ((x & MASK) == 0) ...
```

must be fully parenthesized to give proper results.

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are `&&`, `||`, `?:`, and ````.) For example, in a statement like

```
x = f() + g();
```

`f` may be evaluated before `g` or vice versa; thus if either `f` or `g` alters a variable on which the other depends, `x` can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n)); /* WRONG */
```

can produce different results with different compilers, depending on whether `n` is incremented before `power` is called. The solution, of course, is to write

```
++n;
printf("%d %d\n", n, power(2, n));
```

Function calls, nested assignment statements, and increment and decrement operators cause ``side effects'' - some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated. One unhappy situation is typified by the statement

```
a[i] = i++;
```

The question is whether the subscript is the old value of `i` or the new. Compilers can interpret this in different ways, and generate different answers depending on their interpretation. The standard intentionally leaves most such matters unspecified. When side effects (assignment to variables) take place within an expression is left to the discretion of the compiler, since the best order depends strongly on machine architecture. (The standard does specify that all side effects on arguments take effect before a function is called, but that would not help in the call to `printf` above.)

The moral is that writing code that depends on order of evaluation is a bad programming practice in any language. Naturally, it is necessary to know what things to avoid, but if you don't know *how* they are done on various machines, you won't be tempted to take advantage of a particular implementation.

Chapter 3 - Control Flow

The control-flow of a language specify the order in which computations are performed. We have already met the most common control-flow constructions in earlier examples; here we will complete the set, and be more precise about the ones discussed before.

3.1 Statements and Blocks

An expression such as `x = 0` or `i++` or `printf(...)` becomes a *statement* when it is followed by a semicolon, as in

```
x = 0;
i++;
printf(...);
```

In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal.

Braces `{` and `}` are used to group declarations and statements together into a *compound statement*, or *block*, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an `if`, `else`, `while`, or `for` are another. (Variables can be declared inside *any* block; we will talk about this in [Chapter 4](#).) There is no semicolon after the right brace that ends a block.

3.2 If-Else

The `if-else` statement is used to express decisions. Formally the syntax is

```
if (expression)
    statement1
else
    statement2
```

where the `else` part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), `statement1` is executed. If it is false (*expression* is zero) and if there is an `else` part, `statement2` is executed instead.

Since an `if` tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
if (expression)
instead of
```

```
if (expression != 0)
```

Sometimes this is natural and clear; at other times it can be cryptic.

Because the `else` part of an `if-else` is optional, there is an ambiguity when an `else` is omitted from a nested `if` sequence. This is resolved by associating the `else` with the closest previous `else-less if`. For example, in

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

the `else` goes to the inner `if`, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```

if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;

```

The ambiguity is especially pernicious in situations like this:

```

if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
    else /* WRONG */
        printf("error -- n is negative\n");

```

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the `else` with the inner `if`. This kind of bug can be hard to find; it's a good idea to use braces when there are nested `ifs`.

By the way, notice that there is a semicolon after `z = a` in

```

if (a > b)
    z = a;
else
    z = b;

```

This is because grammatically, a *statement* follows the `if`, and an expression statement like ```z = a;`'' is always terminated by a semicolon.

3.3 Else-If

The construction

```

if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else
    statement

```

occurs so often that it is worth a brief separate discussion. This sequence of `if` statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if an *expression* is true, the *statement* associated with it is executed, and this terminates the whole chain. As always, the code for each *statement* is either a single statement, or a group of them in braces.

The last `else` part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```

else
    statement

```

can be omitted, or it may be used for error checking to catch an ``impossible'' condition.

To illustrate a three-way decision, here is a binary search function that decides if a particular value `x` occurs in the sorted array `v`. The elements of `v` must be in increasing order. The function returns the position (a number between 0 and `n-1`) if `x` occurs in `v`, and -1 if not.

Binary search first compares the input value x to the middle element of the array v . If x is less than the middle value, searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare x to the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.

```
/* binsearch:  find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high)/2;
        if (x < v[mid])
            high = mid + 1;
        else if (x > v[mid])
            low = mid + 1;
        else /* found match */
            return mid;
    }
    return -1; /* no match */
}
```

The fundamental decision is whether x is less than, greater than, or equal to the middle element $v[mid]$ at each step; this is a natural for `else-if`.

Exercise 3-1. Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside.) Write a version with only one test inside the loop and measure the difference in run-time.

3.4 Switch

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```
switch (expression) {
    case const-expr: statements
    case const-expr: statements
    default: statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled `default` is executed if none of the other cases are satisfied. A `default` is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the `default` clause can occur in any order.

In [Chapter 1](#) we wrote a program to count the occurrences of each digit, white space, and all other characters, using a sequence of `if ... else if ... else`. Here is the same program with a `switch`:

```
#include <stdio.h>

main() /* count digits, white space, others */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
```

```

switch (c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        ndigit[c-'0']++;
        break;
    case ' ':
    case '\n':
    case '\t':
        nwhite++;
        break;
    default:
        nother++;
        break;
}
printf("digits =");
for (i = 0; i < 10; i++)
    printf(" %d", ndigit[i]);
printf(", white space = %d, other = %d\n",
       nwhite, nother);
return 0;
}

```

The `break` statement causes an immediate exit from the `switch`. Because cases serve just as labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. `break` and `return` are the most common ways to leave a `switch`. A `break` statement can also be used to force an immediate exit from `while`, `for`, and `do` loops, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a `break` to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented.

As a matter of good form, put a `break` after the last case (the `default` here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

Exercise 3-2. Write a function `escape(s,t)` that converts characters like newline and tab into visible escape sequences like `\n` and `\t` as it copies the string `t` to `s`. Use a `switch`. Write a function for the other direction as well, converting escape sequences into the real characters.

3.5 Loops - While and For

We have already encountered the `while` and `for` loops. In

```

while (expression)
    statement

```

the `expression` is evaluated. If it is non-zero, `statement` is executed and `expression` is re-evaluated. This cycle continues until `expression` becomes zero, at which point execution resumes after `statement`.

The `for` statement

```

for (expr1; expr2; expr3)
    statement

```

is equivalent to

```

expr1;
while (expr2) {

```

```

    statement
    expr3;
}

```

except for the behaviour of `continue`, which is described in [Section 3.7](#).

Grammatically, the three components of a `for` loop are expressions. Most commonly, `expr1` and `expr3` are assignments or function calls and `expr2` is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If `expr1` or `expr3` is omitted, it is simply dropped from the expansion. If the test, `expr2`, is not present, it is taken as permanently true, so

```

for (;;) {
    ...
}

```

is an ``infinite'' loop, presumably to be broken by other means, such as a `break` or `return`.

Whether to use `while` or `for` is largely a matter of personal preference. For example, in

```

while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ; /* skip white space characters */

```

there is no initialization or re-initialization, so the `while` is most natural.

The `for` is preferable when there is a simple initialization and increment since it keeps the loop control statements close together and visible at the top of the loop. This is most obvious in

```

for (i = 0; i < n; i++)
    ...

```

which is the C idiom for processing the first `n` elements of an array, the analog of the Fortran DO loop or the Pascal `for`. The analogy is not perfect, however, since the index variable `i` retains its value when the loop terminates for any reason. Because the components of the `for` are arbitrary expressions, `for` loops are not restricted to arithmetic progressions. Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a `for`, which are better reserved for loop control operations.

As a larger example, here is another version of `atoi` for converting a string to its numeric equivalent. This one is slightly more general than the one in [Chapter 2](#); it copes with optional leading white space and an optional + or - sign. ([Chapter 4](#) shows `atof`, which does the same conversion for floating-point numbers.)

The structure of the program reflects the form of the input:

```

skip white space, if any
get sign, if any
get integer part and convert it

```

Each step does its part, and leaves things in a clean state for the next. The whole process terminates on the first character that could not be part of a number.

```

#include <ctype.h>

/* atoi: convert s to integer; version 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++) /* skip white space */
    ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-') /* skip sign */
        i++;
    ...
}

```

```

    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}

```

The standard library provides a more elaborate function `strtol` for conversion of strings to long integers; see [Section 5 of Appendix B](#).

The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers. The basic idea of this sorting algorithm, which was invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```

/* shellsort:  sort v[0]...v[n-1] into increasing order */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}

```

There are three nested loops. The outermost controls the gap between compared elements, shrinking it from $n/2$ by a factor of two each pass until it becomes zero. The middle loop steps along the elements. The innermost loop compares each pair of elements that is separated by `gap` and reverses any that are out of order. Since `gap` is eventually reduced to one, all elements are eventually ordered correctly. Notice how the generality of the `for` makes the outer loop fit in the same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma `,,', which most often finds use in the `for` statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a `for` statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. This is illustrated in the function `reverse(s)`, which reverses the string `s` in place.

```

#include <string.h>

/* reverse:  reverse string s in place */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

```

The commas that separate function arguments, variables in declarations, etc., are *not* comma operators, and do not guarantee left to right evaluation.

Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the `for` loop in `reverse`, and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for

the exchange of elements in reverse, where the exchange can be thought of a single operation:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Exercise 3-3. Write a function `expand(s1,s2)` that expands shorthand notations like `a-z` in the string `s1` into the equivalent complete list `abc...xyz` in `s2`. Allow for letters of either case and digits, and be prepared to handle cases like `a-b-c` and `a-z0-9` and `-a-z`. Arrange that a leading or trailing `-` is taken literally.

3.6 Loops - Do-While

As we discussed in [Chapter 1](#), the `while` and `for` loops test the termination condition at the top. By contrast, the third loop in C, the `do-while`, tests at the bottom *after* making each pass through the loop body; the body is always executed at least once.

The syntax of the `do` is

```
do
    statement
    while (expression);
```

The *statement* is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, `do-while` is equivalent to the Pascal `repeat-until` statement.

Experience shows that `do-while` is much less used than `while` and `for`. Nonetheless, from time to time it is valuable, as in the following function `itoa`, which converts a number to a character string (the inverse of `atoi`). The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen to generate the string backwards, then reverse it.

```
/* itoa: convert n to characters in s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0) /* record sign */
        n = -n;           /* make n positive */
    i = 0;
    do {                  /* generate digits in reverse order */
        s[i++] = n % 10 + '0'; /* get next digit */
    } while ((n /= 10) > 0); /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

The `do-while` is necessary, or at least convenient, since at least one character must be installed in the array `s`, even if `n` is zero. We also used braces around the single statement that makes up the body of the `do-while`, even though they are unnecessary, so the hasty reader will not mistake the `while` part for the *beginning* of a `while` loop.

Exercise 3-4. In a two's complement number representation, our version of `itoa` does not handle the largest negative number, that is, the value of `n` equal to $-(2^{\text{wordsize}-1})$. Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs.

Exercise 3-5. Write the function `itob(n,s,b)` that converts the integer `n` into a base `b` character representation in the string `s`. In particular, `itob(n,s,16)` formats `s` as a hexadecimal integer in `s`.

Exercise 3-6. Write a version of `itoa` that accepts three arguments instead of two. The third argument is a minimum field width; the converted number must be padded with blanks on the left if necessary to make it wide enough.

3.7 Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The `break` statement provides an early exit from `for`, `while`, and `do`, just as from `switch`. A `break` causes the innermost enclosing loop or `switch` to be exited immediately.

The following function, `trim`, removes trailing blanks, tabs and newlines from the end of a string, using a `break` to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

`strlen` returns the length of the string. The `for` loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when `n` becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters.

The `continue` statement is related to `break`, but less often used; it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin. In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. The `continue` statement applies only to loops, not to `switch`. A `continue` inside a `switch` inside a loop causes the next loop iteration.

As an example, this fragment processes only the non-negative elements in the array `a`; negative values are skipped.

```
for (i = 0; i < n; i++)
    if (a[i] < 0) /* skip negative elements */
        continue;
    ... /* do positive elements */
```

The `continue` statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

3.8 Goto and labels

C provides the infinitely-abusable `goto` statement, and labels to branch to. Formally, the `goto` statement is never necessary, and in practice it is almost always easy to write code without it. We have not used `goto` in this book.

Nevertheless, there are a few situations where `gotos` may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The `break` statement cannot be used directly since it only exits from the innermost loop. Thus:

```
for ( ... )
```

```

    for ( ... ) {
        ...
        if (disaster)
            goto error;
    }
    ...
error:
    /* clean up the mess */

```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several places.

A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the `goto`. The scope of a label is the entire function.

As another example, consider the problem of determining whether two arrays `a` and `b` have an element in common. One possibility is

```

for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        if (a[i] == b[j])
            goto found;
    /* didn't find any common element */
    ...
found:
    /* got one: a[i] == b[j] */
    ...

```

Code involving a `goto` can always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

```

found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* got one: a[i-1] == b[j-1] */
    ...
else
    /* didn't find any common element */
    ...

```

With a few exceptions like those cited here, code that relies on `goto` statements is generally harder to understand and to maintain than code without `gos`. Although we are not dogmatic about the matter, it does seem that `goto` statements should be used rarely, if at all.

Chapter 4 - Functions and Program Structure

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

Function declaration and definition is the area where the ANSI standard has made the most changes to C. As we saw first in [Chapter 1](#), it is now possible to declare the type of arguments when a function is declared. The syntax of function declaration also changes, so that declarations and definitions match. This makes it possible for a compiler to detect many more errors than it could before. Furthermore, when arguments are properly declared, appropriate type coercions are performed automatically.

The standard clarifies the rules on the scope of names; in particular, it requires that there be only one definition of each external object. Initialization is more general: automatic arrays and structures may now be initialized.

The C preprocessor has also been enhanced. New preprocessor facilities include a more complete set of conditional compilation directives, a way to create quoted strings from macro arguments, and better control over the macro expansion process.

4.1 Basics of Functions

To begin with, let us design and write a program to print each line of its input that contains a particular ``pattern'' or string of characters. (This is a special case of the UNIX program `grep`.) For example, searching for the pattern of letters ``ould'' in the set of lines

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

will produce the output

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

The job falls neatly into three pieces:

```
while (there's another line)
    if (the line contains the pattern)
        print it
```

Although it's certainly possible to put the code for all of this in `main`, a better way is to use the structure to advantage by making each part a separate function. Three small pieces are better to deal with than one big one, because irrelevant details can be buried in the functions, and the chance of unwanted interactions is minimized. And the pieces may even be useful in other programs.

``While there's another line'' is `getline`, a function that we wrote in [Chapter 1](#), and ``print it'' is `printf`, which someone has already provided for us. This means we need only write a routine to decide whether the line contains an occurrence of the pattern.

We can solve that problem by writing a function `strindex(s, t)` that returns the position or index in the string `s` where the string `t` begins, or `-1` if `s` does not contain `t`. Because C arrays begin at position zero, indexes will be zero or positive, and so a negative value like `-1` is convenient for signaling failure. When we later need more sophisticated pattern matching, we only have to replace `strindex`; the rest of the code can remain the same. (The standard library provides a function `strstr` that is similar to `strindex`, except that it returns a pointer instead of an index.)

Given this much design, filling in the details of the program is straightforward. Here is the whole thing, so you can see how the pieces fit together. For now, the pattern to be searched for is a literal string, which is not the most general of mechanisms. We will return shortly to a discussion of how to initialize character arrays, and in [Chapter 5](#) will show how to make the pattern a parameter that is set when the program is run. There is also a slightly different version of `getline`; you might find it instructive to compare it to the one in [Chapter 1](#).

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */

int getline(char line[], int max)
int strindex(char source[], char searchfor[]);

char pattern[] = "ould"; /* pattern to search for */

/* find all lines matching pattern */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: get line into s, return length */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            if (j >= MAXLINE)
                return -1;
        if (t[k] == '\0')
            return i;
    }
    return -1;
}
```

```

        ;
if (k > 0 && t[k] == '\0')
    return i;
}
return -1;
}

```

Each function definition has the form

```

return-type function-name(argument declarations)
{
    declarations and statements
}

```

Various parts may be absent; a minimal function is

```
dummy() {}
```

which does nothing and returns nothing. A do-nothing function like this is sometimes useful as a place holder during program development. If the return type is omitted, `int` is assumed.

A program is just a set of definitions of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables. The functions can occur in any order in the source file, and the source program can be split into multiple files, so long as no function is split.

The `return` statement is the mechanism for returning a value from the called function to its caller. Any expression can follow `return`:

```
return expression;
```

The *expression* will be converted to the return type of the function if necessary. Parentheses are often used around the *expression*, but they are optional.

The calling function is free to ignore the returned value. Furthermore, there need to be no expression after `return`; in that case, no value is returned to the caller. Control also returns to the caller with no value when execution ``falls off the end'' of the function by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another. In any case, if a function fails to return a value, its ``value'' is certain to be garbage.

The pattern-searching program returns a status from `main`, the number of matches found. This value is available for use by the environment that called the program

The mechanics of how to compile and load a C program that resides on multiple source files vary from one system to the next. On the UNIX system, for example, the `cc` command mentioned in [Chapter 1](#) does the job. Suppose that the three functions are stored in three files called `main.c`, `getline.c`, and `strindex.c`. Then the command

```
cc main.c getline.c strindex.c
```

compiles the three files, placing the resulting object code in files `main.o`, `getline.o`, and `strindex.o`, then loads them all into an executable file called `a.out`. If there is an error, say in `main.c`, the file can be recompiled by itself and the result loaded with the previous object files, with the command

```
cc main.c getline.o strindex.o
```

The `cc` command uses the ``.c'' versus ``.o'' naming convention to distinguish source files from object files.

Exercise 4-1. Write the function `strindex(s, t)` which returns the position of the *rightmost* occurrence of `t` in `s`, or `-1` if there is none.

4.2 Functions Returning Non-integers

So far our examples of functions have returned either no value (`void`) or an `int`. What if a function must return some other type? many numerical functions like `sqrt`, `sin`, and `cos` return `double`; other specialized functions return other types. To illustrate how to deal with this, let us write and use the function `atof(s)`, which converts the string `s` to its double-precision floating-point equivalent. `atof` is an extension of `atoi`, which we showed versions of in [Chapters 2](#) and [3](#). It handles an optional sign and decimal point, and the presence or absence of either part or fractional part. Our version is *not* a high-quality input conversion routine; that would take more space than we care to use. The standard library includes an `atof`; the header `<stdlib.h>` declares it.

First, `atof` itself must declare the type of value it returns, since it is not `int`. The type name precedes the function name:

```
#include <ctype.h>

/* atof: convert string s to double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++) /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }
    return sign * val / power;
}
```

Second, and just as important, the calling routine must know that `atof` returns a non-`int` value. One way to ensure this is to declare `atof` explicitly in the calling routine. The declaration is shown in this primitive calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded with a sign, and adds them up, printing the running sum after each input:

```
#include <stdio.h>

#define MAXLINE 100

/* rudimentary calculator */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

The declaration

```
double sum, atof(char []);
```

says that `sum` is a `double` variable, and that `atof` is a function that takes one `char[]` argument and returns a `double`.

The function `atof` must be declared and defined consistently. If `atof` itself and the call to it in `main` have inconsistent types in the same source file, the error will be detected by the compiler. But if (as is more likely) `atof` were compiled separately, the mismatch would not be detected, `atof` would return a `double` that `main` would treat as an `int`, and meaningless answers would result.

In the light of what we have said about how declarations must match definitions, this might seem surprising. The reason a mismatch can happen is that if there is no function prototype, a function is implicitly declared by its first appearance in an expression, such as

```
sum += atof(line)
```

If a name that has not been previously declared occurs in an expression and is followed by a left parentheses, it is declared by context to be a function name, the function is assumed to return an `int`, and nothing is assumed about its arguments. Furthermore, if a function declaration does not include arguments, as in

```
double atof();
```

that too is taken to mean that nothing is to be assumed about the arguments of `atof`; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new C programs. If the function takes arguments, declare them; if it takes no arguments, use `void`.

Given `atof`, properly declared, we could write `atoi` (convert a string to `int`) in terms of it:

```
/* atoi: convert string s to integer using atof */
int atoi(char s[])
{
    double atof(char s[]);
    return (int) atof(s);
}
```

Notice the structure of the declarations and the return statement. The value of the expression in

```
return expression;
```

is converted to the type of the function before the return is taken. Therefore, the value of `atof`, a `double`, is converted automatically to `int` when it appears in this `return`, since the function `atoi` returns an `int`. This operation does potentially discard information, however, so some compilers warn of it. The cast states explicitly that the operation is intended, and suppresses any warning.

Exercise 4-2. Extend `atof` to handle scientific notation of the form

```
123.45e-6
```

where a floating-point number may be followed by `e` or `E` and an optionally signed exponent.

4.3 External Variables

A C program consists of a set of external objects, which are either variables or functions. The adjective ``external'' is used in contrast to ``internal'', which describes the arguments and variables defined inside functions. External variables are defined outside of any function, and are thus potentially available to many functions. Functions themselves are always external, because C does not allow functions to be defined inside other functions. By default, external

variables and functions have the property that all references to them by the same name, even from functions compiled separately, are references to the same thing. (The standard calls this property *external linkage*.) In this sense, external variables are analogous to Fortran COMMON blocks or variables in the outermost block in Pascal. We will see later how to define external variables and functions that are visible only within a single source file. Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow.

If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists. As pointed out in [Chapter 1](#), however, this reasoning should be applied with some caution, for it can have a bad effect on program structure, and lead to programs with too many data connections between functions.

External variables are also useful because of their greater scope and lifetime. Automatic variables are internal to a function; they come into existence when the function is entered, and disappear when it is left. External variables, on the other hand, are permanent, so they can retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, it is often most convenient if the shared data is kept in external variables rather than being passed in and out via arguments.

Let us examine this issue with a larger example. The problem is to write a calculator program that provides the operators +, -, *, and /. Because it is easier to implement, the calculator will use reverse Polish notation instead of infix. (Reverse Polish notation is used by some pocket calculators, and in languages like Forth and Postscript.)

In reverse Polish notation, each operator follows its operands; an infix expression like

$(1 - 2) * (4 + 5)$

is entered as

1 2 - 4 5 + *

Parentheses are not needed; the notation is unambiguous as long as we know how many operands each operator expects.

The implementation is simple. Each operand is pushed onto a stack; when an operator arrives, the proper number of operands (two for binary operators) is popped, the operator is applied to them, and the result is pushed back onto the stack. In the example above, for instance, 1 and 2 are pushed, then replaced by their difference, -1. Next, 4 and 5 are pushed and then replaced by their sum, 9. The product of -1 and 9, which is -9, replaces them on the stack. The value on the top of the stack is popped and printed when the end of the input line is encountered.

The structure of the program is thus a loop that performs the proper operation on each operator and operand as it appears:

```

while (next operator or operand is not end-of-file indicator)
    if (number)
        push it
    else if (operator)
        pop operands
        do operation
        push result
    else if (newline)
        pop and print top of stack
    else
        error

```

The operation of pushing and popping a stack are trivial, but by the time error detection and recovery are added, they are long enough that it is better to put each in a separate function than to repeat the code throughout the whole program. And there should be a separate function for fetching the next input operator or operand.

The main design decision that has not yet been discussed is where the stack is, that is, which routines access it directly. One possibility is to keep it in `main`, and pass the stack and the current stack position to the routines that push and pop it. But `main` doesn't need to know about the variables that control the stack; it only does push and pop operations. So we have decided to store the stack and its associated information in external variables accessible to the `push` and `pop` functions but not to `main`.

Translating this outline into code is easy enough. If for now we think of the program as existing in one source file, it will look like this:

```
#includes
#define

function declarations for main

main() { ... }

external variables for push and pop

void push( double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

routines called by getop
```

Later we will discuss how this might be split into two or more source files.

The function `main` is a loop containing a big `switch` on the type of operator or operand; this is a more typical use of `switch` than the one shown in [Section 3.4](#).

```
#include <stdio.h>
#include <stdlib.h> /* for atof() */

#define MAXOP    100 /* max size of operand or operator */
#define NUMBER   '0' /* signal that a number was found */

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
```

```

        break;
    case '-':
        op2 = pop();
        push(pop() - op2);
        break;
    case '/':
        op2 = pop();
        if (op2 != 0.0)
            push(pop() / op2);
        else
            printf("error: zero divisor\n");
        break;
    case '\n':
        printf("\t%.8g\n", pop());
        break;
    default:
        printf("error: unknown command %s\n", s);
        break;
    }
}
return 0;
}

```

Because + and * are commutative operators, the order in which the popped operands are combined is irrelevant, but for - and / the left and right operand must be distinguished. In

```
push(pop() - pop()); /* WRONG */
```

the order in which the two calls of `pop` are evaluated is not defined. To guarantee the right order, it is necessary to pop the first value into a temporary variable as we did in `main`.

```

#define MAXVAL 100 /* maximum depth of val stack */

int sp = 0;           /* next free stack position */
double val[MAXVAL]; /* value stack */

/* push: push f onto value stack */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}

/* pop: pop and return top value from stack */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}

```

A variable is external if it is defined outside of any function. Thus the stack and stack index that must be shared by `push` and `pop` are defined outside these functions. But `main` itself does not refer to the stack or stack position - the representation can be hidden.

Let us now turn to the implementation of `getop`, the function that fetches the next operator or operand. The task is easy. Skip blanks and tabs. If the next character is not a digit or a hexadecimal point, return it. Otherwise, collect a string of digits (which might include a decimal point), and return `NUMBER`, the signal that a number has been collected.

```
#include <ctype.h>
```

```

int getch(void);
void ungetch(int);

/* getop: get next character or numeric operand */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* not a number */
    i = 0;
    if (isdigit(c)) /* collect integer part */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.') /* collect fraction part */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}

```

What are `getch` and `ungetch`? It is often the case that a program cannot determine that it has read enough input until it has read too much. One instance is collecting characters that make up a number: until the first non-digit is seen, the number is not complete. But then the program has read one character too far, a character that it is not prepared for.

The problem would be solved if it were possible to ``un-read'' the unwanted character. Then, every time the program reads one character too many, it could push it back on the input, so the rest of the code could behave as if it had never been read. Fortunately, it's easy to simulate un-getting a character, by writing a pair of cooperating functions. `getch` delivers the next input character to be considered; `ungetch` will return them before reading new input.

How they work together is simple. `ungetch` puts the pushed-back characters into a shared buffer -- a character array. `getch` reads from the buffer if there is anything else, and calls `getchar` if the buffer is empty. There must also be an index variable that records the position of the current character in the buffer.

Since the buffer and the index are shared by `getch` and `ungetch` and must retain their values between calls, they must be external to both routines. Thus we can write `getch`, `ungetch`, and their shared variables as:

```

#define BUFSIZE 100

char buf[BUFSIZE]; /* buffer for ungetch */
int bufp = 0; /* next free position in buf */

int getch(void) /* get a (possibly pushed-back) character */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* push character back on input */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

```

The standard library includes a function `ungetch` that provides one character of pushback; we will discuss it in [Chapter 7](#). We have used an array for the pushback, rather than a single character, to illustrate a more general approach.

Exercise 4-3. Given the basic framework, it's straightforward to extend the calculator. Add the modulus (%) operator and provisions for negative numbers.

Exercise 4-4. Add the commands to print the top elements of the stack without popping, to duplicate it, and to swap the top two elements. Add a command to clear the stack.

Exercise 4-5. Add access to library functions like `sin`, `exp`, and `pow`. See `<math.h>` in [Appendix B, Section 4](#).

Exercise 4-6. Add commands for handling variables. (It's easy to provide twenty-six variables with single-letter names.) Add a variable for the most recently printed value.

Exercise 4-7. Write a routine `ungets(s)` that will push back an entire string onto the input. Should `ungets` know about `buf` and `bufp`, or should it just use `ungetch`?

Exercise 4-8. Suppose that there will never be more than one character of pushback. Modify `getch` and `ungetch` accordingly.

Exercise 4-9. Our `getch` and `ungetch` do not handle a pushed-back `EOF` correctly. Decide what their properties ought to be if an `EOF` is pushed back, then implement your design.

Exercise 4-10. An alternate organization uses `getline` to read an entire input line; this makes `getch` and `ungetch` unnecessary. Revise the calculator to use this approach.

4.4 Scope Rules

The functions and external variables that make up a C program need not all be compiled at the same time; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. Among the questions of interest are

- How are declarations written so that variables are properly declared during compilation?
- How are declarations arranged so that all the pieces will be properly connected when the program is loaded?
- How are declarations organized so there is only one copy?
- How are external variables initialized?

Let us discuss these topics by reorganizing the calculator program into several files. As a practical matter, the calculator is too small to be worth splitting, but it is a fine illustration of the issues that arise in larger programs.

The *scope* of a name is the part of the program within which the name can be used. For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared. Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled. For example, if `main`, `sp`, `val`, `push`, and `pop` are defined in one file, in the order shown above, that is,

```
main() { ... }
```

```

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }

```

then the variables `sp` and `val` may be used in `push` and `pop` simply by naming them; no further declarations are needed. But these names are not visible in `main`, nor are `push` and `pop` themselves.

On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an `extern` declaration is mandatory.

It is important to distinguish between the *declaration* of an external variable and its *definition*. A declaration announces the properties of a variable (primarily its type); a definition also causes storage to be set aside. If the lines

```

int sp;
double val[MAXVAL];

```

appear outside of any function, they *define* the external variables `sp` and `val`, cause storage to be set aside, and also serve as the declarations for the rest of that source file. On the other hand, the lines

```

extern int sp;
extern double val[];

```

declare for the rest of the source file that `sp` is an `int` and that `val` is a `double` array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.

There must be only one *definition* of an external variable among all the files that make up the source program; other files may contain `extern` declarations to access it. (There may also be `extern` declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an `extern` declaration.

Initialization of an external variable goes only with the definition.

Although it is not a likely organization for this program, the functions `push` and `pop` could be defined in one file, and the variables `val` and `sp` defined and initialized in another. Then these definitions and declarations would be necessary to tie them together:

in file1:

```

extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }

```

in file2:

```

int sp = 0;
double val[MAXVAL];

```

Because the `extern` declarations in *file1* lie ahead of and outside the function definitions, they apply to all functions; one set of declarations suffices for all of *file1*. This same organization would also be needed if the definition of `sp` and `val` followed their use in one file.

4.5 Header Files

Let us now consider dividing the calculator program into several source files, as it might be if each of the components were substantially bigger. The `main` function would go in one file,

which we will call `main.c`; `push`, `pop`, and their variables go into a second file, `stack.c`; `getop` goes into a third, `getop.c`. Finally, `getch` and `ungetch` go into a fourth file, `getch.c`; we separate them from the others because they would come from a separately-compiled library in a realistic program.

There is one more thing to worry about - the definitions and declarations shared among files. As much as possible, we want to centralize this, so that there is only one copy to get and keep right as the program evolves. Accordingly, we will place this common material in a *header file*, `calc.h`, which will be included as necessary. (The `#include` line is described in [Section 4.11](#).) The resulting program then looks like this:

calc.h	main.c	getop.c	stack.c	
<pre><code>#define NUMBER '0' void push(double); double pop(void); int getop(char []); int getch(void); void ungetch(int);</code></pre>	<pre><code>#include <stdio.h> #include <stdlib.h> #include "calc.h" #define MAXOP 100 main() { ... }</code></pre>	<pre><code>#include <stdio.h> #include <ctype.h> #include "calc.h" getop() { ... }</code></pre>	<pre><code>#include <stdio.h> #include "calc.h" #define MAXVAL 100 int sp = 0; double val[MAXVAL]; void push(double) { ... } double pop(void) { ... }</code></pre>	
getch.c				
		<pre><code>#include <stdio.h> #define BUFSIZE 100 char buf[BUFSIZE]; int bufp = 0; int getch(void) { ... } void ungetch(int) { ... }</code></pre>		

There is a tradeoff between the desire that each file have access only to the information it needs for its job and the practical reality that it is harder to maintain more header files. Up to some moderate program size, it is probably best to have one header file that contains everything that is to be shared between any two parts of the program; that is the decision we made here. For a much larger program, more organization and more headers would be needed.

4.6 Static Variables

The variables `sp` and `val` in `stack.c`, and `buf` and `bufp` in `getch.c`, are for the private use of the functions in their respective source files, and are not meant to be accessed by anything else. The `static` declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled. External `static` thus provides a way to hide names like `buf` and `bufp` in the `getch-ungetch` combination, which must be external so they can be shared, yet which should not be visible to users of `getch` and `ungetch`.

Static storage is specified by prefixing the normal declaration with the word `static`. If the two routines and the two variables are compiled in one file, as in

```
static char buf[BUFSIZE]; /* buffer for ungetch */
static int bufp = 0;        /* next free position in buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

then no other routine will be able to access `buf` and `bufp`, and those names will not conflict with the same names in other files of the same program. In the same way, the variables that `push` and `pop` use for stack manipulation can be hidden, by declaring `sp` and `val` to be `static`.

The external `static` declaration is most often used for variables, but it can be applied to functions as well. Normally, function names are global, visible to any part of the entire program. If a function is declared `static`, however, its name is invisible outside of the file in which it is declared.

The `static` declaration can also be applied to internal variables. Internal `static` variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal `static` variables provide private, permanent storage within a single function.

Exercise 4-11. Modify `getop` so that it doesn't need to use `ungetch`. Hint: use an internal `static` variable.

4.7 Register Variables

A `register` declaration advises the compiler that the variable in question will be heavily used. The idea is that `register` variables are to be placed in machine registers, which may result in smaller and faster programs. But compilers are free to ignore the advice.

The `register` declaration looks like

```
register int x;
register char c;
```

and so on. The `register` declaration can only be applied to automatic variables and to the formal parameters of a function. In this later case, it looks like

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

In practice, there are restrictions on register variables, reflecting the realities of underlying hardware. Only a few variables in each function may be kept in registers, and only certain types are allowed. Excess register declarations are harmless, however, since the word `register` is ignored for excess or disallowed declarations. And it is not possible to take the address of a register variable (a topic covered in [Chapter 5](#)), regardless of whether the variable is actually placed in a register. The specific restrictions on number and types of register variables vary from machine to machine.

4.8 Block Structure

C is not a block-structured language in the sense of Pascal or similar languages, because functions may not be defined within other functions. On the other hand, variables can be defined in a block-structured fashion within a function. Declarations of variables (including initializations) may follow the left brace that introduces *any* compound statement, not just the one that begins a function. Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching right brace. For example, in

```
if (n > 0) {
    int i; /* declare a new i */
    for (i = 0; i < n; i++)
        ...
}
```

the scope of the variable *i* is the ``true'' branch of the `if`; this *i* is unrelated to any *i* outside the block. An automatic variable declared and initialized in a block is initialized each time the block is entered.

Automatic variables, including formal parameters, also hide external variables and functions of the same name. Given the declarations

```
int x;
int y;

f(double x)
{
    double y;
}
```

then within the function *f*, occurrences of *x* refer to the parameter, which is a `double`; outside *f*, they refer to the external `int`. The same is true of the variable *y*.

As a matter of style, it's best to avoid variable names that conceal names in an outer scope; the potential for confusion and error is too great.

4.9 Initialization

Initialization has been mentioned in passing many times so far, but always peripherally to some other topic. This section summarizes some of the rules, now that we have discussed the various storage classes.

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic and register variables have undefined (i.e., garbage) initial values.

Scalar variables may be initialized when they are defined, by following the name with an equals sign and an expression:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

For external and static variables, the initializer must be a constant expression; the initialization is done once, conceptionally before the program begins execution. For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls. For example, the initialization of the binary search program in [Section 3.3](#) could be written as

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
```

```

    int mid;
    ...
}

```

instead of

```

int low, high, mid;

low = 0;
high = n - 1;

```

In effect, initialization of automatic variables are just shorthand for assignment statements. Which form to prefer is largely a matter of taste. We have generally used explicit assignments, because initializers in declarations are harder to see and further away from the point of use.

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas. For example, to initialize an array `days` with the number of days in each month:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case.

If there are fewer initializers for an array than the specified size, the others will be zero for external, static and automatic variables. It is an error to have too many initializers. There is no way to specify repetition of an initializer, nor to initialize an element in the middle of an array without supplying all the preceding values as well.

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

```
char pattern = "ould";
```

is a shorthand for the longer but equivalent

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

In this case, the array size is five (four characters plus the terminating '\0').

4.10 Recursion

C functions may be used recursively; that is, a function may call itself either directly or indirectly. Consider printing a number as a character string. As we mentioned before, the digits are generated in the wrong order: low-order digits are available before high-order digits, but they have to be printed the other way around.

There are two solutions to this problem. One is to store the digits in an array as they are generated, then print them in the reverse order, as we did with `itoa` in [section 3.6](#). The alternative is a recursive solution, in which `printd` first calls itself to cope with any leading digits, then prints the trailing digit. Again, this version can fail on the largest negative number.

```
#include <stdio.h>

/* printd: print n in decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set. This in `printd(123)` the first `printd` receives the argument `n = 123`. It passes `12` to a second `printd`, which in turn passes `1` to a third. The third-level `printd` prints `1`, then returns to the second level. That `printd` prints `2`, then returns to the first level. That one prints `3` and terminates.

Another good example of recursion is quicksort, a sorting algorithm developed by C.A.R. Hoare in 1962. Given an array, one element is chosen and the others partitioned in two subsets - those less than the partition element and those greater than or equal to it. The same process is then applied recursively to the two subsets. When a subset has fewer than two elements, it doesn't need any sorting; this stops the recursion.

Our version of quicksort is not the fastest possible, but it's one of the simplest. We use the middle element of each subarray for partitioning.

```
/* qsort: sort v[left]...v[right] into increasing order */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right) /* do nothing if array contains */
        return;           /* fewer than two elements */
    swap(v, left, (left + right)/2); /* move partition elem */
    last = left;           /* to v[0] */
    for (i = left + 1; i <= right; i++) /* partition */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);           /* restore partition elem */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

We moved the swapping operation into a separate function `swap` because it occurs three times in `qsort`.

```
/* swap: interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

The standard library includes a version of `qsort` that can sort objects of any type.

Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent. Recursion is especially convenient for recursively defined data structures like trees, we will see a nice example in [Section 6.6](#).

Exercise 4-12. Adapt the ideas of `printd` to write a recursive version of `itoa`; that is, convert an integer into a string by calling a recursive routine.

Exercise 4-13. Write a recursive version of the function `reverse(s)`, which reverses the string `s` in place.

4.11 The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptionally a separate first step in compilation. The two most frequently used features are `#include`, to include the contents of a file during compilation, and `#define`, to replace a token by an arbitrary sequence of characters. Other features described in this section include conditional compilation and macros with arguments.

4.11.1 File Inclusion

File inclusion makes it easy to handle collections of `#defines` and declarations (among other things). Any source line of the form

```
#include "filename"
```

or

```
#include <filename>
```

is replaced by the contents of the file *filename*. If the *filename* is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is enclosed in `<` and `>`, searching follows an implementation-defined rule to find the file. An included file may itself contain `#include` lines.

There are often several `#include` lines at the beginning of a source file, to include common `#define` statements and `extern` declarations, or to access the function prototype declarations for library functions from headers like `<stdio.h>`. (Strictly speaking, these need not be files; the details of how headers are accessed are implementation-dependent.)

`#include` is the preferred way to tie the declarations together for a large program. It guarantees that all the source files will be supplied with the same definitions and variable declarations, and thus eliminates a particularly nasty kind of bug. Naturally, when an included file is changed, all files that depend on it must be recompiled.

4.11.2 Macro Substitution

A definition has the form

```
#define name replacement text
```

It calls for a macro substitution of the simplest kind - subsequent occurrences of the token *name* will be replaced by the *replacement text*. The name in a `#define` has the same form as a variable name; the replacement text is arbitrary. Normally the replacement text is the rest of the line, but a long definition may be continued onto several lines by placing a \ at the end of each line to be continued. The scope of a name defined with `#define` is from its point of definition to the end of the source file being compiled. A definition may use previous definitions. Substitutions are made only for tokens, and do not take place within quoted strings. For example, if YES is a defined name, there would be no substitution in `printf("YES")` or in YESMAN.

Any name may be defined with any replacement text. For example

```
#define forever for (;;) /* infinite loop */
```

defines a new word, `forever`, for an infinite loop.

It is also possible to define macros with arguments, so the replacement text can be different for different calls of the macro. As an example, define a macro called `max`:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Although it looks like a function call, a use of `max` expands into in-line code. Each occurrence of a formal parameter (here `A` or `B`) will be replaced by the corresponding actual argument. Thus the line

```
x = max(p+q, r+s);
will be replaced by the line
```

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

So long as the arguments are treated consistently, this macro will serve for any data type; there is no need for different kinds of `max` for different data types, as there would be with functions.

If you examine the expansion of `max`, you will notice some pitfalls. The expressions are evaluated twice; this is bad if they involve side effects like increment operators or input and output. For instance

```
max(i++, j++) /* WRONG */
```

will increment the larger twice. Some care also has to be taken with parentheses to make sure the order of evaluation is preserved; consider what happens when the macro

```
#define square(x) x * x /* WRONG */
```

is invoked as `square(z+1)`.

Nonetheless, macros are valuable. One practical example comes from `<stdio.h>`, in which `getchar` and `putchar` are often defined as macros to avoid the run-time overhead of a function call per character processed. The functions in `<ctype.h>` are also usually implemented as macros.

Names may be undefined with `#undef`, usually to ensure that a routine is really a function, not a macro:

```
#undef getchar
```

```
int getchar(void) { ... }
```

Formal parameters are not replaced within quoted strings. If, however, a parameter name is preceded by a `#` in the replacement text, the combination will be expanded into a quoted string with the parameter replaced by the actual argument. This can be combined with string concatenation to make, for example, a debugging print macro:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

When this is invoked, as in

```
dprint(x/y)
```

the macro is expanded into

```
printf("x/y" " = &g\n", x/y);
```

and the strings are concatenated, so the effect is

```
printf("x/y = &g\n", x/y);
```

Within the actual argument, each `"` is replaced by `\"` and each `\` by `\\`, so the result is a legal string constant.

The preprocessor operator `##` provides a way to concatenate actual arguments during macro expansion. If a parameter in the replacement text is adjacent to a `##`, the parameter is replaced by the actual argument, the `##` and surrounding white space are removed, and the result is re-scanned. For example, the macro `paste` concatenates its two arguments:

```
#define paste(front, back) front ## back
```

so `paste(name, 1)` creates the token `name1`.

The rules for nested uses of `##` are arcane; further details may be found in [Appendix A](#).

Exercise 4-14. Define a macro `swap(t,x,y)` that interchanges two arguments of type `t`. (Block structure will help.)

4.11.3 Conditional Inclusion

It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing. This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

The `#if` line evaluates a constant integer expression (which may not include `sizeof`, casts, or `enum` constants). If the expression is non-zero, subsequent lines until an `#endif` or `#elif` or `#else` are included. (The preprocessor statement `#elif` is like `else-if`.) The expression `defined(name)` in a `#if` is 1 if the `name` has been defined, and 0 otherwise.

For example, to make sure that the contents of a file `hdr.h` are included only once, the contents of the file are surrounded with a conditional like this:

```
#if !defined(HDR)
#define HDR

/* contents of hdr.h go here */

#endif
```

The first inclusion of `hdr.h` defines the name `HDR`; subsequent inclusions will find the name `defined` and skip down to the `#endif`. A similar style can be used to avoid including files multiple times. If this style is used consistently, then each header can itself include any other headers on which it depends, without the user of the header having to deal with the interdependence.

This sequence tests the name `SYSTEM` to decide which version of a header to include:

```
#if SYSTEM == SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

The `#ifdef` and `#ifndef` lines are specialized forms that test whether a name is defined. The first example of `#if` above could have been written

```
#ifndef HDR
#define HDR

/* contents of hdr.h go here */

#endif
```

Chapter 5 - Pointers and Arrays

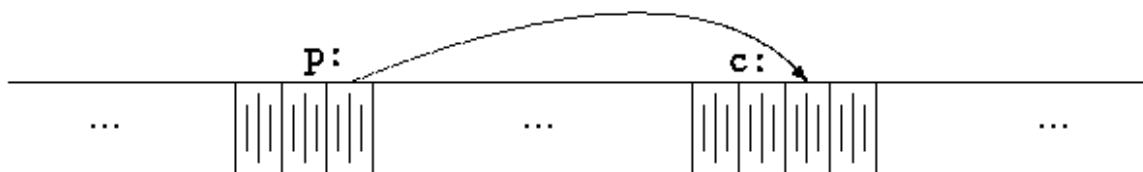
A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to void) replaces `char *` as the proper type for a generic pointer.

5.1 Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short integer`, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `p` is a pointer that points to it, we could represent the situation this way:



The unary operator `&` gives the address of an object, so the statement

```
p = &c;
```

assigns the address of `c` to the variable `p`, and `p` is said to ``point to'' `c`. The `&` operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

The unary operator `*` is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that `x` and `y` are integers and `ip` is a pointer to `int`. This artificial sequence shows how to declare a pointer and how to use `&` and `*`:

```
int x = 1, y = 2, z[10];
int *ip;           /* ip is a pointer to int */

ip = &x;           /* ip now points to x */
y = *ip;           /* y is now 1 */
*ip = 0;           /* x is now 0 */
ip = &z[0];        /* ip now points to z[0] */
```

The declaration of `x`, `y`, and `z` are what we've seen all along. The declaration of the pointer `ip`,

```
int *ip;
```

is intended as a mnemonic; it says that the expression `*ip` is an `int`. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression `*dp` and `atof(s)` have values of `double`, and that the argument of `atof` is a pointer to `char`.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a ``pointer to `void`'' is used to hold any type of pointer but cannot be dereferenced itself. We'll come back to it in [Section 5.11.](#))

If `ip` points to the integer `x`, then `*ip` can occur in any context where `x` could, so

```
*ip = *ip + 10;  
increments *ip by 10.
```

The unary operators `*` and `&` bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever `ip` points at, adds 1, and assigns the result to `y`, while

```
*ip += 1
```

increments what `ip` points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment `ip` instead of what it points to, because unary operators like `*` and `++` associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if `iq` is another pointer to `int`,

```
iq = ip
```

copies the contents of `ip` into `iq`, thus making `iq` point to whatever `ip` pointed to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order arguments with a function called `swap`. It is not enough to write

```
swap(a, b);
```

where the `swap` function is defined as

```
void swap(int x, int y) /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Because of call by value, `swap` can't affect the arguments `a` and `b` in the routine that called it. The function above swaps *copies* of `a` and `b`.

The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:

```
swap(&a, &b);
```

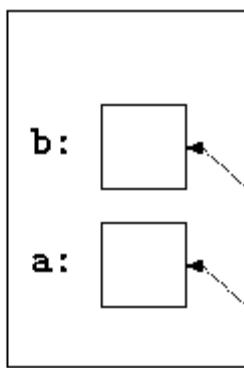
Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py) /* interchange *px and *py */
{
    int temp;

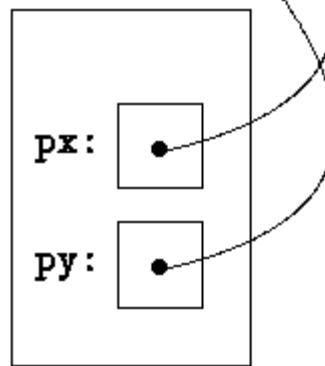
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pictorially:

in caller:



in swap:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for `EOF`, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf` as well; see [Section 7.4](#).

The following loop fills an array with integers by calls to `getint`:

```

int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
;

```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint` returns `EOF` for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```

#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch())) /* skip white space */
    ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c); /* it is not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c), c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}

```

Throughout `getint`, `*pn` is used as an ordinary `int` variable. We have also used `getch` and `ungetch` (described in [Section 4.3](#)) so the one extra character that must be read can be pushed back onto the input.

Exercise 5-1. As written, `getint` treats a `+` or `-` not followed by a digit as a valid representation of zero. Fix it to push such a character back on the input.

Exercise 5-2. Write `getfloat`, the floating-point analog of `getint`. What type does `getfloat` return as its function value?

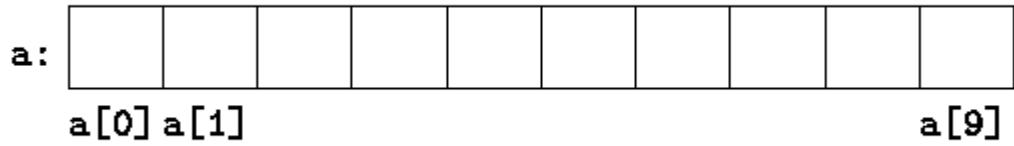
5.3 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The declaration

```
int a[10];
```

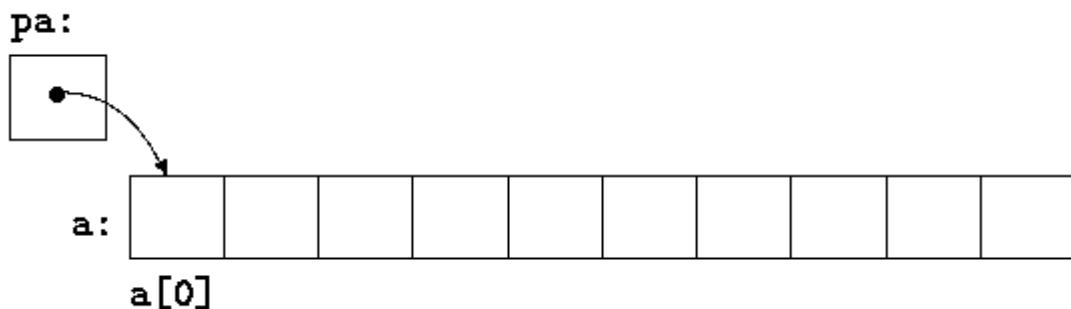
defines an array of size 10, that is, a block of 10 consecutive objects named `a[0]`, `a[1]`, ..., `a[9]`.



The notation `a[i]` refers to the i -th element of the array. If `pa` is a pointer to an integer, declared as

int *pa;
then the assignment

`pa = &a[0];`
sets `pa` to point to element zero of `a`; that is, `pa` contains the address of `a[0]`.

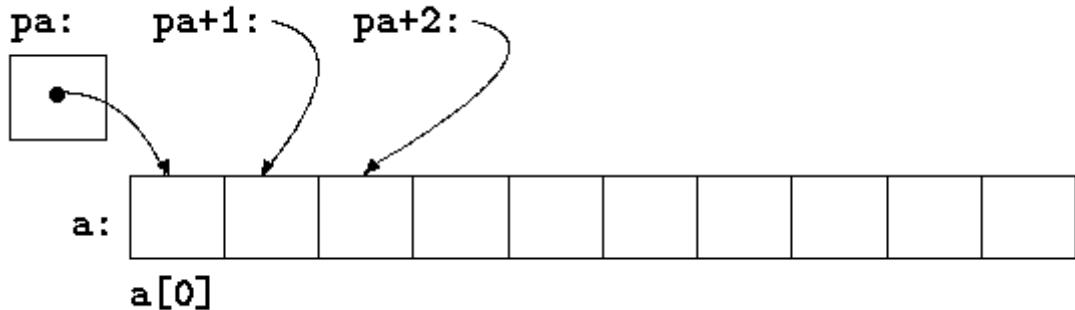


Now the assignment

`x = *pa;`
will copy the contents of `a[0]` into `x`.

If pa points to a particular element of an array, then by definition $\text{pa}+1$ points to the next element, $\text{pa}+i$ points i elements after pa , and $\text{pa}-i$ points i elements before. Thus, if pa points to $a[0]$,

$*(pa+i)$ refers to the contents of $a[1]$, $pa+i$ is the address of $a[i]$, and $*(pa+i)$ is the contents of $a[i]$.



These remarks are true regardless of the type or size of the variables in the array `a`. The meaning of ``adding 1 to a pointer," and by extension, all pointer arithmetic, is that `pa+1` points to the next object, and `pa+i` points to the `i`-th object beyond `pa`.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0];
```

`pa` and `a` have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as

```
pa = a;
```

Rather more surprising, at first sight, is the fact that a reference to `a[i]` can also be written as `*(a+i)`. In evaluating `a[i]`, C converts it to `*(a+i)` immediately; the two forms are equivalent. Applying the operator `&` to both parts of this equivalence, it follows that `&a[i]` and `a+i` are also identical: `a+i` is the address of the `i`-th element beyond `a`. As the other side of this coin, if `pa` is a pointer, expressions might use it with a subscript; `pa[i]` is identical to `*(pa+i)`. In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; constructions like `a=pa` and `a++` are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of `strlen`, which computes the length of a string.

```
/* strlen:  return length of string s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0', s++)
        n++;
    return n;
}
```

Since `s` is a pointer, incrementing it is perfectly legal; `s++` has no effect on the character string in the function that called `strlen`, but merely increments `strlen`'s private copy of the pointer. That means that calls like

```
strlen("hello, world"); /* string constant */
strlen(array);           /* char array[100]; */
strlen(ptr);             /* char *ptr; */
```

all work.

As formal parameters in a function definition,

```
char s[];
```

and

```
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the variable is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if `a` is an array,

```
f(&a[2])
```

and

```
f(a+2)
```

both pass to the function `f` the address of the subarray that starts at `a[2]`. Within `f`, the parameter declaration can read

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

So as far as `f` is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

If one is sure that the elements exist, it is also possible to index backwards in an array; `p[-1]`, `p[-2]`, and so on are syntactically legal, and refer to the elements that immediately precede `p[0]`. Of course, it is illegal to refer to objects that are not within the array bounds.

5.4 Address Arithmetic

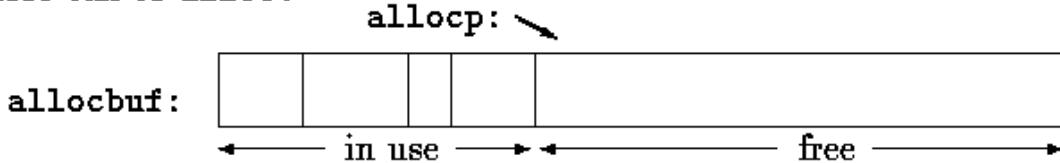
If `p` is a pointer to some element of an array, then `p++` increments `p` to point to the next element, and `p+=i` increments it to point `i` elements beyond where it currently does. These and similar constructions are the simplest forms of pointer or address arithmetic.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language. Let us illustrate by writing a rudimentary storage allocator. There are two routines. The first, `alloc(n)`, returns a pointer to `n` consecutive character positions, which can be used by the caller of `alloc` for storing characters. The second, `afree(p)`, releases the storage thus acquired so it can be reused later. The routines are ``rudimentary'' because the calls to `afree` must be made in the opposite order to the calls made on `alloc`. That is, the storage managed by `alloc` and `afree` is a stack, or last-in, first-out. The standard library provides analogous functions called `malloc` and `free` that have no such restrictions; in [Section 8.7](#) we will show how they can be implemented.

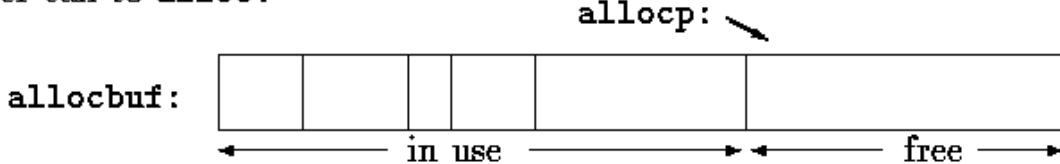
The easiest implementation is to have `alloc` hand out pieces of a large character array that we will call `allocbuf`. This array is private to `alloc` and `afree`. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared `static` in the source file containing `alloc` and `afree`, and thus be invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained by calling `malloc` or by asking the operating system for a pointer to some unnamed block of storage.

The other information needed is how much of `allocbuf` has been used. We use a pointer, called `allocp`, that points to the next free element. When `alloc` is asked for `n` characters, it checks to see if there is enough room left in `allocbuf`. If so, `alloc` returns the current value of `allocp` (i.e., the beginning of the free block), then increments it by `n` to point to the next free area. If there is no room, `alloc` returns zero. `afree(p)` merely sets `allocp` to `p` if `p` is inside `allocbuf`.

before call to alloc:



after call to alloc:



```
#define ALLOCSIZE 10000 /* size of available space */

static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf; /* next free position */

char *alloc(int n) /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
        allocp += n;
        return allocp - n; /* old p */
    } else /* not enough room */
        return 0;
}

void afree(char *p) /* free storage pointed to by p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the address of previously defined data of appropriate type. The declaration

```
static char *allocp = allocbuf;
```

defines `allocp` to be a character pointer and initializes it to point to the beginning of `allocbuf`, which is the next free position when the program starts. This could also have been written

```
static char *allocp = &allocbuf[0];
```

since the array name *is* the address of the zeroth element.

The test

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

checks if there's enough room to satisfy a request for `n` characters. If there is, the new value of `allocp` would be at most one beyond the end of `allocbuf`. If the request can be satisfied, `alloc` returns a pointer to the beginning of a block of characters (notice the declaration of the function itself). If not, `alloc` must return some signal that there is no space left. C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event, in this case no space.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero. The symbolic

constant `NULL` is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. `NULL` is defined in `<stdio.h>`. We will use `NULL` henceforth.

Tests like

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
```

and

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

show several important facets of pointer arithmetic. First, pointers may be compared under certain circumstances. If `p` and `q` point to members of the same array, then relations like `==`, `!=`, `<`, `>=`, etc., work properly. For example,

```
p < q
```

is true if `p` points to an earlier element of the array than `q` does. Any pointer can be meaningfully compared for equality or inequality with zero. But the behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array. (There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the address of the `n`-th object beyond the one `p` currently points to. This is true regardless of the kind of object `p` points to; `n` is scaled according to the size of the objects `p` points to, which is determined by the declaration of `p`. If an `int` is four bytes, for example, the `int` will be scaled by four.

Pointer subtraction is also valid: if `p` and `q` point to elements of the same array, and `p < q`, then `q-p+1` is the number of elements from `p` to `q` inclusive. This fact can be used to write yet another version of `strlen`:

```
/* strlen: return length of string s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

In its declaration, `p` is initialized to `s`, that is, to point to the first character of the string. In the `while` loop, each character in turn is examined until the '`\0`' at the end is seen. Because `p` points to characters, `p++` advances `p` to the next character each time, and `p-s` gives the number of characters advanced over, that is, the string length. (The number of characters in the string could be too large to store in an `int`. The header `<stddef.h>` defines a type `ptrdiff_t` that is large enough to hold the signed difference of two pointer values. If we were being cautious, however, we would use `size_t` for the return value of `strlen`, to match the standard library version. `size_t` is the unsigned integer type returned by the `sizeof` operator.

Pointer arithmetic is consistent: if we had been dealing with `floats`, which occupy more storage than `chars`, and if `p` were a pointer to `float`, `p++` would advance to the next `float`. Thus we could write another version of `alloc` that maintains `floats` instead of `chars`, merely by changing `char` to `float` throughout `alloc` and `afree`. All the pointer manipulations automatically take into account the size of the objects pointed to.

The valid pointer operations are assignment of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero. All other pointer arithmetic is illegal. It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add `float` or `double` to them, or even, except for `void *`, to assign a pointer of one type to a pointer of another type without a cast.

5.5 Character Pointers and Functions

A *string constant*, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminated with the null character '`\0`' so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; `printf` receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If `pmessage` is declared as

```
char *pmessage;
```

then the statement

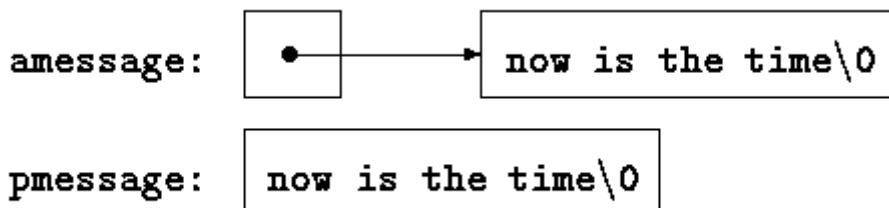
```
pmessage = "now is the time";
```

assigns to `pmessage` a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
```

`amessage` is an array, just big enough to hold the sequence of characters and '`\0`' that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage. On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is `strcpy(s, t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the characters, we need a loop. The array version first:

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

For contrast, here is a version of `strcpy` with pointers:

```
/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the '`\0`' that terminates `t` has been copied into `s`.

In practice, `strcpy` would not be written as we showed it above. Experienced C programmers would prefer

```
/* strcpy: copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

This moves the increment of `s` and `t` into the test part of the loop. The value of `*t++` is the character that `t` pointed to before `t` was incremented; the postfix `++` doesn't change `t` until after this character has been fetched. In the same way, the character is stored into the old `s` position before `s` is incremented. This character is also the value that is compared against '`\0`' to control the loop. The net effect is that characters are copied from `t` to `s`, up and including the terminating '`\0`'.

As the final abbreviation, observe that a comparison against '`\0`' is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```
/* strcpy: copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The `strcpy` in the standard library (`<string.h>`) returns the target string as its function value.

The second routine that we will examine is `strcmp(s,t)`, which compares the character strings `s` and `t`, and returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
```

```

int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}

```

The pointer version of `strcmp`:

```

/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}

```

Since `++` and `--` are either prefix or postfix operators, other combinations of `*` and `++` and `--` occur, although less frequently. For example,

`*--p`

decrements `p` before fetching the character that `p` points to. In fact, the pair of expressions

```

*p++ = val; /* push val onto stack */
val = *--p; /* pop top of stack into val */

```

are the standard idiom for pushing and popping a stack; see [Section 4.3](#).

The header `<string.h>` contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

Exercise 5-3. Write a pointer version of the function `strcat` that we showed in [Chapter 2](#): `strcat(s,t)` copies the string `t` to the end of `s`.

Exercise 5-4. Write the function `strend(s,t)`, which returns 1 if the string `t` occurs at the end of the string `s`, and zero otherwise.

Exercise 5-5. Write versions of the library functions `strncpy`, `strncat`, and `strcmp`, which operate on at most the first `n` characters of their argument strings. For example, `strncpy(s,t,n)` copies at most `n` characters of `t` to `s`. Full descriptions are in [Appendix B](#).

Exercise 5-6. Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include `getline` ([Chapters 1 and 4](#)), `atoi`, `itoa`, and their variants ([Chapters 2, 3, and 4](#)), `reverse` ([Chapter 3](#)), and `strindex` and `getop` ([Chapter 4](#)).

5.6 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can. Let us illustrate by writing a program that will sort a set of text lines into alphabetic order, a stripped-down version of the UNIX program `sort`.

In [Chapter 3](#), we presented a Shell sort function that would sort an array of integers, and in [Chapter 4](#) we improved on it with a quicksort. The same algorithms will work, except that now we have to deal with lines of text, which are of different lengths, and which, unlike integers, can't be compared or moved in a single operation. We need a data representation that will cope efficiently and conveniently with variable-length text lines.

This is where the array of pointers enters. If the lines to be sorted are stored end-to-end in one long character array, then each line can be accessed by a pointer to its first character. The

pointers themselves can be stored in an array. Two lines can be compared by passing their pointers to `strcmp`. When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.



This eliminates the twin problems of complicated storage management and high overhead that would go with moving the lines themselves.

The sorting process has three steps:

*read all the lines of input
sort them
print them in order*

As usual, it's best to divide the program into functions that match this natural division, with the main routine controlling the other functions. Let us defer the sorting step for a moment, and concentrate on the data structure and the input and output.

The input routine has to collect and save the characters of each line, and build an array of pointers to the lines. It will also have to count the number of input lines, since that information is needed for sorting and printing. Since the input function can only cope with a finite number of input lines, it can return some illegal count like `-1` if too much input is presented.

The output routine only has to print the lines in the order in which they appear in the array of pointers.

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

#define MAXLINES 5000      /* max #lines to be sorted */

char *lineptr[MAXLINES]; /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* sort input lines */
main()
{
    int nlines;      /* number of input lines read */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}

#define MAXLEN 1000 /* max length of any input line */
int getline(char *, int);
```

```

char *alloc(int);

/* readlines: read input lines */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || p = alloc(len) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}

```

The function `getline` is from [Section 1.9](#).

The main new thing is the declaration for `lineptr`:

```
char *lineptr[MAXLINES]
```

says that `lineptr` is an array of `MAXLINES` elements, each element of which is a pointer to a `char`. That is, `lineptr[i]` is a character pointer, and `*lineptr[i]` is the character it points to, the first character of the `i`-th saved text line.

Since `lineptr` is itself the name of an array, it can be treated as a pointer in the same manner as in our earlier examples, and `writelines` can be written instead as

```

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}

```

Initially, `*lineptr` points to the first line; each element advances it to the next line pointer while `nlines` is counted down.

With input and output under control, we can proceed to sorting. The quicksort from [Chapter 4](#) needs minor changes: the declarations have to be modified, and the comparison operation must be done by calling `strcmp`. The algorithm remains the same, which gives us some confidence that it will still work.

```

/* qsort: sort v[left]...v[right] into increasing order */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right) /* do nothing if array contains */
        return;           /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;

```

```

        for (i = left+1; i <= right; i++)
            if (strcmp(v[i], v[left]) < 0)
                swap(v, ++last, i);
        swap(v, left, last);
        qsort(v, left, last-1);
        qsort(v, last+1, right);
    }
}

```

Similarly, the swap routine needs only trivial changes:

```

/* swap: interchange v[i] and v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

```

Since any individual element of `v` (alias `lineptr`) is a character pointer, `temp` must be also, so one can be copied to the other.

Exercise 5-7. Rewrite `readlines` to store lines in an array supplied by `main`, rather than calling `alloc` to maintain storage. How much faster is the program?

5.7 Multi-dimensional Arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties.

Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year. Let us define two functions to do the conversions: `day_of_year` converts the month and day into the day of the year, and `month_day` converts the day of the year into the month and day. Since this latter function computes two values, the month and day arguments will be pointers:

```

month_day(1988, 60, &m, &d)
sets m to 2 and d to 29 (February 29th).

```

These functions both need the same information, a table of the number of days in each month ("thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easier to separate them into two rows of a two-dimensional array than to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```

static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int i, leap;
    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day: set month, day from day of year */
void month_day(int year, int yearday, int *pmonth, int *pday)
{
    int i, leap;

```

```

leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
for (i = 1; yearday > daytab[leap][i]; i++)
    yearday -= daytab[leap][i];
*pmonth = i;
*pday = yearday;
}

```

Recall that the arithmetic value of a logical expression, such as the one for `leap`, is either zero (false) or one (true), so it can be used as a subscript of the array `daytab`.

The array `daytab` has to be external to both `day_of_year` and `month_day`, so they can both use it. We made it `char` to illustrate a legitimate use of `char` for storing small non-character integers.

`daytab` is the first two-dimensional array we have dealt with. In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

```
daytab[i][j] /* [row][col] */
```

rather than

```
daytab[i, j] /* WRONG */
```

Other than this notational distinction, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializers in braces; each row of a two-dimensional array is initialized by a corresponding sub-list. We started the array `daytab` with a column of zero so that month numbers can run from the natural 1 to 12 instead of 0 to 11. Since space is not at a premium here, this is clearer than adjusting the indices.

If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 `ints`. In this particular case, it is a pointer to objects that are arrays of 13 `ints`. Thus if the array `daytab` is to be passed to a function `f`, the declaration of `f` would be:

```
f(int daytab[2][13]) { ... }
```

It could also be

```
f(int daytab[][13]) { ... }
```

since the number of rows is irrelevant, or it could be

```
f(int (*daytab)[13]) { ... }
```

which says that the parameter is a pointer to an array of 13 integers. The parentheses are necessary since brackets `[]` have higher precedence than `*`. Without parentheses, the declaration

```
int *daytab[13]
```

is an array of 13 pointers to integers. More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

[Section 5.12](#) has a further discussion of complicated declarations.

Exercise 5-8. There is no error checking in `day_of_year` or `month_day`. Remedy this defect.

5.8 Initialization of Pointer Arrays

Consider the problem of writing a function `month_name(n)`, which returns a pointer to a character string containing the name of the n -th month. This is an ideal application for an internal static array. `month_name` contains a private array of character strings, and returns a pointer to the proper one when called. This section shows how that array of names is initialized.

The syntax is similar to previous initializations:

```
/* month_name: return name of n-th month */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

The declaration of `name`, which is an array of character pointers, is the same as `lineptr` in the sorting example. The initializer is a list of character strings; each is assigned to the corresponding position in the array. The characters of the i -th string are placed somewhere, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler counts the initializers and fills in the correct number.

5.9 Pointers vs. Multi-dimensional Arrays

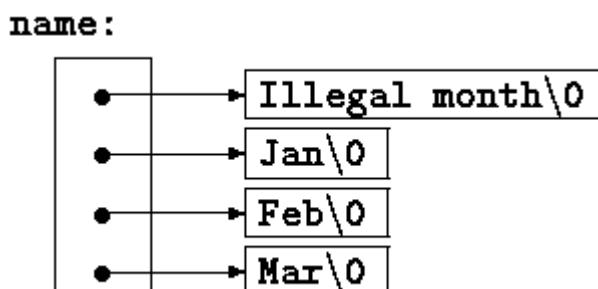
Newcomers to C are sometimes confused about the difference between a two-dimensional array and an array of pointers, such as `name` in the example above. Given the definitions

```
int a[10][20];
int *b[10];
```

then `a[3][4]` and `b[3][4]` are both syntactically legal references to a single `int`. But `a` is a true two-dimensional array: 200 `int`-sized locations have been set aside, and the conventional rectangular subscript calculation $20 * \text{row} + \text{col}$ is used to find the element `a[row,col]`. For `b`, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code. Assuming that each element of `b` does point to a twenty-element array, then there will be 200 `int`s set aside, plus ten cells for the pointers. The important advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of `b` need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all.

Although we have phrased this discussion in terms of integers, by far the most frequent use of arrays of pointers is to store character strings of diverse lengths, as in the function `month_name`. Compare the declaration and picture for an array of pointers:

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```



with those for a two-dimensional array:

```
char fname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

<u>fname:</u>			
Illegal month\0	Jan\0	Feb\0	Mar\0
0	15	30	45

Exercise 5-9. Rewrite the routines `day_of_year` and `month_day` with pointers instead of indexing.

5.10 Command-line Arguments

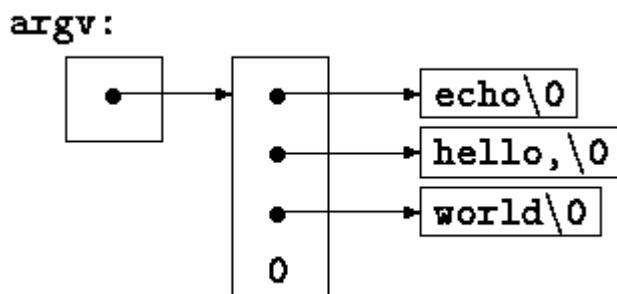
In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

The simplest illustration is the program `echo`, which echoes its command-line arguments on a single line, separated by blanks. That is, the command

```
echo hello, world  
prints the output
```

```
hello, world
```

By convention, `argv[0]` is the name by which the program was invoked, so `argc` is at least 1. If `argc` is 1, there are no command-line arguments after the program name. In the example above, `argc` is 3, and `argv[0]`, `argv[1]`, and `argv[2]` are "echo", "hello,", and "world" respectively. The first optional argument is `argv[1]` and the last is `argv[argc-1]`; additionally, the standard requires that `argv[argc]` be a null pointer.



The first version of `echo` treats `argv` as an array of character pointers:

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

```
}
```

Since `argv` is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variant is based on incrementing `argv`, which is a pointer to pointer to `char`, while `argc` is counted down:

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since `argv` is a pointer to the beginning of the array of argument strings, incrementing it by 1 (`*++argv`) makes it point at the original `argv[1]` instead of `argv[0]`. Each successive increment moves it along to the next argument; `*argv` is then the pointer to that argument. At the same time, `argc` is decremented; when it becomes zero, there are no arguments left to print.

Alternatively, we could write the `printf` statement as

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

This shows that the format argument of `printf` can be an expression too.

As a second example, let us make some enhancements to the pattern-finding program from [Section 4.1](#). If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX program `grep`, let us enhance the program so the pattern to be matched is specified by the first argument on the command line.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

The standard library function `strstr(s,t)` returns a pointer to the first occurrence of the string `t` in the string `s`, or `NULL` if there is none. It is declared in `<string.h>`.

The model can now be elaborated to illustrate further pointer constructions. Suppose we want to allow two optional arguments. One says ``print all the lines *except* those that match the pattern;'' the second says ``precede each printed line by its line number.''

A common convention for C programs on UNIX systems is that an argument that begins with a minus sign introduces an optional flag or parameter. If we choose `-x` (for ``except'') to signal the inversion, and `-n` (``number'') to request line numbering, then the command

```
find -x -n pattern
```

will print each line that doesn't match the pattern, preceded by its line number.

Optional arguments should be permitted in any order, and the rest of the program should be independent of the number of arguments that we present. Furthermore, it is convenient for users if option arguments can be combined, as in

```
find -nx pattern
```

Here is the program:

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find: print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
                case 'x':
                    except = 1;
                    break;
                case 'n':
                    number = 1;
                    break;
                default:
                    printf("find: illegal option %c\n", c);
                    argc = 0;
                    found = -1;
                    break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

`argc` is decremented and `argv` is incremented before each optional argument. At the end of the loop, if there are no errors, `argc` tells how many arguments remain unprocessed and `argv` points to the first of these. Thus `argc` should be 1 and `*argv` should point at the pattern. Notice that `*++argv` is a pointer to an argument string, so `(*++argv)[0]` is its first character. (An alternate valid form would be `*++*argv`.) Because `[]` binds tighter than `*` and `++`, the parentheses are necessary; without them the expression would be taken as `*++(argv[0])`. In

fact, that is what we have used in the inner loop, where the task is to walk along a specific argument string. In the inner loop, the expression `*++argv[0]` increments the pointer `argv[0]`!

It is rare that one uses pointer expressions more complicated than these; in such cases, breaking them into two or three steps will be more intuitive.

Exercise 5-10. Write the program `expr`, which evaluates a reverse Polish expression from the command line, where each operator or operand is a separate argument. For example,

```
expr 2 3 4 + *
evaluates 2 * (3+4).
```

Exercise 5-11. Modify the program `entab` and `datab` (written as exercises in [Chapter 1](#)) to accept a list of tab stops as arguments. Use the default tab settings if there are no arguments.

Exercise 5-12. Extend `entab` and `datab` to accept the shorthand

```
entab -m +n
```

to mean tab stops every n columns, starting at column m . Choose convenient (for the user) default behavior.

Exercise 5-13. Write the program `tail`, which prints the last n lines of its input. By default, n is set to 10, let us say, but it can be changed by an optional argument so that

```
tail -n
```

prints the last n lines. The program should behave rationally no matter how unreasonable the input or the value of n . Write the program so it makes the best use of available storage; lines should be stored as in the sorting program of [Section 5.6](#), not in a two-dimensional array of fixed size.

5.11 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on. We will illustrate this by modifying the sorting procedure written earlier in this chapter so that if the optional argument `-n` is given, it will sort the input lines numerically instead of lexicographically.

A sort often consists of three parts - a comparison that determines the ordering of any pair of objects, an exchange that reverses their order, and a sorting algorithm that makes comparisons and exchanges until the objects are in order. The sorting algorithm is independent of the comparison and exchange operations, so by passing different comparison and exchange functions to it, we can arrange to sort by different criteria. This is the approach taken in our new sort.

Lexicographic comparison of two lines is done by `strcmp`, as before; we will also need a routine `numcmp` that compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does. These functions are declared ahead of `main` and a pointer to the appropriate one is passed to `qsort`. We have skimped on error processing for arguments, so as to concentrate on the main issues.

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

#define MAXLINES 5000      /* max #lines to be sorted */
char *lineptr[MAXLINES]; /* pointers to text lines */
```

```

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* sort input lines */
main(int argc, char *argv[])
{
    int nlines;          /* number of input lines read */
    int numeric = 0;     /* 1 if numeric sort */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void**) lineptr, 0, nlines-1,
               (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}

```

In the call to `qsort`, `strcmp` and `numcmp` are addresses of functions. Since they are known to be functions, the `&` is not necessary, in the same way that it is not needed before an array name.

We have written `qsort` so it can process any data type, not just character strings. As indicated by the function prototype, `qsort` expects an array of pointers, two integers, and a function with two pointer arguments. The generic pointer type `void *` is used for the pointer arguments. Any pointer can be cast to `void *` and back again without loss of information, so we can call `qsort` by casting arguments to `void *`. The elaborate cast of the function argument casts the arguments of the comparison function. These will generally have no effect on actual representation, but assure the compiler that all is well.

```

/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *));
{
    int i, last;

    void swap(void *v[], int, int);

    if (left >= right)      /* do nothing if array contains */
        return;                /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}

```

The declarations should be studied with some care. The fourth parameter of `qsort` is

```
int (*comp)(void *, void *)
```

which says that `comp` is a pointer to a function that has two `void *` arguments and returns an `int`.

The use of `comp` in the line

```
if ((*comp)(v[i], v[left]) < 0)
```

is consistent with the declaration: `comp` is a pointer to a function, `*comp` is the function, and

```
( *comp)(v[i], v[left])
```

is the call to it. The parentheses are needed so the components are correctly associated; without them,

```
int *comp(void *, void *) /* WRONG */
```

says that `comp` is a function returning a pointer to an `int`, which is very different.

We have already shown `strcmp`, which compares two strings. Here is `numcmp`, which compares two strings on a leading numeric value, computed by calling `atof`:

```
#include <stdlib.h>

/* numcmp: compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

The `swap` function, which exchanges two pointers, is identical to what we presented earlier in the chapter, except that the declarations are changed to `void *`.

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

A variety of other options can be added to the sorting program; some make challenging exercises.

Exercise 5-14. Modify the sort program to handle a `-r` flag, which indicates sorting in reverse (decreasing) order. Be sure that `-r` works with `-n`.

Exercise 5-15. Add the option `-f` to fold upper and lower case together, so that case distinctions are not made during sorting; for example, `a` and `A` compare equal.

Exercise 5-16. Add the `-d` ("directory order") option, which makes comparisons only on letters, numbers and blanks. Make sure it works in conjunction with `-f`.

Exercise 5-17. Add a field-searching capability, so sorting may be done on fields within lines, each field sorted according to an independent set of options. (The index for this book was sorted with `-df` for the index category and `-n` for the page numbers.)

5.12 Complicated Declarations

C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions. The syntax is an attempt to make the declaration and the use agree; it works well for simple cases, but it can be confusing for the harder ones, because declarations cannot be read left to right, and because parentheses are over-used. The difference between

```
int *f();           /* f: function returning pointer to int */
and
```

```
int (*pf)();      /* pf: pointer to function returning int */
```

illustrates the problem: * is a prefix operator and it has lower precedence than (), so parentheses are necessary to force the proper association.

Although truly complicated declarations rarely arise in practice, it is important to know how to understand them, and, if necessary, how to create them. One good way to synthesize declarations is in small steps with `typedef`, which is discussed in [Section 6.7](#). As an alternative, in this section we will present a pair of programs that convert from valid C to a word description and back again. The word description reads left to right.

The first, `dcl`, is the more complex. It converts a C declaration into a word description, as in these examples:

```
char **argv
    argv: pointer to char
int (*daytab)[13]
    daytab: pointer to array[13] of int
int *daytab[13]
    daytab: array[13] of pointer to int
void *comp()
    comp: function returning pointer to void
void (*comp]()
    comp: pointer to function returning void
char (*(*x())[])()
    x: function returning pointer to array[] of
        pointer to function returning char
char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
        pointer to array[5] of char
```

`dcl` is based on the grammar that specifies a declarator, which is spelled out precisely in [Appendix A, Section 8.5](#); this is a simplified form:

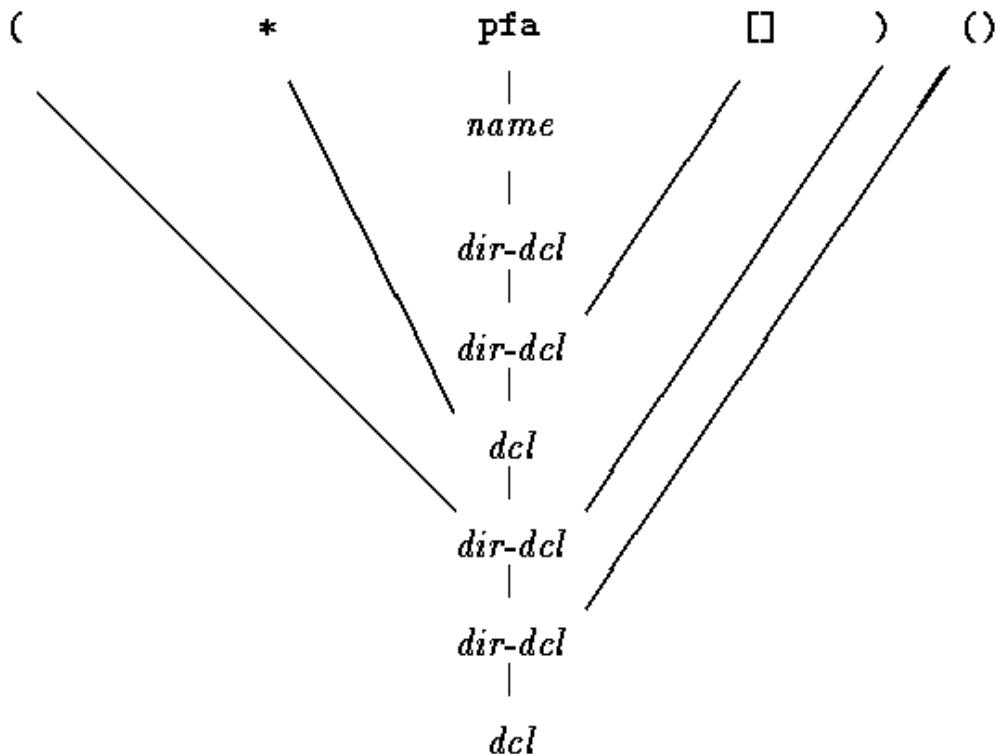
```
dcl:      optional '*'s direct-dcl
direct-dcl name
          (dcl)
          direct-dcl()
          direct-dcl[optional size]
```

In words, a `dcl` is a `direct-dcl`, perhaps preceded by '*'s. A `direct-dcl` is a name, or a parenthesized `dcl`, or a `direct-dcl` followed by parentheses, or a `direct-dcl` followed by brackets with an optional size.

This grammar can be used to parse functions. For instance, consider this declarator:

```
(*pfa[])()
```

`pfa` will be identified as a `name` and thus as a `direct-dcl`. Then `pfa[]` is also a `direct-dcl`. Then `*pfa[]` is recognized as a `dcl`, so `(*pfa[])` is a `direct-dcl`. Then `(*pfa[])()` is a `direct-dcl` and thus a `dcl`. We can also illustrate the parse with a tree like this (where `direct-dcl` has been abbreviated to `dir-dcl`):



The heart of the `dcl` program is a pair of functions, `dcl` and `dirdcl`, that parse a declaration according to this grammar. Because the grammar is recursively defined, the functions call each other recursively as they recognize pieces of a declaration; the program is called a recursive-descent parser.

```

/* dcl: parse a declarator */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*' ; ) /* count '*'s */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " pointer to");
}

/* dirdcl: parse a direct declarator */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {           /* ( dcl ) */
        dcl();
        if (tokentype != ')')
            printf("error: missing )\n");
    } else if (tokentype == NAME) /* variable name */
        strcpy(name, token);
    else
        printf("error: expected name or (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)

```

```

    if (type == PARENS)
        strcat(out, " function returning");
    else {
        strcat(out, " array");
        strcat(out, token);
        strcat(out, " of");
    }
}
}

```

Since the programs are intended to be illustrative, not bullet-proof, there are significant restrictions on dcl. It can only handle a simple data type like char or int. It does not handle argument types in functions, or qualifiers like const. Spurious blanks confuse it. It doesn't do much error recovery, so invalid declarations will also confuse it. These improvements are left as exercises.

Here are the global variables and the main routine:

```

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

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype;           /* type of last token */
char token[MAXTOKEN];   /* last token string */
char name[MAXTOKEN];    /* identifier name */
char datatype[MAXTOKEN]; /* data type = char, int, etc. */
char out[1000];

main() /* convert declaration to words */
{
    while (gettoken() != EOF) { /* 1st token on line */
        strcpy(datatype, token); /* is the datatype */
        out[0] = '\0';
        dcl();                 /* parse rest of line */
        if (tokentype != '\n')
            printf("syntax error\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}

```

The function gettoken skips blanks and tabs, then finds the next token in the input; a ``token'' is a name, a pair of parentheses, a pair of brackets perhaps including a number, or any other single character.

```

int gettoken(void) /* return next token */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    }
}

```

```

        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}

```

getch and ungetch are discussed in [Chapter 4](#).

Going in the other direction is easier, especially if we do not worry about generating redundant parentheses. The program `undcl` converts a word description like ```x` is a function returning a pointer to an array of pointers to functions returning `char`,'' which we will express as

```
x () * [] * () char
to
```

```
char (*(*x())[])()
```

The abbreviated input syntax lets us reuse the `gettken` function. `undcl` also uses the same external variables as `dcl` does.

```
/* undcl: convert word descriptions to declarations */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettken() != EOF) {
        strcpy(out, token);
        while ((type = gettken()) != '\n')
            if (type == PARENTS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
    }
    return 0;
}
```

Exercise 5-18. Make `dcl` recover from input errors.

Exercise 5-19. Modify `undcl` so that it does not add redundant parentheses to declarations.

Exercise 5-20. Expand `dcl` to handle declarations with function argument types, qualifiers like `const`, and so on.

Chapter 6 - Structures

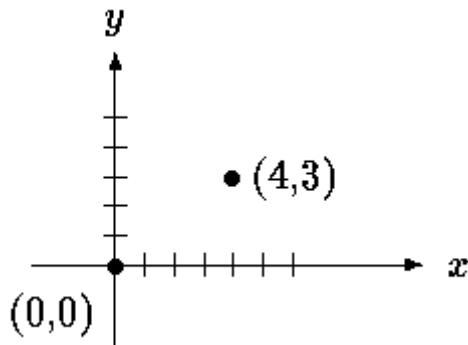
A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called ``records'' in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinate, a rectangle is a pair of points, and so on.

The main change made by the ANSI standard is to define structure assignment - structures may be copied and assigned to, passed to functions, and returned by functions. This has been supported by most compilers for many years, but the properties are now precisely defined. Automatic structures and arrays may now also be initialized.

6.1 Basics of Structures

Let us create a few structures suitable for graphics. The basic object is a point, which we will assume has an *x* coordinate and a *y* coordinate, both integers.



The two components can be placed in a structure declared like this:

```
struct point {
    int x;
    int y;
};
```

The keyword `struct` introduces a structure declaration, which is a list of declarations enclosed in braces. An optional name called a *structure tag* may follow the word `struct` (as with `point` here). The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces.

The variables named in a structure are called *members*. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict, since they can always be distinguished by context. Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

A `struct` declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct { ... } x, y, z;
```

is syntactically analogous to

```
int x, y, z;
```

in the sense that each statement declares `x`, `y` and `z` to be variables of the named type and causes space to be set aside for them.

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or shape of a structure. If the declaration is tagged, however, the tag can be used later in definitions of instances of the structure. For example, given the declaration of `point` above,

```
struct point pt;
```

defines a variable `pt` which is a structure of type `struct point`. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct maxpt = { 320, 200 };
```

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type.

A member of a particular structure is referred to in an expression by a construction of the form

structure-name.member

The structure member operator ``.'' connects the structure name and the member name. To print the coordinates of the point `pt`, for instance,

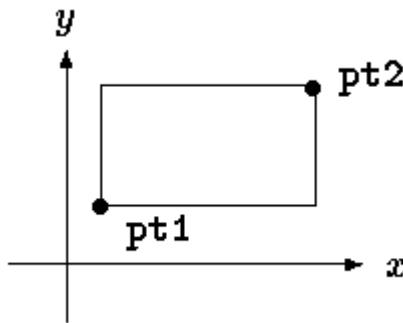
```
printf("%d,%d", pt.x, pt.y);
```

or to compute the distance from the origin (0,0) to `pt`,

```
double dist, sqrt(double);
```

```
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:



```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

The `rect` structure contains two `point` structures. If we declare `screen` as

```
struct rect screen;
```

then

```
screen.pt1.x
```

refers to the *x* coordinate of the `pt1` member of `screen`.

6.2 Structures and Functions

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with `&`, and accessing its members. Copy and assignment include passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

Let us investigate structures by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure, or pass a pointer to it. Each has its good points and bad points.

The first function, `makepoint`, will take two integers and return a point structure:

```
/* makepoint: make a point from x and y components */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship.

`makepoint` can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
/* addpoints: add two points */
struct addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Here both the arguments and the return value are structures. We incremented the components in `p1` rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others.

As another example, the function `ptinrect` tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
/* ptinrect: return 1 if p in r, 0 if not */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

This assumes that the rectangle is presented in a standard form where the pt1 coordinates are less than the pt2 coordinates. The following function returns a rectangle guaranteed to be in canonical form:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect: canonicalize coordinates of rectangle */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables. The declaration

```
struct point *pp;
```

says that pp is a pointer to a structure of type struct point. If pp points to a point structure, *pp is the structure, and (*pp).x and (*pp).y are the members. To use pp, we might write, for example,

```
struct point origin, *pp;

pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher than *. The expression *pp.x means *(pp.x), which is illegal here because x is not a pointer.

Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then

p->member-of-structure

refers to the particular member. So we could write instead

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Both . and -> associate from left to right, so if we have

```
struct rect r, *rp = &r;
```

then these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

The structure operators . and ->, together with () for function calls and [] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly. For example, given the declaration

```
struct {
    int len;
    char *str;
} *p;
```

then

```
++p->len
```

increments `len`, not `p`, because the implied parenthesization is `++(p->len)`. Parentheses can be used to alter binding: `(++p)->len` increments `p` before accessing `len`, and `(p++)->len` increments `p` afterward. (This last set of parentheses is unnecessary.)

In the same way, `*p->str` fetches whatever `str` points to; `*p->str++` increments `str` after accessing whatever it points to (just like `*s++`); `(*p->str)++` increments whatever `str` points to; and `*p++->str` increments `p` after accessing whatever `str` points to.

6.3 Arrays of Structures

Consider writing a program to count the occurrences of each C keyword. We need an array of character strings to hold the names, and an array of integers for the counts. One possibility is to use two parallel arrays, `keyword` and `keycount`, as in

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

But the very fact that the arrays are parallel suggests a different organization, an array of structures. Each keyword is a pair:

```
char *word;
int cout;
```

and there is an array of pairs. The structure declaration

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

declares a structure type `key`, defines an array `keytab` of structures of this type, and sets aside storage for them. Each element of the array is a structure. This could also be written

```
struct key {
    char *word;
    int count;
};

struct key keytab[NKEYS];
```

Since the structure `keytab` contains a constant set of names, it is easiest to make it an external variable and initialize it once and for all when it is defined. The structure initialization is analogous to earlier ones - the definition is followed by a list of initializers enclosed in braces:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
};
```

The initializers are listed in pairs corresponding to the structure members. It would be more precise to enclose the initializers for each "row" or structure in braces, as in

```
{ "auto", 0 },
```

```
{
    "break", 0 },
{
    "case", 0 },
...
```

but inner braces are not necessary when the initializers are simple variables or character strings, and when all are present. As usual, the number of entries in the array `keytab` will be computed if the initializers are present and the `[]` is left empty.

The keyword counting program begins with the definition of `keytab`. The main routine reads the input by repeatedly calling a function `getword` that fetches one word at a time. Each word is looked up in `keytab` with a version of the binary search function that we wrote in [Chapter 3](#). The list of keywords must be sorted in increasing order in the table.

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

#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* count C keywords */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                   keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch: find word in tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

We will show the function `getword` in a moment; for now it suffices to say that each call to `getword` finds a word, which is copied into the array named as its first argument.

The quantity `NKEYS` is the number of keywords in `keytab`. Although we could count this by hand, it's a lot easier and safer to do it by machine, especially if the list is subject to change. One possibility would be to terminate the list of initializers with a null pointer, then loop along `keytab` until the end is found.

But this is more than is needed, since the size of the array is completely determined at compile time. The size of the array is the size of one entry times the number of entries, so the number of entries is just

size of keytab / size of struct key

C provides a compile-time unary operator called `sizeof` that can be used to compute the size of any object. The expressions

`sizeof object`

and

`sizeof (type name)`

yield an integer equal to the size of the specified object or type in bytes. (Strictly, `sizeof` produces an unsigned integer value whose type, `size_t`, is defined in the header `<stddef.h>`.) An object can be a variable or array or structure. A type name can be the name of a basic type like `int` or `double`, or a derived type like a structure or a pointer.

In our case, the number of keywords is the size of the array divided by the size of one element. This computation is used in a `#define` statement to set the value of `NKEYS`:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Another way to write this is to divide the array size by the size of a specific element:

```
#define NKEYS (sizeof keytab / sizeof(keytab[0]))
```

This has the advantage that it does not need to be changed if the type changes.

A `sizeof` can not be used in a `#if` line, because the preprocessor does not parse type names. But the expression in the `#define` is not evaluated by the preprocessor, so the code here is legal.

Now for the function `getword`. We have written a more general `getword` than is necessary for this program, but it is not complicated. `getword` fetches the next "word" from the input, where a word is either a string of letters and digits beginning with a letter, or a single non-white space character. The function value is the first character of the word, or `EOF` for end of file, or the character itself if it is not alphabetic.

```
/* getword: get next word or character from input */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```

`getword` uses the `getch` and `ungetch` that we wrote in [Chapter 4](#). When the collection of an alphanumeric token stops, `getword` has gone one character too far. The call to `ungetch` pushes that character back on the input for the next call. `getword` also uses `isspace` to skip whitespace, `isalpha` to identify letters, and `isalnum` to identify letters and digits; all are from the standard header `<ctype.h>`.

Exercise 6-1. Our version of `getword` does not properly handle underscores, string constants, comments, or preprocessor control lines. Write a better version.

6.4 Pointers to Structures

To illustrate some of the considerations involved with pointers to and arrays of structures, let us write the keyword-counting program again, this time using pointers instead of array indices.

The external declaration of `keytab` need not change, but `main` and `binsearch` do need modification.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* count C keywords; pointer version */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

There are several things worthy of note here. First, the declaration of `binsearch` must indicate that it returns a pointer to `struct key` instead of an integer; this is declared both in the

function prototype and in `binsearch`. If `binsearch` finds the word, it returns a pointer to it; if it fails, it returns `NULL`.

Second, the elements of `keytab` are now accessed by pointers. This requires significant changes in `binsearch`.

The initializers for `low` and `high` are now pointers to the beginning and just past the end of the table.

The computation of the middle element can no longer be simply

```
mid = (low+high) / 2      /* WRONG */
```

because the addition of pointers is illegal. Subtraction is legal, however, so `high-low` is the number of elements, and thus

```
mid = low + (high-low) / 2
```

sets `mid` to the element halfway between `low` and `high`.

The most important change is to adjust the algorithm to make sure that it does not generate an illegal pointer or attempt to access an element outside the array. The problem is that `&tab[-1]` and `&tab[n]` are both outside the limits of the array `tab`. The former is strictly illegal, and it is illegal to dereference the latter. The language definition does guarantee, however, that pointer arithmetic that involves the first element beyond the end of an array (that is, `&tab[n]`) will work correctly.

In `main` we wrote

```
for (p = keytab; p < keytab + NKEYS; p++)
```

If `p` is a pointer to a structure, arithmetic on `p` takes into account the size of the structure, so `p++` increments `p` by the correct amount to get the next element of the array of structures, and the test stops the loop at the right time.

Don't assume, however, that the size of a structure is the sum of the sizes of its members. Because of alignment requirements for different objects, there may be unnamed ``holes'' in a structure. Thus, for instance, if a `char` is one byte and an `int` four bytes, the structure

```
struct {
    char c;
    int i;
};
```

might well require eight bytes, not five. The `sizeof` operator returns the proper value.

Finally, an aside on program format: when a function returns a complicated type like a structure pointer, as in

```
struct key *binsearch(char *word, struct key *tab, int n)
```

the function name can be hard to see, and to find with a text editor. Accordingly an alternate style is sometimes used:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

This is a matter of personal taste; pick the form you like and hold to it.

6.5 Self-referential Structures

Suppose we want to handle the more general problem of counting the occurrences of *all* the words in some input. Since the list of words isn't known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each word as it arrives, to see if it's already been seen; the program would take too long. (More precisely, its running time is

likely to grow quadratically with the number of input words.) How can we organize the data to copy efficiently with a list or arbitrary words?

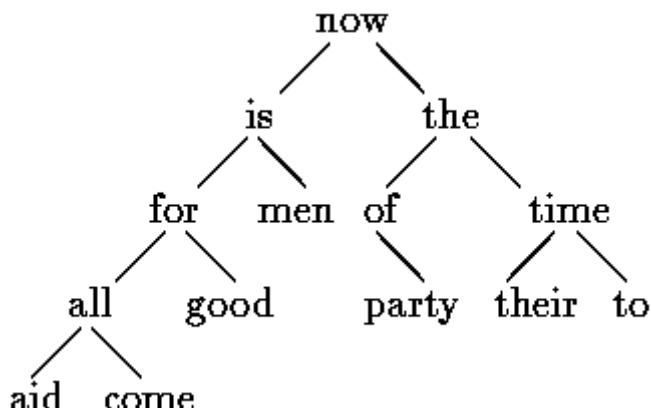
One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though - that also takes too long. Instead we will use a data structure called a *binary tree*.

The tree contains one ``node" per distinct word; each node contains

- A pointer to the text of the word,
- A count of the number of occurrences,
- A pointer to the left child node,
- A pointer to the right child node.

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words that are lexicographically less than the word at the node, and the right subtree contains only words that are greater. This is the tree for the sentence ``now is the time for all good men to come to the aid of their party", as built by inserting each word as it is encountered:



To find out whether a new word is already in the tree, start at the root and compare the new word to the word stored at that node. If they match, the question is answered affirmatively. If the new record is less than the tree word, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree, and in fact the empty slot is the proper place to add the new word. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is most conveniently represented as a structure with four components:

```

struct tnode {      /* the tree node: */
    char *word;        /* points to the text */
    int count;         /* number of occurrences */
    struct tnode *left; /* left child */
    struct tnode *right; /* right child */
};
  
```

This recursive declaration of a node might look chancy, but it's correct. It is illegal for a structure to contain an instance of itself, but

```
    struct tnode *left;
```

declares `left` to be a pointer to a `tnode`, not a `tnode` itself.

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```
struct t {
    ...
    struct s *p; /* p points to an s */
};

struct s {
    ...
    struct t *q; /* q points to a t */
};
```

The code for the whole program is surprisingly small, given a handful of supporting routines like `getword` that we have already written. The main routine reads words with `getword` and installs them in the tree with `addtree`.

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

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* word frequency count */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

The function `addtree` is recursive. A word is presented by `main` to the top level (the root) of the tree. At each stage, that word is compared to the word already stored at the node, and is percolated down to either the left or right subtree by a recursive call to `adtree`. Eventually, the word either matches something already in the tree (in which case the count is incremented), or a null pointer is encountered, indicating that a node must be created and added to the tree. If a new node is created, `adtree` returns a pointer to it, which is installed in the parent node.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree: add a node with w, at or below p */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) { /* a new word has arrived */
        p = talloc(); /* make a new node */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++; /* repeated word */
    else if (cond < 0) /* less than into left subtree */
        p->left = addtree(p->left, w);
```

```

    else          /* greater than into right subtree */
        p->right = addtree(p->right, w);
    return p;
}

```

Storage for the new node is fetched by a routine `talloc`, which returns a pointer to a free space suitable for holding a tree node, and the new word is copied into a hidden space by `strdup`. (We will discuss these routines in a moment.) The count is initialized, and the two children are made null. This part of the code is executed only at the leaves of the tree, when a new node is being added. We have (unwisely) omitted error checking on the values returned by `strdup` and `talloc`.

`treeprint` prints the tree in sorted order; at each node, it prints the left subtree (all the words less than this word), then the word itself, then the right subtree (all the words greater). If you feel shaky about how recursion works, simulate `treeprint` as it operates on the tree shown above.

```

/* treeprint: in-order print of tree p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}

```

A practical note: if the tree becomes ``unbalanced'' because the words don't arrive in random order, the running time of the program can grow too much. As a worst case, if the words are already in order, this program does an expensive simulation of linear search. There are generalizations of the binary tree that do not suffer from this worst-case behavior, but we will not describe them here.

Before leaving this example, it is also worth a brief digression on a problem related to storage allocators. Clearly it's desirable that there be only one storage allocator in a program, even though it allocates different kinds of objects. But if one allocator is to process requests for, say, pointers to `chars` and pointers to `struct tnodes`, two questions arise. First, how does it meet the requirement of most real machines that objects of certain types must satisfy alignment restrictions (for example, integers often must be located at even addresses)? Second, what declarations can cope with the fact that an allocator must necessarily return different kinds of pointers?

Alignment requirements can generally be satisfied easily, at the cost of some wasted space, by ensuring that the allocator always returns a pointer that meets *all* alignment restrictions. The `alloc` of [Chapter 5](#) does not guarantee any particular alignment, so we will use the standard library function `malloc`, which does. In [Chapter 8](#) we will show one way to implement `malloc`.

The question of the type declaration for a function like `malloc` is a vexing one for any language that takes its type-checking seriously. In C, the proper method is to declare that `malloc` returns a pointer to `void`, then explicitly coerce the pointer into the desired type with a cast. `malloc` and related routines are declared in the standard header `<stdlib.h>`. Thus `talloc` can be written as

```

#include <stdlib.h>

/* talloc: make a tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}

```

```
}
```

`strup` merely copies the string given by its argument into a safe place, obtained by a call on `malloc`:

```
char *strup(char *s) /* make a duplicate of s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1); /* +1 for '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

`malloc` returns `NULL` if no space is available; `strup` passes that value on, leaving error-handling to its caller.

Storage obtained by calling `malloc` may be freed for re-use by calling `free`; see [Chapters 8](#) and [7](#).

Exercise 6-2. Write a program that reads a C program and prints in alphabetical order each group of variable names that are identical in the first 6 characters, but different somewhere thereafter. Don't count words within strings and comments. Make 6 a parameter that can be set from the command line.

Exercise 6-3. Write a cross-referencer that prints a list of all words in a document, and for each word, a list of the line numbers on which it occurs. Remove noise words like ``the," ``and," and so on.

Exercise 6-4. Write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence. Precede each word by its count.

6.6 Table Lookup

In this section we will write the innards of a table-lookup package, to illustrate more aspects of structures. This code is typical of what might be found in the symbol table management routines of a macro processor or a compiler. For example, consider the `#define` statement. When a line like

```
#define IN 1
```

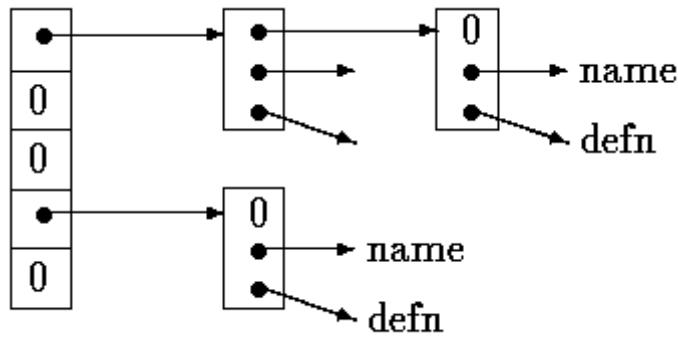
is encountered, the name `IN` and the replacement text `1` are stored in a table. Later, when the name `IN` appears in a statement like

```
state = IN;
```

it must be replaced by `1`.

There are two routines that manipulate the names and replacement texts. `install(s, t)` records the name `s` and the replacement text `t` in a table; `s` and `t` are just character strings. `lookup(s)` searches for `s` in the table, and returns a pointer to the place where it was found, or `NULL` if it wasn't there.

The algorithm is a hash-search - the incoming name is converted into a small non-negative integer, which is then used to index into an array of pointers. An array element points to the beginning of a linked list of blocks describing names that have that hash value. It is `NULL` if no names have hashed to that value.



A block in the list is a structure containing pointers to the name, the replacement text, and the next block in the list. A null next-pointer marks the end of the list.

```
struct nlist {           /* table entry: */
    struct nlist *next;   /* next entry in chain */
    char *name;          /* defined name */
    char *defn;          /* replacement text */
};
```

The pointer array is just

```
#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE]; /* pointer table */
```

The hashing function, which is used by both `lookup` and `install`, adds each character value in the string to a scrambled combination of the previous ones and returns the remainder modulo the array size. This is not the best possible hash function, but it is short and effective.

```
/* hash: form hash value for string s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Unsigned arithmetic ensures that the hash value is non-negative.

The hashing process produces a starting index in the array `hashtab`; if the string is to be found anywhere, it will be in the list of blocks beginning there. The search is performed by `lookup`. If `lookup` finds the entry already present, it returns a pointer to it; if not, it returns `NULL`.

```
/* lookup: look for s in hashtab */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np; /* found */
    return NULL; /* not found */
}
```

The `for` loop in `lookup` is the standard idiom for walking along a linked list:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

`install` uses `lookup` to determine whether the name being installed is already present; if so, the new definition will supersede the old one. Otherwise, a new entry is created. `install` returns `NULL` if for any reason there is no room for a new entry.

```

struct nlist *lookup(char *);
char * strdup(char *);

/* install: put (name, defn) in hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) { /* not found */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else /* already there */
        free((void *) np->defn); /*free previous defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}

```

Exercise 6-5. Write a function `undef` that will remove a name and definition from the table maintained by `lookup` and `install`.

Exercise 6-6. Implement a simple version of the `#define` processor (i.e., no arguments) suitable for use with C programs, based on the routines of this section. You may also find `getch` and `ungetch` helpful.

6.7 Typedef

C provides a facility called `typedef` for creating new data type names. For example, the declaration

```
typedef int Length;
```

makes the name `Length` a synonym for `int`. The type `Length` can be used in declarations, casts, etc., in exactly the same ways that the `int` type can be:

```
Length len, maxlen;
Length *lengths[];
```

Similarly, the declaration

```
typedef char *String;
```

makes `String` a synonym for `char *` or character pointer, which may then be used in declarations and casts:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Notice that the type being declared in a `typedef` appears in the position of a variable name, not right after the word `typedef`. Syntactically, `typedef` is like the storage classes `extern`, `static`, etc. We have used capitalized names for `typedefs`, to make them stand out.

As a more complicated example, we could make `typedefs` for the tree nodes shown earlier in this chapter:

```

typedef struct tnode *Treeptr;

typedef struct tnode { /* the tree node: */
    char *word;           /* points to the text */
    int count;            /* number of occurrences */
    struct tnode *left;   /* left child */

```

```

    struct tnode *right; /* right child */
} Treenode;

```

This creates two new type keywords called `Treenode` (a structure) and `Treeptr` (a pointer to the structure). Then the routine `talloc` could become

```

Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}

```

It must be emphasized that a `typedef` declaration does not create a new type in any sense; it merely adds a new name for some existing type. Nor are there any new semantics: variables declared this way have exactly the same properties as variables whose declarations are spelled out explicitly. In effect, `typedef` is like `#define`, except that since it is interpreted by the compiler, it can cope with textual substitutions that are beyond the capabilities of the preprocessor. For example,

```
typedef int (*PFI)(char *, char *);
```

creates the type `PFI`, for ``pointer to function (of two `char *` arguments) returning `int`," which can be used in contexts like

```
PFI strcmp, numcmp;
```

in the sort program of [Chapter 5](#).

Besides purely aesthetic issues, there are two main reasons for using `typedefs`. The first is to parameterize a program against portability problems. If `typedefs` are used for data types that may be machine-dependent, only the `typedefs` need change when the program is moved. One common situation is to use `typedef` names for various integer quantities, then make an appropriate set of choices of `short`, `int`, and `long` for each host machine. Types like `size_t` and `ptrdiff_t` from the standard library are examples.

The second purpose of `typedefs` is to provide better documentation for a program - a type called `Treeptr` may be easier to understand than one declared only as a pointer to a complicated structure.

6.8 Unions

A *union* is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program. They are analogous to variant records in pascal.

As an example such as might be found in a compiler symbol table manager, suppose that a constant may be an `int`, a `float`, or a character pointer. The value of a particular constant must be stored in a variable of the proper type, yet it is most convenient for table management if the value occupies the same amount of storage and is stored in the same place regardless of its type. This is the purpose of a union - a single variable that can legitimately hold any of one of several types. The syntax is based on structures:

```

union u_tag {
    int ival;
    float fval;
    char *sval;
} u;

```

The variable `u` will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any of these types may be assigned to `u` and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently

stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.

Syntactically, members of a union are accessed as

union-name.member

or

union-pointer->member

just as for structures. If the variable *utype* is used to keep track of the current type stored in *u*, then one might see code such as

```
if (utype == INT)
    printf("%d\n", u.ival);
if (utype == FLOAT)
    printf("%f\n", u.fval);
if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("bad type %d in utype\n", utype);
```

Unions may occur within structures and arrays, and vice versa. The notation for accessing a member of a union in a structure (or vice versa) is identical to that for nested structures. For example, in the structure array defined by

```
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

the member *ival* is referred to as

symtab[i].u.ival

and the first character of the string *sval* by either of

**symtab[i].u.sval*

symtab[i].u.sval[0]

In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the ``widest'' member, and the alignment is appropriate for all of the types in the union. The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address, and accessing a member.

A union may only be initialized with a value of the type of its first member; thus union *u* described above can only be initialized with an integer value.

The storage allocator in [Chapter 8](#) shows how a union can be used to force a variable to be aligned on a particular kind of storage boundary.

6.9 Bit-fields

When storage space is at a premium, it may be necessary to pack several objects into a single machine word; one common use is a set of single-bit flags in applications like compiler symbol tables. Externally-imposed data formats, such as interfaces to hardware devices, also often require the ability to get at pieces of a word.

Imagine a fragment of a compiler that manipulates a symbol table. Each identifier in a program has certain information associated with it, for example, whether or not it is a keyword, whether or not it is external and/or static, and so on. The most compact way to encode such information is a set of one-bit flags in a single `char` or `int`.

The usual way this is done is to define a set of ``masks'' corresponding to the relevant bit positions, as in

```
#define KEYWORD 01
#define EXTERNAL 02
#define STATIC 04
```

or

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

The numbers must be powers of two. Then accessing the bits becomes a matter of ``bit-fiddling'' with the shifting, masking, and complementing operators that were described in [Chapter 2](#).

Certain idioms appear frequently:

```
flags |= EXTERNAL | STATIC;
```

turns on the `EXTERNAL` and `STATIC` bits in `flags`, while

```
flags &= ~(EXTERNAL | STATIC);
```

turns them off, and

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

is true if both bits are off.

Although these idioms are readily mastered, as an alternative C offers the capability of defining and accessing fields within a word directly rather than by bitwise logical operators. A *bit-field*, or *field* for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a ``word.'' For example, the symbol table `#defines` above could be replaced by the definition of three fields:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
} flags;
```

This defines a variable table called `flags` that contains three 1-bit fields. The number following the colon represents the field width in bits. The fields are declared `unsigned int` to ensure that they are unsigned quantities.

Individual fields are referenced in the same way as other structure members: `flags.is_keyword`, `flags.is_extern`, etc. Fields behave like small integers, and may participate in arithmetic expressions just like other integers. Thus the previous examples may be written more naturally as

```
flags.is_extern = flags.is_static = 1;
```

to turn the bits on;

```
flags.is_extern = flags.is_static = 0;
```

to turn them off; and

```
if (flags.is_extern == 0 && flags.is_static == 0)
```

```
...
```

to test them.

Almost everything about fields is implementation-dependent. Whether a field may overlap a word boundary is implementation-defined. Fields need not be names; unnamed fields (a colon and width only) are used for padding. The special width 0 may be used to force alignment at the next word boundary.

Fields are assigned left to right on some machines and right to left on others. This means that although fields are useful for maintaining internally-defined data structures, the question of which end comes first has to be carefully considered when picking apart externally-defined data; programs that depend on such things are not portable. Fields may be declared only as `ints`; for portability, specify `signed` or `unsigned` explicitly. They are not arrays and they do not have addresses, so the `&` operator cannot be applied on them.

Chapter 7 - Input and Output

Input and output are not part of the C language itself, so we have not emphasized them in our presentation thus far. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs. We will concentrate on input and output.

The ANSI standard defines these library functions precisely, so that they can exist in compatible form on any system where C exists. Programs that confine their system interactions to facilities provided by the standard library can be moved from one system to another without change.

The properties of library functions are specified in more than a dozen headers; we have already seen several of these, including `<stdio.h>`, `<string.h>`, and `<cctype.h>`. We will not present the entire library here, since we are more interested in writing C programs that use it. The library is described in detail in [Appendix B](#).

7.1 Standard Input and Output

As we said in [Chapter 1](#), the library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.

The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with `getchar`:

```
int getchar(void)
```

`getchar` returns the next input character each time it is called, or `EOF` when it encounters end of file. The symbolic constant `EOF` is defined in `<stdio.h>`. The value is typically `-1`, but tests should be written in terms of `EOF` so as to be independent of the specific value.

In many environments, a file may be substituted for the keyboard by using the `<` convention for input redirection: if a program `prog` uses `getchar`, then the command line

```
prog <infile
```

causes `prog` to read characters from `infile` instead. The switching of the input is done in such a way that `prog` itself is oblivious to the change; in particular, the string `"<infile"` is not included in the command-line arguments in `argv`. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line

```
otherprog | prog
```

runs the two programs `otherprog` and `prog`, and pipes the standard output of `otherprog` into the standard input for `prog`.

The function

```
int putchar(int)
```

is used for output: `putchar(c)` puts the character `c` on the standard output, which is by default the screen. `putchar` returns the character written, or `EOF` if an error occurs. Again, output can usually be directed to a file with `>filename`: if `prog` uses `putchar`,

```
prog >outfile
```

will write the standard output to `outfile` instead. If pipes are supported,

```
prog | anotherprog
```

puts the standard output of `prog` into the standard input of `anotherprog`.

Output produced by `printf` also finds its way to the standard output. Calls to `putchar` and `printf` may be interleaved - output happens in the order in which the calls are made.

Each source file that refers to an input/output library function must contain the line

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

before the first reference. When the name is bracketed by `<` and `>` a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory `/usr/include`).

Many programs read only one input stream and write only one output stream; for such programs, input and output with `getchar`, `putchar`, and `printf` may be entirely adequate, and is certainly enough to get started. This is particularly true if redirection is used to connect the output of one program to the input of the next. For example, consider the program `lower`, which converts its input to lower case:

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

main() /* lower: convert input to lower case*/
{
    int c

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

The function `tolower` is defined in `<ctype.h>`; it converts an upper case letter to lower case, and returns other characters untouched. As we mentioned earlier, ``functions'' like `getchar` and `putchar` in `<stdio.h>` and `tolower` in `<ctype.h>` are often macros, thus avoiding the overhead of a function call per character. We will show how this is done in [Section 8.5](#). Regardless of how the `<ctype.h>` functions are implemented on a given machine, programs that use them are shielded from knowledge of the character set.

Exercise 7-1. Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in `argv[0]`.

7.2 Formatted Output - `printf`

The output function `printf` translates internal values to characters. We have used `printf` informally in previous chapters. The description here covers most typical uses but is not complete; for the full story, see [Appendix B](#).

```
int printf(char *format, arg1, arg2, ...);
```

`printf` converts, formats, and prints its arguments on the standard output under control of the `format`. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`. Each conversion specification begins with a `%` and ends with a conversion character. Between the `%` and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
- An `h` if the integer is to be printed as a `short`, or `l` (letter ell) if as a `long`.

Conversion characters are shown in Table 7.1. If the character after the `%` is not a conversion specification, the behavior is undefined.

Table 7.1 Basic `Printf` Conversions

Character	Argument type; Printed As
<code>d, i</code>	<code>int</code> ; decimal number
<code>o</code>	<code>int</code> ; unsigned octal number (without a leading zero)
<code>x, X</code>	<code>int</code> ; unsigned hexadecimal number (without a leading <code>0x</code> or <code>0X</code>), using <code>abcdef</code> or <code>ABCDEF</code> for 10, ..., 15.
<code>u</code>	<code>int</code> ; unsigned decimal number
<code>c</code>	<code>int</code> ; single character
<code>s</code>	<code>char *</code> ; print characters from the string until a ' <code>\0</code> ' or the number of characters given by the precision.
<code>f</code>	<code>double</code> ; <code>[-]m.ddddddd</code> , where the number of <code>d</code> 's is given by the precision (default 6).
<code>e, E</code>	<code>double</code> ; <code>[-]m.dddddddE+/-xx</code> or <code>[-]m.dddddddE+/-xx</code> , where the number of <code>d</code> 's is given by the precision (default 6).
<code>g, G</code>	<code>double</code> ; use <code>%e</code> or <code>%E</code> if the exponent is less than -4 or greater than or equal to the precision; otherwise use <code>%f</code> . Trailing zeros and a trailing decimal point are not printed.
<code>p</code>	<code>void *</code> ; pointer (implementation-dependent representation).
<code>%</code>	no argument is converted; print a <code>%</code>

A width or precision may be specified as `*`, in which case the value is computed by converting the next argument (which must be an `int`). For example, to print at most `max` characters from a string `s`,

```
printf("%.*s", max, s);
```

Most of the format conversions have been illustrated in earlier chapters. One exception is the precision as it relates to strings. The following table shows the effect of a variety of specifications in printing ``hello, world'' (12 characters). We have put colons around each field so you can see its extent.

<code>:%s:</code>	<code>:hello, world:</code>
<code>:%10s:</code>	<code>:hello, world:</code>
<code>:%.10s:</code>	<code>:hello, wor:</code>
<code>:%-10s:</code>	<code>:hello, world:</code>
<code>:%.15s:</code>	<code>:hello, world:</code>
<code>:%-15s:</code>	<code>:hello, world :</code>
<code>:%15.10s:</code>	<code>: hello, wor:</code>
<code>:%-15.10s:</code>	<code>:hello, wor :</code>

A warning: `printf` uses its first argument to decide how many arguments follow and what their type is. It will get confused, and you will get wrong answers, if there are not enough arguments or if they are the wrong type. You should also be aware of the difference between these two calls:

```
printf(s);          /* FAILS if s contains % */
printf("%s", s);   /* SAFE */
```

The function `sprintf` does the same conversions as `printf` does, but stores the output in a string:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

`sprintf` formats the arguments in `arg1`, `arg2`, etc., according to `format` as before, but places the result in `string` instead of the standard output; `string` must be big enough to receive the result.

Exercise 7-2. Write a program that will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines.

7.3 Variable-length Argument Lists

This section contains an implementation of a minimal version of `printf`, to show how to write a function that processes a variable-length argument list in a portable way. Since we are mainly interested in the argument processing, `minprintf` will process the format string and arguments but will call the real `printf` to do the format conversions.

The proper declaration for `printf` is

```
int printf(char *fmt, ...)
```

where the declaration `...` means that the number and types of these arguments may vary. The declaration `...` can only appear at the end of an argument list. Our `minprintf` is declared as

```
void minprintf(char *fmt, ...)
```

since we will not return the character count that `printf` does.

The tricky bit is how `minprintf` walks along the argument list when the list doesn't even have a name. The standard header `<stdarg.h>` contains a set of macro definitions that define how to step through an argument list. The implementation of this header will vary from machine to machine, but the interface it presents is uniform.

The type `va_list` is used to declare a variable that will refer to each argument in turn; in `minprintf`, this variable is called `ap`, for ``argument pointer.'' The macro `va_start` initializes `ap` to point to the first unnamed argument. It must be called once before `ap` is used. There must be at least one named argument; the final named argument is used by `va_start` to get started.

Each call of `va_arg` returns one argument and steps `ap` to the next; `va_arg` uses a type name to determine what type to return and how big a step to take. Finally, `va_end` does whatever cleanup is necessary. It must be called before the program returns.

These properties form the basis of our simplified `printf`:

```
#include <stdarg.h>

/* minprintf: minimal printf with variable argument list */
void minprintf(char *fmt, ...)
{
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval;
```

```

int ival;
double dval;

va_start(ap, fmt); /* make ap point to 1st unnamed arg */
for (p = fmt; *p; p++) {
    if (*p != '%') {
        putchar(*p);
        continue;
    }
    switch (*++p) {
    case 'd':
        ival = va_arg(ap, int);
        printf("%d", ival);
        break;
    case 'f':
        dval = va_arg(ap, double);
        printf("%f", dval);
        break;
    case 's':
        for (sval = va_arg(ap, char *); *sval; sval++)
            putchar(*sval);
        break;
    default:
        putchar(*p);
        break;
    }
}
va_end(ap); /* clean up when done */
}

```

Exercise 7-3. Revise `minprintf` to handle more of the other facilities of `printf`.

7.4 Formatted Input - `scanf`

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
```

`scanf` reads characters from the standard input, interprets them according to the specification in `format`, and stores the results through the remaining arguments. The `format` argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored. As with `printf`, this section is a summary of the most useful features, not an exhaustive list.

`scanf` stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On the end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to `scanf` resumes searching immediately after the last character already converted.

There is also a function `sscanf` that reads from a string instead of the standard input:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

It scans the `string` according to the format in `format` and stores the resulting values through `arg1`, `arg2`, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are not ignored.

- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l or L indicating the width of the target, and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that `scanf` will read across boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer, as required by the call-by-value semantics of C. Conversion characters are shown in Table 7.2.

Table 7.2: Basic Scanf Conversions

Character	Input Data; Argument type
d	decimal integer; int *
i	integer; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); int *
u	unsigned decimal integer; unsigned int *
x	hexadecimal integer (with or without leading 0x or 0X); int *
c	characters; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use %ls
s	character string (not quoted); char *, pointing to an array of characters long enough for the string and a terminating '\0' that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; float *
%	literal %; no assignment is made.

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list.

As a first example, the rudimentary calculator of [Chapter 4](#) can be written with `scanf` to do the input conversion:

```
#include <stdio.h>

main() /* rudimentary calculator */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Suppose we want to read input lines that contain dates of the form

25 Dec 1988

The `scanf` statement is

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

No `&` is used with `monthname`, since an array name is a pointer.

Literal characters can appear in the `scanf` format string; they must match the same characters in the input. So we could read dates of the form `mm/dd/yy` with the `scanf` statement:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

`scanf` ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with `scanf`. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line); /* 25 Dec 1988 form */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line); /* mm/dd/yy form */
    else
        printf("invalid: %s\n", line); /* invalid form */
}
```

Calls to `scanf` can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by `scanf`.

A final warning: the arguments to `scanf` and `sscanf` *must* be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

This error is not generally detected at compile time.

Exercise 7-4. Write a private version of `scanf` analogous to `minprintf` from the previous section.

Exercise 5-5. Rewrite the postfix calculator of [Chapter 4](#) to use `scanf` and/or `sscanf` to do the input and number conversion.

7.5 File Access

The examples so far have all read the standard input and written the standard output, which are automatically defined for a program by the local operating system.

The next step is to write a program that accesses a file that is *not* already connected to the program. One program that illustrates the need for such operations is `cat`, which concatenates a set of named files into the standard output. `cat` is used for printing files on the screen, and as a general-purpose input collector for programs that do not have the capability of accessing files by name. For example, the command

```
cat x.c y.c
```

prints the contents of the files `x.c` and `y.c` (and nothing else) on the standard output.

The question is how to arrange for the named files to be read - that is, how to connect the external names that a user thinks of to the statements that read the data.

The rules are simple. Before it can be read or written, a file has to be *opened* by the library function `fopen`. `fopen` takes an external name like `x.c` or `y.c`, does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the *file pointer*, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. Users don't need to know the details, because the definitions obtained from `<stdio.h>` include a structure declaration called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. Notice that `FILE` is a type name, like `int`, not a structure tag; it is defined with a `typedef`. (Details of how `fopen` can be implemented on the UNIX system are given in [Section 8.5](#).)

The call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is a character string containing the name of the file. The second argument is the *mode*, also a character string, which indicates how one intends to use the file. Allowable modes include read ("`r`"), write ("`w`"), and append ("`a`"). Some systems distinguish between text and binary files; for the latter, a "`b`" must be appended to the mode string.

If a file that does not exist is opened for writing or appending, it is created if possible. Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves them. Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission. If there is any error, `fopen` will return `NULL`. (The error can be identified more precisely; see the discussion of error-handling functions at the end of [Section 1 in Appendix B](#).)

The next thing needed is a way to read or write the file once it is open. `getc` returns the next character from a file; it needs the file pointer to tell it which file.

```
int getc(FILE *fp)
```

`getc` returns the next character from the stream referred to by `fp`; it returns `EOF` for end of file or error.

`putc` is an output function:

```
int putc(int c, FILE *fp)
```

`putc` writes the character `c` to the file `fp` and returns the character written, or `EOF` if an error occurs. Like `getchar` and `putchar`, `getc` and `putc` may be macros instead of functions.

When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`, and are declared in `<stdio.h>`. Normally `stdin` is connected to the keyboard and `stdout` and `stderr` are connected to the screen, but `stdin` and `stdout` may be redirected to files or pipes as described in [Section 7.1](#).

`getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin`, and `stdout` as follows:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

For formatted input or output of files, the functions `fscanf` and `fprintf` may be used. These are identical to `scanf` and `printf`, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

With these preliminaries out of the way, we are now in a position to write the program `cat` to concatenate files. The design is one that has been found convenient for many programs. If there are command-line arguments, they are interpreted as filenames, and processed in order. If there are no arguments, the standard input is processed.

```
#include <stdio.h>

/* cat: concatenate files, version 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *)

    if (argc == 1) /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while(--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: can't open %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy: copy file ifp to file ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

The file pointers `stdin` and `stdout` are objects of type `FILE *`. They are constants, however, *not* variables, so it is not possible to assign to them.

The function

```
int fclose(FILE *fp)
```

is the inverse of `fopen`, it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since most operating systems have some limit on the number of files that a program may have open simultaneously, it's a good idea to free the file pointers when they are no longer needed, as we did in `cat`. There is also another reason for `fclose` on an output file - it flushes the buffer in which `putc` is collecting output. `fclose` is called automatically for each open file when a program terminates normally. (You can close `stdin` and `stdout` if they are not needed. They can also be reassigned by the library function `freopen`.)

7.6 Error Handling - Stderr and Exit

The treatment of errors in `cat` is not ideal. The trouble is that if one of the files can't be accessed for some reason, the diagnostic is printed at the end of the concatenated output. That might be acceptable if the output is going to a screen, but not if it's going into a file or into another program via a pipeline.

To handle this situation better, a second output stream, called `stderr`, is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` normally appears on the screen even if the standard output is redirected.

Let us revise `cat` to write its error messages on the standard error.

```
#include <stdio.h>

/* cat: concatenate files, version 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0]; /* program name for errors */

    if (argc == 1) /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n",
                        prog, *argv);
                exit(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: error writing stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

The program signals errors in two ways. First, the diagnostic output produced by `fprintf` goes to `stderr`, so it finds its way to the screen instead of disappearing down a pipeline or into an output file. We included the program name, from `argv[0]`, in the message, so if this program is used with others, the source of an error is identified.

Second, the program uses the standard library function `exit`, which terminates program execution when it is called. The argument of `exit` is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. Conventionally, a return value of 0 signals that all is well; non-zero values usually signal abnormal situations. `exit` calls `fclose` for each open output file, to flush out any buffered output.

Within `main`, return `expr` is equivalent to `exit(expr)`. `exit` has the advantage that it can be called from other functions, and that calls to it can be found with a pattern-searching program like those in [Chapter 5](#).

The function `ferror` returns non-zero if an error occurred on the stream `fp`.

```
int ferror(FILE *fp)
```

Although output errors are rare, they do occur (for example, if a disk fills up), so a production program should check this as well.

The function `feof(FILE *)` is analogous to `ferror`; it returns non-zero if end of file has occurred on the specified file.

```
int feof(FILE *fp)
```

We have generally not worried about exit status in our small illustrative programs, but any serious program should take care to return sensible, useful status values.

7.7 Line Input and Output

The standard library provides an input and output routine `fgets` that is similar to the `getline` function that we have used in earlier chapters:

```
char *fgets(char *line, int maxline, FILE *fp)
```

`fgets` reads the next input line (including the newline) from file `fp` into the character array `line`; at most `maxline-1` characters will be read. The resulting line is terminated with '`\0`'. Normally `fgets` returns `line`; on end of file or error it returns `NULL`. (Our `getline` returns the line length, which is a more useful value; zero means end of file.)

For output, the function `fputs` writes a string (which need not contain a newline) to a file:

```
int fputs(char *line, FILE *fp)
```

It returns `EOF` if an error occurs, and non-negative otherwise.

The library functions `gets` and `puts` are similar to `fgets` and `fputs`, but operate on `stdin` and `stdout`. Confusingly, `gets` deletes the terminating '`\n`', and `puts` adds it.

To show that there is nothing special about functions like `fgets` and `fputs`, here they are, copied from the standard library on our system:

```
/* fgets: get at most n chars from iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: put string s on file iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

For no obvious reason, the standard specifies different return values for `ferror` and `fputs`.

It is easy to implement our `getline` from `fgets`:

```
/* getline: read a line, return length */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
```

```

    return strlen(line);
}

```

Exercise 7-6. Write a program to compare two files, printing the first line where they differ.

Exercise 7-7. Modify the pattern finding program of [Chapter 5](#) to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found?

Exercise 7-8. Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file.

7.8 Miscellaneous Functions

The standard library provides a wide variety of functions. This section is a brief synopsis of the most useful. More details and many other functions can be found in [Appendix B](#).

7.8.1 String Operations

We have already mentioned the string functions `strlen`, `strcpy`, `strcat`, and `strcmp`, found in `<string.h>`. In the following, `s` and `t` are `char *`'s, and `c` and `n` are `ints`.

<code>strcat(s,t)</code>	concatenate <code>t</code> to end of <code>s</code>
<code>strncat(s,t,n)</code>	concatenate <code>n</code> characters of <code>t</code> to end of <code>s</code>
<code>strcmp(s,t)</code>	return negative, zero, or positive for <code>s < t</code> , <code>s == t</code> , <code>s > t</code>
<code>strncmp(s,t,n)</code>	same as <code>strcmp</code> but only in first <code>n</code> characters
<code>strcpy(s,t)</code>	copy <code>t</code> to <code>s</code>
<code>strncpy(s,t,n)</code>	copy at most <code>n</code> characters of <code>t</code> to <code>s</code>
<code>strlen(s)</code>	return length of <code>s</code>
<code>strchr(s,c)</code>	return pointer to first <code>c</code> in <code>s</code> , or <code>NULL</code> if not present
<code> strrchr(s,c)</code>	return pointer to last <code>c</code> in <code>s</code> , or <code>NULL</code> if not present

7.8.2 Character Class Testing and Conversion

Several functions from `<cctype.h>` perform character tests and conversions. In the following, `c` is an `int` that can be represented as an `unsigned char` or `EOF`. The function returns `int`.

<code>isalpha(c)</code>	non-zero if <code>c</code> is alphabetic, 0 if not
<code>isupper(c)</code>	non-zero if <code>c</code> is upper case, 0 if not
<code>islower(c)</code>	non-zero if <code>c</code> is lower case, 0 if not
<code>isdigit(c)</code>	non-zero if <code>c</code> is digit, 0 if not
<code>isalnum(c)</code>	non-zero if <code>isalpha(c)</code> or <code>isdigit(c)</code> , 0 if not
<code>isspace(c)</code>	non-zero if <code>c</code> is blank, tab, newline, return, formfeed, vertical tab
<code>toupper(c)</code>	return <code>c</code> converted to upper case
<code>tolower(c)</code>	return <code>c</code> converted to lower case

7.8.3 Ungetc

The standard library provides a rather restricted version of the function `ungetch` that we wrote in [Chapter 4](#); it is called `ungetc`.

```

int ungetc(int c, FILE *fp)

```

pushes the character `c` back onto file `fp`, and returns either `c`, or `EOF` for an error. Only one character of pushback is guaranteed per file. `ungetc` may be used with any of the input functions like `scanf`, `getc`, or `getchar`.

7.8.4 Command Execution

The function `system(char *s)` executes the command contained in the character string `s`, then resumes execution of the current program. The contents of `s` depend strongly on the local operating system. As a trivial example, on UNIX systems, the statement

```
system("date");
```

causes the program date to be run; it prints the date and time of day on the standard output. system returns a system-dependent integer status from the command executed. In the UNIX system, the status return is the value returned by exit.

7.8.5 Storage Management

The functions malloc and calloc obtain blocks of memory dynamically.

```
void *malloc(size_t n)
```

returns a pointer to n bytes of uninitialized storage, or NULL if the request cannot be satisfied.

```
void *calloc(size_t n, size_t size)
```

returns a pointer to enough free space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero.

The pointer returned by malloc or calloc has the proper alignment for the object in question, but it must be cast into the appropriate type, as in

```
int *ip;  
ip = (int *) calloc(n, sizeof(int));
```

free(p) frees the space pointed to by p, where p was originally obtained by a call to malloc or calloc. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling malloc or calloc.

It is also an error to use something after it has been freed. A typical but incorrect piece of code is this loop that frees items from a list:

```
for (p = head; p != NULL; p = p->next) /* WRONG */  
    free(p);
```

The right way is to save whatever is needed before freeing:

```
for (p = head; p != NULL; p = q) {  
    q = p->next;  
    free(p);  
}
```

[Section 8.7](#) shows the implementation of a storage allocator like malloc, in which allocated blocks may be freed in any order.

7.8.6 Mathematical Functions

There are more than twenty mathematical functions declared in `<math.h>`; here are some of the more frequently used. Each takes one or two double arguments and returns a double.

<code>sin(x)</code>	sine of x , x in radians
<code>cos(x)</code>	cosine of x , x in radians
<code>atan2(y,x)</code>	arctangent of y/x , in radians
<code>exp(x)</code>	exponential function e^x
<code>log(x)</code>	natural (base e) logarithm of x ($x > 0$)
<code>log10(x)</code>	common (base 10) logarithm of x ($x > 0$)
<code>pow(x,y)</code>	x^y
<code>sqrt(x)</code>	square root of x ($x > 0$)
<code>fabs(x)</code>	absolute value of x

7.8.7 Random Number generation

The function rand() computes a sequence of pseudo-random integers in the range zero to RAND_MAX, which is defined in `<stdlib.h>`. One way to produce random floating-point numbers greater than or equal to zero but less than one is

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(If your library already provides a function for floating-point random numbers, it is likely to have better statistical properties than this one.)

The function `srand(unsigned)` sets the seed for `rand`. The portable implementation of `rand` and `srand` suggested by the standard appears in [Section 2.7](#).

Exercise 7-9. Functions like `isupper` can be implemented to save space or to save time. Explore both possibilities.

Chapter 8 - The UNIX System Interface

The UNIX operating system provides its services through a set of *system calls*, which are in effect functions within the operating system that may be called by user programs. This chapter describes how to use some of the most important system calls from C programs. If you use UNIX, this should be directly helpful, for it is sometimes necessary to employ system calls for maximum efficiency, or to access some facility that is not in the library. Even if you use C on a different operating system, however, you should be able to glean insight into C programming from studying these examples; although details vary, similar code will be found on any system. Since the ANSI C library is in many cases modeled on UNIX facilities, this code may help your understanding of the library as well.

This chapter is divided into three major parts: input/output, file system, and storage allocation. The first two parts assume a modest familiarity with the external characteristics of UNIX systems.

[Chapter 7](#) was concerned with an input/output interface that is uniform across operating systems. On any particular system the routines of the standard library have to be written in terms of the facilities provided by the host system. In the next few sections we will describe the UNIX system calls for input and output, and show how parts of the standard library can be implemented with them.

8.1 File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even keyboard and screen, are files in the file system. This means that a single homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before you read and write a file, you must inform the system of your intent to do so, a process called *opening* the file. If you are going to write on a file it may also be necessary to create it or to discard its previous contents. The system checks your right to do so (Does the file exist? Do you have permission to access it?) and if all is well, returns to the program a small non-negative integer called a *file descriptor*. Whenever input or output is to be done on the file, the file descriptor is used instead of the name to identify the file. (A file descriptor is analogous to the file pointer used by the standard library, or to the file handle of MS-DOS.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

Since input and output involving keyboard and screen is so common, special arrangements exist to make this convenient. When the command interpreter (the ``shell'') runs a program, three files are open, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error. If a program reads 0 and writes 1 and 2, it can do input and output without worrying about opening files.

The user of a program can redirect I/O to and from files with < and >:

```
prog <infile >outfile
```

In this case, the shell changes the default assignments for the file descriptors 0 and 1 to the named files. Normally file descriptor 2 remains attached to the screen, so error messages can go there. Similar observations hold for input or output associated with a pipe. In all cases, the file assignments are changed by the shell, not by the program. The program does not know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

8.2 Low Level I/O - Read and Write

Input and output uses the `read` and `write` system calls, which are accessed from C programs through two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a character array in your program where the data is to go to or to come from. The third argument is the number is the number of bytes to be transferred.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Each call returns a count of the number of bytes transferred. On reading, the number of bytes returned may be less than the number requested. A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the return value is the number of bytes written; an error has occurred if this isn't equal to the number requested.

Any number of bytes can be read or written in one call. The most common values are `1`, which means one character at a time ("unbuffered"), and a number like `1024` or `4096` that corresponds to a physical block size on a peripheral device. Larger sizes will be more efficient because fewer system calls will be made.

Putting these facts together, we can write a simple program to copy its input to its output, the equivalent of the file copying program written for [Chapter 1](#). This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#include "syscalls.h"

main() /* copy input to output */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

We have collected function prototypes for the system calls into a file called `syscalls.h` so we can include it in the programs of this chapter. This name is not standard, however.

The parameter `BUFSIZ` is also defined in `syscalls.h`; its value is a good size for the local system. If the file size is not a multiple of `BUFSIZ`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher-level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` that does unbuffered input, by reading the standard input one character at a time.

```
#include "syscalls.h"

/* getchar: unbuffered single character input */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

`c` must be a `char`, because `read` needs a character pointer. Casting `c` to `unsigned char` in the return statement eliminates any problem of sign extension.

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.

```
#include "syscalls.h"

/* getchar: simple buffered version */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) { /* buffer is empty */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

If these versions of `getchar` were to be compiled with `<stdio.h>` included, it would be necessary to `#undef` the name `getchar` in case it is implemented as a macro.

8.3 Open, Creat, Close, Unlink

Other than the default standard input, output and error, you must explicitly open files in order to read or write them. There are two system calls for this, `open` and `creat` [sic].

`open` is rather like the `fopen` discussed in [Chapter 7](#), except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`. `open` returns `-1` if any error occurs.

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

As with `fopen`, the `name` argument is a character string containing the filename. The second argument, `flags`, is an `int` that specifies how the file is to be opened; the main values are

- `O_RDONLY` open for reading only
- `O_WRONLY` open for writing only
- `O_RDWR` open for both reading and writing

These constants are defined in `<fcntl.h>` on System V UNIX systems, and in `<sys/file.h>` on Berkeley (BSD) versions.

To open an existing file for reading,

```
fd = open(name, O_RDONLY, 0);
```

The `perms` argument is always zero for the uses of `open` that we will discuss.

It is an error to try to open a file that does not exist. The system call `creat` is provided to create new files, or to re-write old ones.

```
int creat(char *name, int perms);

fd = creat(name, perms);
```

returns a file descriptor if it was able to create the file, and `-1` if not. If the file already exists, `creat` will truncate it to zero length, thereby discarding its previous contents; it is not an error to `creat` a file that already exists.

If the file does not already exist, `creat` creates it with the permissions specified by the `perms` argument. In the UNIX file system, there are nine bits of permission information associated with a file that control read, write and execute access for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is convenient for specifying the

permissions. For example, 0775 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX program `cp`, which copies one file to another. Our version copies only one file, it does not permit the second argument to be a directory, and it invents permissions instead of copying them.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666      /* RW for owner, group, others */

void error(char *, ...);

/* cp: copy f1 to f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3)
        error("Usage: cp from to");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: can't create %s, mode %o",
              argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error on file %s", argv[2]);
    return 0;
}
```

This program creates the output file with fixed permissions of 0666. With the `stat` system call, described in [Section 8.6](#), we can determine the mode of an existing file and thus give the same mode to the copy.

Notice that the function `error` is called with variable argument lists much like `printf`. The implementation of `error` illustrates how to use another member of the `printf` family. The standard library function `vprintf` is like `printf` except that the variable argument list is replaced by a single argument that has been initialized by calling the `va_start` macro. Similarly, `vfprintf` and `vsprintf` match `fprintf` and `sprintf`.

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

/* error: print an error message and die */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "error: ");
    vprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

There is a limit (often about 20) on the number of files that a program may open simultaneously. Accordingly, any program that intends to process many files must be prepared to re-use file descriptors. The function `close(int fd)` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file; it

corresponds to `fclose` in the standard library except that there is no buffer to flush. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(char *name)` removes the file `name` from the file system. It corresponds to the standard library function `remove`.

Exercise 8-1. Rewrite the program `cat` from [Chapter 7](#) using `read`, `write`, `open`, and `close` instead of their standard library equivalents. Perform experiments to determine the relative speeds of the two versions.

8.4 Random Access - Lseek

Input and output are normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without reading or writing any data:

```
long lseek(int fd, long offset, int origin);
```

sets the current position in the file whose descriptor is `fd` to `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file (the redirection `>>` in the UNIX shell, or "a" for `fopen`), seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("`rewind`"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0` or just as `0` if `lseek` is properly declared.

With `lseek`, it is possible to treat files more or less like arrays, at the price of slower access. For example, the following function reads any number of bytes from any arbitrary place in a file. It returns the number read, or `-1` on error.

```
#include "syscalls.h"

/*get:  read n bytes from position pos */
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0) /* get to pos */
        return read(fd, buf, n);
    else
        return -1;
}
```

The return value from `lseek` is a long that gives the new position in the file, or `-1` if an error occurs. The standard library function `fseek` is similar to `lseek` except that the first argument is a `FILE *` and the return is non-zero if an error occurred.

8.5 Example - An implementation of Fopen and Getc

Let us illustrate how some of these pieces fit together by showing an implementation of the standard library routines `fopen` and `getc`.

Recall that files in the standard library are described by file pointers rather than file descriptors. A file pointer is a pointer to a structure that contains several pieces of information about the file: a pointer to a buffer, so the file can be read in large chunks; a count of the number of characters left in the buffer; a pointer to the next character position in the buffer; the file descriptor; and flags describing read/write mode, error status, etc.

The data structure that describes a file is contained in `<stdio.h>`, which must be included (by `#include`) in any source file that uses routines from the standard input/output library. It is also included by functions in that library. In the following excerpt from a typical `<stdio.h>`, names that are intended for use only by functions of the library begin with an underscore so they are less likely to collide with names in a user's program. This convention is used by all standard library routines.

```

#define NULL      0
#define EOF       (-1)
#define BUFSIZ    1024
#define OPEN_MAX   20 /* max #files open at once */

typedef struct _iobuf {
    int cnt;          /* characters left */
    char *ptr;        /* next character position */
    char *base;       /* location of buffer */
    int flag;         /* mode of file access */
    int fd;           /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin    (&_iob[0])
#define stdout   (&_iob[1])
#define stderr   (&_iob[2])

enum _flags {
    _READ    = 01,    /* file open for reading */
    _WRITE   = 02,    /* file open for writing */
    _UNBUF   = 04,    /* file is unbuffered */
    _EOF     = 010,   /* EOF has occurred on this file */
    _ERR     = 020    /* error occurred on this file */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)      ((p)->flag & _EOF) != 0
#define ferror(p)    ((p)->flag & _ERR) != 0
#define fileno(p)    ((p)->fd)

#define getc(p)      (--(p)->cnt >= 0 \
                  ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p)    (--(p)->cnt >= 0 \
                  ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()    getc(stdin)
#define putchar(x)   putc((x), stdout)

```

The `getc` macro normally decrements the count, advances the pointer, and returns the character. (Recall that a long `#define` is continued with a backslash.) If the count goes negative, however, `getc` calls the function `_fillbuf` to replenish the buffer, re-initialize the structure contents, and return a character. The characters are returned `unsigned`, which ensures that all characters will be positive.

Although we will not discuss any details, we have included the definition of `putc` to show that it operates in much the same way as `getc`, calling a function `_flushbuf` when its buffer is full. We have also included macros for accessing the error and end-of-file status and the file descriptor.

The function `fopen` can now be written. Most of `fopen` is concerned with getting the file opened and positioned at the right place, and setting the flag bits to indicate the proper state. `fopen` does not allocate any buffer space; this is done by `_fillbuf` when the file is first read.

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666 /* RW for owner, group, others */

FILE *fopen(char *name, char *mode)
{
    int fd;
    FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
            break; /* found free slot */
    if (fp >= _iob + OPEN_MAX) /* no free slots */
        return NULL;

    if (*mode == 'w')
        fd = creat(name, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(name, O_WRONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1) /* couldn't access name */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

This version of `fopen` does not handle all of the access mode possibilities of the standard, though adding them would not take much code. In particular, our `fopen` does not recognize the ``b'' that signals binary access, since that is meaningless on UNIX systems, nor the ``+'' that permits both reading and writing.

The first call to `getc` for a particular file finds a count of zero, which forces a call of `_fillbuf`. If `_fillbuf` finds that the file is not open for reading, it returns `EOF` immediately. Otherwise, it tries to allocate a buffer (if reading is to be buffered).

Once the buffer is established, `_fillbuf` calls `read` to fill it, sets the count and pointers, and returns the character at the beginning of the buffer. Subsequent calls to `_fillbuf` will find a buffer allocated.

```
#include "syscalls.h"

/* _fillbuf: allocate and fill input buffer */
int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag & (_READ | _EOF_ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL) /* no buffer yet */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF; /* can't get buffer */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
```

```

        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}

```

The only remaining loose end is how everything gets started. The array `_iob` must be defined and initialized for `stdin`, `stdout` and `stderr`:

```

FILE _iob[OPEN_MAX] = { /* stdin, stdout, stderr */
    { 0, (char *) 0, (char *) 0, _READ, 0 },
    { 0, (char *) 0, (char *) 0, _WRITE, 1 },
    { 0, (char *) 0, (char *) 0, _WRITE, | _UNBUF, 2 }
};

```

The initialization of the `flag` part of the structure shows that `stdin` is to be read, `stdout` is to be written, and `stderr` is to be written unbuffered.

Exercise 8-2. Rewrite `fopen` and `_fillbuf` with fields instead of explicit bit operations. Compare code size and execution speed.

Exercise 8-3. Design and write `_flushbuf`, `fflush`, and `fclose`.

Exercise 8-4. The standard library function

```
int fseek(FILE *fp, long offset, int origin)
```

is identical to `lseek` except that `fp` is a file pointer instead of a file descriptor and return value is an `int` status, not a position. Write `fseek`. Make sure that your `fseek` coordinates properly with the buffering done for the other functions of the library.

8.6 Example - Listing Directories

A different kind of file system interaction is sometimes called for - determining information *about* a file, not what it contains. A directory-listing program such as the UNIX command `ls` is an example - it prints the names of files in a directory, and, optionally, other information, such as sizes, permissions, and so on. The MS-DOS `dir` command is analogous.

Since a UNIX directory is just a file, `ls` need only read it to retrieve the filenames. But it is necessary to use a system call to access other information about a file, such as its size. On other systems, a system call may be needed even to access filenames; this is the case on MS-DOS for instance. What we want is provide access to the information in a relatively system-independent way, even though the implementation may be highly system-dependent.

We will illustrate some of this by writing a program called `fsize`. `fsize` is a special form of `ls` that prints the sizes of all files named in its commandline argument list. If one of the files is a directory, `fsize` applies itself recursively to that directory. If there are no arguments at all, it processes the current directory.

Let us begin with a short review of UNIX file system structure. A *directory* is a file that contains a list of filenames and some indication of where they are located. The ``location'' is an index into another table called the ``inode list.'' The *inode* for a file is where all information about the file except its name is kept. A directory entry generally consists of only two items, the filename and an inode number.

Regrettably, the format and precise contents of a directory are not the same on all versions of the system. So we will divide the task into two pieces to try to isolate the non-portable parts. The outer level defines a structure called a `Dirent` and three routines `opendir`, `readdir`, and `closedir` to provide system-independent access to the name and inode number in a directory entry. We will write `fsize` with this interface. Then we will show how to implement these on

systems that use the same directory structure as Version 7 and System V UNIX; variants are left as exercises.

The `Dirent` structure contains the inode number and the name. The maximum length of a filename component is `NAME_MAX`, which is a system-dependent value. `opendir` returns a pointer to a structure called `DIR`, analogous to `FILE`, which is used by `readdir` and `closedir`. This information is collected into a file called `dirent.h`.

```
#define NAME_MAX    14 /* longest filename component; */
                      /* system-dependent */

typedef struct {      /* portable directory entry */
    long ino;          /* inode number */
    char name[NAME_MAX+1]; /* name + '\0' terminator */
} DIRENT;

typedef struct {      /* minimal DIR: no buffering, etc. */
    int fd;            /* file descriptor for the directory */
    DIRENT d;          /* the directory entry */
} DIR;

DIR *opendir(char *dirname);
DIRENT *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

The system call `stat` takes a filename and returns all of the information in the inode for that file, or `-1` if there is an error. That is,

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

fills the structure `stbuf` with the inode information for the file name. The structure describing the value returned by `stat` is in `<sys/stat.h>`, and typically looks like this:

```
struct stat /* inode information returned by stat */
{
    dev_t     st_dev;      /* device of inode */
    ino_t     st_ino;      /* inode number */
    short     st_mode;     /* mode bits */
    short     st_nlink;    /* number of links to file */
    short     st_uid;      /* owners user id */
    short     st_gid;      /* owners group id */
    dev_t     st_rdev;     /* for special files */
    off_t     st_size;     /* file size in characters */
    time_t    st_atime;    /* time last accessed */
    time_t    st_mtime;    /* time last modified */
    time_t    st_ctime;    /* time originally created */
};
```

Most of these values are explained by the comment fields. The types like `dev_t` and `ino_t` are defined in `<sys/types.h>`, which must be included too.

The `st_mode` entry contains a set of flags describing the file. The flag definitions are also included in `<sys/types.h>`; we need only the part that deals with file type:

```
#define S_IFMT    0160000 /* type of file: */
#define S_IFDIR   0040000 /* directory */
#define S_IFCHR   0020000 /* character special */
#define S_IFBLK   0060000 /* block special */
#define S_IFREG   0010000 /* regular */
/* ... */
```

Now we are ready to write the program `fsize`. If the mode obtained from `stat` indicates that a file is not a directory, then the size is at hand and can be printed directly. If the name is a directory, however, then we have to process that directory one file at a time; it may in turn contain sub-directories, so the process is recursive.

The main routine deals with command-line arguments; it hands each argument to the function `fsize`.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>      /* flags for read and write */
#include <sys/types.h>   /* typedefs */
#include <sys/stat.h>    /* structure returned by stat */
#include "dirent.h"

void fsize(char *)

/* print file name */
main(int argc, char **argv)
{
    if (argc == 1) /* default: current directory */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```

The function `fsize` prints the size of the file. If the file is a directory, however, `fsize` first calls `dirwalk` to handle all the files in it. Note how the flag names `S_IFMT` and `S_IFDIR` are used to decide if the file is a directory. Parenthesization matters, because the precedence of `&` is lower than that of `==`.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize: print the name of file "name" */
void fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't access %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

The function `dirwalk` is a general routine that applies a function to each file in a directory. It opens the directory, loops through the files in it, calling the function on each, then closes the directory and returns. Since `fsize` calls `dirwalk` on each directory, the two functions call each other recursively.

```
#define MAX_PATH 1024

/* dirwalk: apply fcn to all files in dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    DIRENT *dp;
    DIR *dfd;
```

```

if ((dfd = opendir(dir)) == NULL) {
    fprintf(stderr, "dirwalk: can't open %s\n", dir);
    return;
}
while ((dp = readdir(dfd)) != NULL) {
    if (strcmp(dp->name, ".") == 0
        || strcmp(dp->name, ".."))
        continue; /* skip self and parent */
    if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
        fprintf(stderr, "dirwalk: name %s %s too long\n",
                dir, dp->name);
    else {
        sprintf(name, "%s/%s", dir, dp->name);
        (*fcn)(name);
    }
}
closedir(dfd);
}

```

Each call to `readdir` returns a pointer to information for the next file, or `NULL` when there are no files left. Each directory always contains entries for itself, called `"."`, and its parent, `".."`; these must be skipped, or the program will loop forever.

Down to this last level, the code is independent of how directories are formatted. The next step is to present minimal versions of `opendir`, `readdir`, and `closedir` for a specific system. The following routines are for Version 7 and System V UNIX systems; they use the directory information in the header `<sys/dir.h>`, which looks like this:

```

#ifndef DIRSIZ
#define DIRSIZ 14
#endif
struct direct { /* directory entry */
    ino_t d_ino; /* inode number */
    char d_name[DIRSIZ]; /* long name does not have '\0' */
};

```

Some versions of the system permit much longer names and have a more complicated directory structure.

The type `ino_t` is a `typedef` that describes the index into the inode list. It happens to be `unsigned short` on the systems we use regularly, but this is not the sort of information to embed in a program; it might be different on a different system, so the `typedef` is better. A complete set of ``system'' types is found in `<sys/types.h>`.

`opendir` opens the directory, verifies that the file is a directory (this time by the system call `fstat`, which is like `stat` except that it applies to a file descriptor), allocates a directory structure, and records the information:

```

int fstat(int fd, struct stat *);

/* opendir: open a directory for readdir calls */
DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
        || fstat(fd, &stbuf) == -1
        || (stbuf.st_mode & S_IFMT) != S_IFDIR
        || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
        return NULL;
    dp->fd = fd;
    return dp;
}

```

`closedir` closes the directory file and frees the space:

```
/* closedir: close directory opened by opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

Finally, `readdir` uses `read` to read each directory entry. If a directory slot is not currently in use (because a file has been removed), the inode number is zero, and this position is skipped. Otherwise, the inode number and name are placed in a static structure and a pointer to that is returned to the user. Each call overwrites the information from the previous one.

```
#include <sys/dir.h> /* local directory structure */

/* readdir: read directory entries in sequence */
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf; /* local directory structure */
    static Dirent d;      /* return: portable structure */

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
          == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0) /* slot not in use */
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0'; /* ensure termination */
        return &d;
    }
    return NULL;
}
```

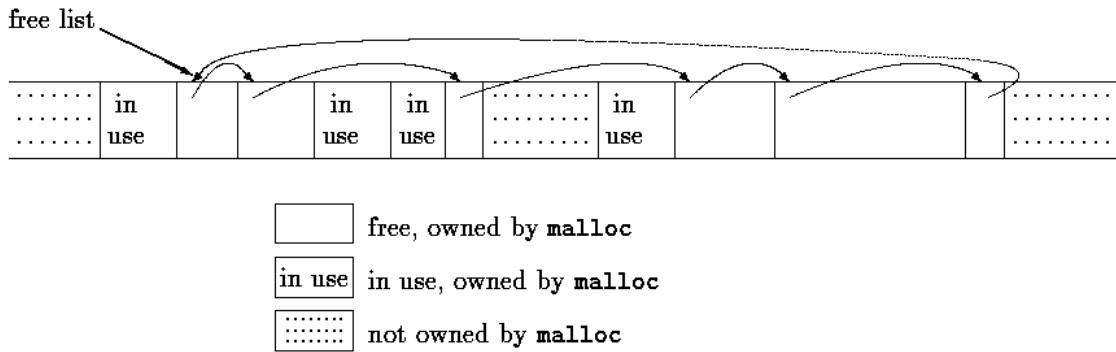
Although the `fsize` program is rather specialized, it does illustrate a couple of important ideas. First, many programs are not ``system programs''; they merely use information that is maintained by the operating system. For such programs, it is crucial that the representation of the information appear only in standard headers, and that programs include those headers instead of embedding the declarations in themselves. The second observation is that with care it is possible to create an interface to system-dependent objects that is itself relatively system-independent. The functions of the standard library are good examples.

Exercise 8-5. Modify the `fsize` program to print the other information contained in the inode entry.

8.7 Example - A Storage Allocator

In [Chapter 5](#), we presented a very limited stack-oriented storage allocator. The version that we will now write is unrestricted. Calls to `malloc` and `free` may occur in any order; `malloc` calls upon the operating system to obtain more memory as necessary. These routines illustrate some of the considerations involved in writing machine-dependent code in a relatively machine-independent way, and also show a real-life application of structures, unions and `typedef`.

Rather than allocating from a compiled-in fixed-size array, `malloc` will request space from the operating system as needed. Since other activities in the program may also request space without calling this allocator, the space that `malloc` manages may not be contiguous. Thus its free storage is kept as a list of free blocks. Each block contains a size, a pointer to the next block, and the space itself. The blocks are kept in order of increasing storage address, and the last block (highest address) points to the first.



When a request is made, the free list is scanned until a big-enough block is found. This algorithm is called ``first fit," by contrast with ``best fit," which looks for the smallest block that will satisfy the request. If the block is exactly the size requested it is unlinked from the list and returned to the user. If the block is too big, it is split, and the proper amount is returned to the user while the residue remains on the free list. If no big-enough block is found, another large chunk is obtained by the operating system and linked into the free list.

Freeing also causes a search of the free list, to find the proper place to insert the block being freed. If the block being freed is adjacent to a free block on either side, it is coalesced with it into a single bigger block, so storage does not become too fragmented. Determining the adjacency is easy because the free list is maintained in order of decreasing address.

One problem, which we alluded to in [Chapter 5](#), is to ensure that the storage returned by `malloc` is aligned properly for the objects that will be stored in it. Although machines vary, for each machine there is a most restrictive type: if the most restrictive type can be stored at a particular address, all other types may be also. On some machines, the most restrictive type is a `double`; on others, `int` or `long` suffices.

A free block contains a pointer to the next block in the chain, a record of the size of the block, and then the free space itself; the control information at the beginning is called the ``header.'' To simplify alignment, all blocks are multiples of the header size, and the header is aligned properly. This is achieved by a union that contains the desired header structure and an instance of the most restrictive alignment type, which we have arbitrarily made a `long`:

```

typedef long Align;      /* for alignment to long boundary */

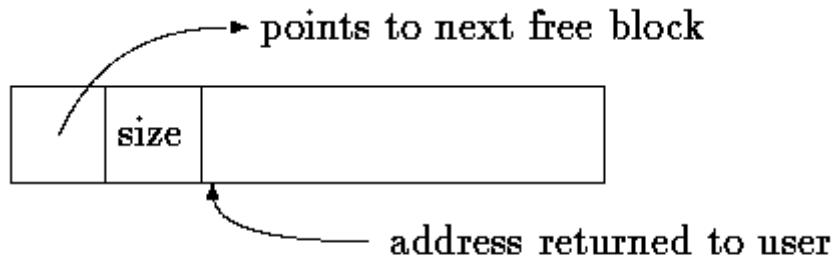
union header {           /* block header */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;     /* size of this block */
    } s;
    Align x;             /* force alignment of blocks */
};

typedef union header Header;

```

The `Align` field is never used; it just forces each header to be aligned on a worst-case boundary.

In `malloc`, the requested size in characters is rounded up to the proper number of header-sized units; the block that will be allocated contains one more unit, for the header itself, and this is the value recorded in the `size` field of the header. The pointer returned by `malloc` points at the free space, not at the header itself. The user can do anything with the space requested, but if anything is written outside of the allocated space the list is likely to be scrambled.



A block returned by malloc

The size field is necessary because the blocks controlled by `malloc` need not be contiguous - it is not possible to compute sizes by pointer arithmetic.

The variable `base` is used to get started. If `freep` is `NULL`, as it is at the first call of `malloc`, then a degenerate free list is created; it contains one block of size zero, and points to itself. In any case, the free list is then searched. The search for a free block of adequate size begins at the point (`freep`) where the last block was found; this strategy helps keep the list homogeneous. If a too-big block is found, the tail end is returned to the user; in this way the header of the original needs only to have its size adjusted. In all cases, the pointer returned to the user points to the free space within the block, which begins one unit beyond the header.

```

static Header base;           /* empty list to get started */
static Header *freep = NULL;  /* start of free list */

/* malloc: general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
    if ((prevp = freep) == NULL) { /* no free list yet */
        base.s.ptr = freeptr = prevptr = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; p = p->s.ptr) {
        if (p->s.size >= nunits) { /* big enough */
            if (p->s.size == nunits) /* exactly */
                prevp->s.ptr = p->s.ptr;
            else { /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void*)(p+1);
        }
        if (p == freep) /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL; /* none left */
    }
}

```

The function `morecore` obtains storage from the operating system. The details of how it does this vary from system to system. Since asking the system for memory is a comparatively expensive operation, we don't want to do that on every call to `malloc`, so `morecore` requests at least `NALLOC` units; this larger block will be chopped up as needed. After setting the size field, `morecore` inserts the additional memory into the arena by calling `free`.

The UNIX system call `sbrk(n)` returns a pointer to `n` more bytes of storage. `sbrk` returns `-1` if there was no space, even though `NULL` could have been a better design. The `-1` must be cast to `char *` so it can be compared with the return value. Again, casts make the function relatively immune to the details of pointer representation on different machines. There is still one assumption, however, that pointers to different blocks returned by `sbrk` can be meaningfully compared. This is not guaranteed by the standard, which permits pointer comparisons only within an array. Thus this version of `malloc` is portable only among machines for which general pointer comparison is meaningful.

```
#define NALLOC 1024 /* minimum #units to request */

/* morecore: ask system for more memory */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1) /* no space at all */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

`free` itself is the last thing. It scans the free list, starting at `freep`, looking for the place to insert the free block. This is either between two existing blocks or at the end of the list. In any case, if the block being freed is adjacent to either neighbor, the adjacent blocks are combined. The only troubles are keeping the pointers pointing to the right things and the sizes correct.

```
/* free: put block ap in free list */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1; /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break; /* freed block at start or end of arena */

    if (bp + bp->size == p->s.ptr) { /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->size == bp) { /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Although storage allocation is intrinsically machine-dependent, the code above illustrates how the machine dependencies can be controlled and confined to a very small part of the program. The use of `typedef` and `union` handles alignment (given that `sbrk` supplies an appropriate pointer). Casts arrange that pointer conversions are made explicit, and even cope with a badly-designed system interface. Even though the details here are related to storage allocation, the general approach is applicable to other situations as well.

Exercise 8-6. The standard library function `calloc(n, size)` returns a pointer to `n` objects of size `size`, with the storage initialized to zero. Write `calloc`, by calling `malloc` or by modifying it.

Exercise 8-7. `malloc` accepts a size request without checking its plausibility; `free` believes that the block it is asked to free contains a valid size field. Improve these routines so they make more pains with error checking.

Exercise 8-8. Write a routine `bfree(p, n)` that will free any arbitrary block `p` of `n` characters into the free list maintained by `malloc` and `free`. By using `bfree`, a user can add a static or external array to the free list at any time.

Appendix A - Reference Manual

A.1 Introduction

This manual describes the C language specified by the draft submitted to ANSI on 31 October, 1988, for approval as ``American Standard for Information Systems - programming Language C, X3.159-1989.'' The manual is an interpretation of the proposed standard, not the standard itself, although care has been taken to make it a reliable guide to the language.

For the most part, this document follows the broad outline of the standard, which in turn follows that of the first edition of this book, although the organization differs in detail. Except for renaming a few productions, and not formalizing the definitions of the lexical tokens or the preprocessor, the grammar given here for the language proper is equivalent to that of the standard.

Throughout this manual, commentary material is indented and written in smaller type, as this is. Most often these comments highlight ways in which ANSI Standard C differs from the language defined by the first edition of this book, or from refinements subsequently introduced in various compilers.

A.2 Lexical Conventions

A program consists of one or more *translation units* stored in files. It is translated in several phases, which are described in [Par.A.12](#). The first phases do low-level lexical transformations, carry out directives introduced by the lines beginning with the # character, and perform macro definition and expansion. When the preprocessing of [Par.A.12](#) is complete, the program has been reduced to a sequence of tokens.

A.2.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds and comments as described below (collectively, ``white space'') are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

A.2.2 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest, and they do not occur within a string or character literals.

A.2.3 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different. Identifiers may have any length, and for internal identifiers, at least the first 31 characters are significant; some implementations may take more characters significant. Internal identifiers include preprocessor macro names and all other names that do not have external linkage ([Par.A.11.2](#)). Identifiers with external linkage are more restricted: implementations may make as few as the first six characters significant, and may ignore case distinctions.

A.2.4 Keywords

The following identifiers are reserved for the use as keywords, and may not be used otherwise:

auto	double	int	struct
break	else	long	switch

case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Some implementations also reserve the words `fortran` and `asm`.

The keywords `const`, `signed`, and `volatile` are new with the ANSI standard; `enum` and `void` are new since the first edition, but in common use; `entry`, formerly reserved but never used, is no longer reserved.

A.2.5 Constants

There are several kinds of constants. Each has a data type; [Par.A.4.2](#) discusses the basic types:

constant:

integer-constant

character-constant

floating-constant

enumeration-constant

A.2.5.1 Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. Octal constants do not contain the digits 8 or 9. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15.

An integer constant may be suffixed by the letter u or U, to specify that it is unsigned. It may also be suffixed by the letter l or L to specify that it is long.

The type of an integer constant depends on its form, value and suffix. (See [Par.A.4](#) for a discussion of types). If it is unsuffixed and decimal, it has the first of these types in which its value can be represented: `int`, `long int`, `unsigned long int`. If it is unsuffixed, octal or hexadecimal, it has the first possible of these types: `int`, `unsigned int`, `long int`, `unsigned long int`. If it is suffixed by u or U, then `unsigned int`, `unsigned long int`. If it is suffixed by l or L, then `long int`, `unsigned long int`. If an integer constant is suffixed by UL, it is `unsigned long`.

The elaboration of the types of integer constants goes considerably beyond the first edition, which merely caused large integer constants to be long. The U suffixes are new.

A.2.5.2 Character Constants

A character constant is a sequence of one or more characters enclosed in single quotes as in '`x`'. The value of a character constant with only one character is the numeric value of the character in the machine's character set at execution time. The value of a multi-character constant is implementation-defined.

Character constants do not contain the ' character or newlines; in order to represent them, and certain other characters, the following escape sequences may be used:

newline	NL (LF)	\n	backslash	\	\\
horizontal tab	HT	\t	question mark	?	\?
vertical tab	VT	\v	single quote	'	\'
backspace	BS	\b	double quote	"	\"
carriage return	CR	\r	octal number	ooo	\ooo
formfeed	FF	\f	hex number	hh	\xhh
audible alert	BEL	\a			

The escape `\ooo` consists of the backslash followed by 1, 2, or 3 octal digits, which are taken to specify the value of the desired character. A common example of this construction is `\0` (not followed by a digit), which specifies the character NUL. The escape `\xhh` consists of the backslash, followed by `x`, followed by hexadecimal digits, which are taken to specify the value of the desired character. There is no limit on the number of digits, but the behavior is undefined if the resulting character value exceeds that of the largest character. For either octal or hexadecimal escape characters, if the implementation treats the `char` type as signed, the value is sign-extended as if cast to `char` type. If the character following the `\` is not one of those specified, the behavior is undefined.

In some implementations, there is an extended set of characters that cannot be represented in the `char` type. A constant in this extended set is written with a preceding `L`, for example `L'x'`, and is called a wide character constant. Such a constant has type `wchar_t`, an integral type defined in the standard header `<stddef.h>`. As with ordinary character constants, hexadecimal escapes may be used; the effect is undefined if the specified value exceeds that representable with `wchar_t`.

Some of these escape sequences are new, in particular the hexadecimal character representation. Extended characters are also new. The character sets commonly used in the Americas and western Europe can be encoded to fit in the `char` type; the main intent in adding `wchar_t` was to accommodate Asian languages.

A.2.5.3 Floating Constants

A floating constant consists of an integer part, a decimal part, a fraction part, an `e` or `E`, an optionally signed integer exponent and an optional type suffix, one of `f`, `F`, `l`, or `L`. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing. The type is determined by the suffix; `F` or `f` makes it `float`, `L` or `l` makes it `long double`, otherwise it is `double`.

A.2.5.4 Enumeration Constants

Identifiers declared as enumerators (see [Par.A.8.4](#)) are constants of type `int`.

A.2.6 String Literals

A string literal, also called a string constant, is a sequence of characters surrounded by double quotes as in `"..."`. A string has type ``array of characters'' and storage class `static` (see [Par.A.3](#) below) and is initialized with the given characters. Whether identical string literals are distinct is implementation-defined, and the behavior of a program that attempts to alter a string literal is undefined.

Adjacent string literals are concatenated into a single string. After any concatenation, a null byte `\0` is appended to the string so that programs that scan the string can find its end. String literals do not contain newline or double-quote characters; in order to represent them, the same escape sequences as for character constants are available.

As with character constants, string literals in an extended character set are written with a preceding `L`, as in `L"..."`. Wide-character string literals have type ``array of `wchar_t`.'' Concatenation of ordinary and wide string literals is undefined.

The specification that string literals need not be distinct, and the prohibition against modifying them, are new in the ANSI standard, as is the concatenation of adjacent string literals. Wide-character string literals are new.

A.3 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in `typewriter` style. Alternative categories are usually listed on separate lines; in a few cases, a long set of narrow alternatives is presented on one line, marked

by the phrase ``one of.'' An optional terminal or nonterminal symbol carries the subscript ``*opt*,'' so that, for example,

$$\{ \ expression_{opt} \ }$$

means an optional expression, enclosed in braces. The syntax is summarized in [Par.A.13](#).

Unlike the grammar given in the first edition of this book, the one given here makes precedence and associativity of expression operators explicit.

A.4 Meaning of Identifiers

Identifiers, or names, refer to a variety of things: functions; tags of structures, unions, and enumerations; members of structures or unions; enumeration constants; typedef names; and objects. An object, sometimes called a variable, is a location in storage, and its interpretation depends on two main attributes: its *storage class* and its *type*. The storage class determines the lifetime of the storage associated with the identified object; the type determines the meaning of the values found in the identified object. A name also has a scope, which is the region of the program in which it is known, and a linkage, which determines whether the same name in another scope refers to the same object or function. Scope and linkage are discussed in [Par.A.11](#).

A.4.1 Storage Class

There are two storage classes: automatic and static. Several keywords, together with the context of an object's declaration, specify its storage class. Automatic objects are local to a block ([Par.9.3](#)), and are discarded on exit from the block. Declarations within a block create automatic objects if no storage class specification is mentioned, or if the `auto` specifier is used. Objects declared `register` are automatic, and are (if possible) stored in fast registers of the machine.

Static objects may be local to a block or external to all blocks, but in either case retain their values across exit from and reentry to functions and blocks. Within a block, including a block that provides the code for a function, static objects are declared with the keyword `static`. The objects declared outside all blocks, at the same level as function definitions, are always static. They may be made local to a particular translation unit by use of the `static` keyword; this gives them *internal linkage*. They become global to an entire program by omitting an explicit storage class, or by using the keyword `extern`; this gives them *external linkage*.

A.4.2 Basic Types

There are several fundamental types. The standard header `<limits.h>` described in [Appendix B](#) defines the largest and smallest values of each type in the local implementation. The numbers given in [Appendix B](#) show the smallest acceptable magnitudes.

Objects declared as characters (`char`) are large enough to store any member of the execution character set. If a genuine character from that set is stored in a `char` object, its value is equivalent to the integer code for the character, and is non-negative. Other quantities may be stored into `char` variables, but the available range of values, and especially whether the value is signed, is implementation-dependent.

Unsigned characters declared `unsigned char` consume the same amount of space as plain characters, but always appear non-negative; explicitly signed characters declared `signed char` likewise take the same space as plain characters.

`unsigned char` type does not appear in the first edition of this book, but is in common use. `signed char` is new.

Besides the `char` types, up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Plain `int` objects have the natural size suggested by the host machine

architecture; the other sizes are provided to meet special needs. Longer integers provide at least as much storage as shorter ones, but the implementation may make plain integers equivalent to either short integers, or long integers. The `int` types all represent signed values unless specified otherwise.

Unsigned integers, declared using the keyword `unsigned`, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation, and thus arithmetic on unsigned quantities can never overflow. The set of non-negative values that can be stored in a signed object is a subset of the values that can be stored in the corresponding unsigned object, and the representation for the overlapping values is the same.

Any of single precision floating point (`float`), double precision floating point (`double`), and extra precision floating point (`long double`) may be synonymous, but the ones later in the list are at least as precise as those before.

`long double` is new. The first edition made `long float` equivalent to `double`; the locution has been withdrawn.

Enumerations are unique types that have integral values; associated with each enumeration is a set of named constants ([Par.A.8.4](#)). Enumerations behave like integers, but it is common for a compiler to issue a warning when an object of a particular enumeration is assigned something other than one of its constants, or an expression of its type.

Because objects of these types can be interpreted as numbers, they will be referred to as *arithmetic* types. Types `char`, and `int` of all sizes, each with or without sign, and also enumeration types, will collectively be called *integral* types. The types `float`, `double`, and `long double` will be called *floating* types.

The `void` type specifies an empty set of values. It is used as the type returned by functions that generate no value.

A.4.3 Derived types

Beside the basic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- arrays* of objects of a given type;
- functions* returning objects of a given type;
- pointers* to objects of a given type;
- structures* containing a sequence of objects of various types;
- unions* capable of containing any of one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

A.4.4 Type Qualifiers

An object's type may have additional qualifiers. Declaring an object `const` announces that its value will not be changed; declaring it `volatile` announces that it has special properties relevant to optimization. Neither qualifier affects the range of values or arithmetic properties of the object. Qualifiers are discussed in [Par.A.8.2](#).

A.5 Objects and Lvalues

An *Object* is a named region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier with suitable type and storage class. There are operators that yield lvalues, if E is an expression of pointer type, then $*E$ is an lvalue expression referring to the object to which E points. The name ``lvalue'' comes from the assignment expression $E_1 = E_2$ in which the left operand E_1 must be an lvalue expression. The

discussion of each operator specifies whether it expects lvalue operands and whether it yields an lvalue.

A.6 Conversions

Some operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. [Par.6.5](#) summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

A.6.1 Integral Promotion

A character, a short integer, or an integer bit-field, all either signed or not, or an object of enumeration type, may be used in an expression wherever an integer may be used. If an `int` can represent all the values of the original type, then the value is converted to `int`; otherwise the value is converted to `unsigned int`. This process is called *integral promotion*.

A.6.2 Integral Conversions

Any integer is converted to a given unsigned type by finding the smallest non-negative value that is congruent to that integer, modulo one more than the largest value that can be represented in the unsigned type. In a two's complement representation, this is equivalent to left-truncation if the bit pattern of the unsigned type is narrower, and to zero-filling unsigned values and sign-extending signed values if the unsigned type is wider.

When any integer is converted to a signed type, the value is unchanged if it can be represented in the new type and is implementation-defined otherwise.

A.6.3 Integer and Floating

When a value of floating type is converted to integral type, the fractional part is discarded; if the resulting value cannot be represented in the integral type, the behavior is undefined. In particular, the result of converting negative floating values to unsigned integral types is not specified.

When a value of integral type is converted to floating, and the value is in the representable range but is not exactly representable, then the result may be either the next higher or next lower representable value. If the result is out of range, the behavior is undefined.

A.6.4 Floating Types

When a less precise floating value is converted to an equally or more precise floating type, the value is unchanged. When a more precise floating value is converted to a less precise floating type, and the value is within representable range, the result may be either the next higher or the next lower representable value. If the result is out of range, the behavior is undefined.

A.6.5 Arithmetic Conversions

Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

- First, if either operand is `long double`, the other is converted to `long double`.
- Otherwise, if either operand is `double`, the other is converted to `double`.
- Otherwise, if either operand is `float`, the other is converted to `float`.
- Otherwise, the integral promotions are performed on both operands; then, if either operand is `unsigned long int`, the other is converted to `unsigned long int`.

- Otherwise, if one operand is `long int` and the other is `unsigned int`, the effect depends on whether a `long int` can represent all values of an `unsigned int`; if so, the `unsigned int` operand is converted to `long int`; if not, both are converted to `unsigned long int`.
- Otherwise, if one operand is `long int`, the other is converted to `long int`.
- Otherwise, if either operand is `unsigned int`, the other is converted to `unsigned int`.
- Otherwise, both operands have type `int`.

There are two changes here. First, arithmetic on `float` operands may be done in single precision, rather than double; the first edition specified that all floating arithmetic was double precision. Second, shorter unsigned types, when combined with a larger signed type, do not propagate the unsigned property to the result type; in the first edition, the unsigned always dominated. The new rules are slightly more complicated, but reduce somewhat the surprises that may occur when an unsigned quantity meets signed. Unexpected results may still occur when an unsigned expression is compared to a signed expression of the same size.

A.6.6 Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case the integral expression is converted as specified in the discussion of the addition operator ([Par.A.7.7](#)).

Two pointers to objects of the same type, in the same array, may be subtracted; the result is converted to an integer as specified in the discussion of the subtraction operator ([Par.A.7.7](#)).

An integral constant expression with value 0, or such an expression cast to type `void *`, may be converted, by a cast, by assignment, or by comparison, to a pointer of any type. This produces a null pointer that is equal to another null pointer of the same type, but unequal to any pointer to a function or object.

Certain other conversions involving pointers are permitted, but have implementation-defined aspects. They must be specified by an explicit type-conversion operator, or cast ([Pars.A.7.5](#) and [A.8.8](#)).

A pointer may be converted to an integral type large enough to hold it; the required size is implementation-dependent. The mapping function is also implementation-dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object may be converted to a pointer to an object whose type requires less or equally strict storage alignment and back again without change; the notion of "alignment" is implementation-dependent, but objects of the `char` types have least strict alignment requirements. As described in [Par.A.6.8](#), a pointer may also be converted to type `void *` and back again without change.

A pointer may be converted to another pointer whose type is the same except for the addition or removal of qualifiers ([Pars.A.4.4](#), [A.8.2](#)) of the object type to which the pointer refers. If qualifiers are added, the new pointer is equivalent to the old except for restrictions implied by the new qualifiers. If qualifiers are removed, operations on the underlying object remain subject to the qualifiers in its actual declaration.

Finally, a pointer to a function may be converted to a pointer to another function type. Calling the function specified by the converted pointer is implementation-dependent; however, if the converted pointer is reconverted to its original type, the result is identical to the original pointer.

A.6.7 Void

The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversion to any non-void type may be applied. Because a `void` expression denotes a nonexistent value, such an expression may be used only where the value is not required, for example as an expression statement ([Par.A.9.2](#)) or as the left operand of a comma operator ([Par.A.7.18](#)).

An expression may be converted to type `void` by a cast. For example, a `void` cast documents the discarding of the value of a function call used as an expression statement.

`void` did not appear in the first edition of this book, but has become common since.

A.6.8 Pointers to Void

Any pointer to an object may be converted to type `void *` without loss of information. If the result is converted back to the original pointer type, the original pointer is recovered. Unlike the pointer-to-pointer conversions discussed in [Par.A.6.6](#), which generally require an explicit cast, pointers may be assigned to and from pointers of type `void *`, and may be compared with them.

This interpretation of `void *` pointers is new; previously, `char *` pointers played the role of generic pointer. The ANSI standard specifically blesses the meeting of `void *` pointers with object pointers in assignments and relationals, while requiring explicit casts for other pointer mixtures.

A.7 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of `+` ([Par.A.7.7](#)) are those expressions defined in [Pars.A.7.1-A.7.6](#). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The grammar given in [Par.13](#) incorporates the precedence and associativity of the operators.

The precedence and associativity of operators is fully specified, but the order of evaluation of expressions is, with certain exceptions, undefined, even if the subexpressions involve side effects. That is, unless the definition of the operator guarantees that its operands are evaluated in a particular order, the implementation is free to evaluate operands in any order, or even to interleave their evaluation. However, each operator combines the values produced by its operands in a way compatible with the parsing of the expression in which it appears.

This rule revokes the previous freedom to reorder expressions with operators that are mathematically commutative and associative, but can fail to be computationally associative. The change affects only floating-point computations near the limits of their accuracy, and situations where overflow is possible.

The handling of overflow, divide check, and other exceptions in expression evaluation is not defined by the language. Most existing implementations of C ignore overflow in evaluation of signed integral expressions and assignments, but this behavior is not guaranteed. Treatment of division by 0, and all floating-point exceptions, varies among implementations; sometimes it is adjustable by a non-standard library function.

A.7.1 Pointer Conversion

If the type of an expression or subexpression is ``array of *T*," for some type *T*, then the value of the expression is a pointer to the first object in the array, and the type of the expression is altered to ``pointer to *T*." This conversion does not take place if the expression is in the operand of the unary `&` operator, or of `++, --, sizeof`, or as the left operand of an assignment operator or the `. .` operator. Similarly, an expression of type ``function returning *T*," except when used as the operand of the `&` operator, is converted to ``pointer to function returning *T*."

A.7.2 Primary Expressions

Primary expressions are identifiers, constants, strings, or expressions in parentheses.

```
primary-expression
  identifier
  constant
  string
  (expression)
```

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. An identifier is an lvalue if it refers to an object ([Par.A.5](#)) and if its type is arithmetic, structure, union, or pointer.

A constant is a primary expression. Its type depends on its form as discussed in [Par.A.2.5](#).

A string literal is a primary expression. Its type is originally ``array of `char`'' (for wide-char strings, ``array of `wchar_t`''), but following the rule given in [Par.A.7.1](#), this is usually modified to ``pointer to `char`'' (`wchar_t`) and the result is a pointer to the first character in the string. The conversion also does not occur in certain initializers; see [Par.A.8.7](#).

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The precedence of parentheses does not affect whether the expression is an lvalue.

A.7.3 Postfix Expressions

The operators in postfix expressions group left to right.

```
postfix-expression:
  primary-expression
  postfix-expression[expression]
  postfix-expression(argument-expression-listopt)
  postfix-expression.identifier
  postfix-expression->identifier
  postfix-expression++
  postfix-expression--

argument-expression-list:
  assignment-expression
  assignment-expression-list , assignment-expression
```

A.7.3.1 Array References

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. One of the two expressions must have type ``pointer to *T*'', where *T* is some type, and the other must have integral type; the type of the subscript expression is *T*. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. See [Par.A.8.6.2](#) for further discussion.

A.7.3.2 Function Calls

A function call is a postfix expression, called the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions ([Par.A7.17](#)), which constitute the arguments to the function. If the postfix expression consists of an identifier for which no declaration exists in the current scope, the identifier is implicitly declared as if the declaration

```
extern int identifier( );
```

had been given in the innermost block containing the function call. The postfix expression (after possible explicit declaration and pointer generation, [Par.A7.1](#)) must be of type ``pointer to function returning T ," for some type T , and the value of the function call has type T .

In the first edition, the type was restricted to ``function," and an explicit `*` operator was required to call through pointers to functions. The ANSI standard blesses the practice of some existing compilers by permitting the same syntax for calls to functions and to functions specified by pointers. The older syntax is still usable.

The term *argument* is used for an expression passed by a function call; the term *parameter* is used for an input object (or its identifier) received by a function definition, or described in a function declaration. The terms ``actual argument (parameter)" and ``formal argument (parameter)" respectively are sometimes used for the same distinction.

In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments. However, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points.

There are two styles in which functions may be declared. In the new style, the types of parameters are explicit and are part of the type of the function; such a declaration is also called a function prototype. In the old style, parameter types are not specified. Function declaration is issued in [Pars.A.8.6.3](#) and [A.10.1](#).

If the function declaration in scope for a call is old-style, then default argument promotion is applied to each argument as follows: integral promotion ([Par.A.6.1](#)) is performed on each argument of integral type, and each `float` argument is converted to `double`. The effect of the call is undefined if the number of arguments disagrees with the number of parameters in the definition of the function, or if the type of an argument after promotion disagrees with that of the corresponding parameter. Type agreement depends on whether the function's definition is new-style or old-style. If it is old-style, then the comparison is between the promoted type of the arguments of the call, and the promoted type of the parameter, if the definition is new-style, the promoted type of the argument must be that of the parameter itself, without promotion.

If the function declaration in scope for a call is new-style, then the arguments are converted, as if by assignment, to the types of the corresponding parameters of the function's prototype. The number of arguments must be the same as the number of explicitly described parameters, unless the declaration's parameter list ends with the ellipsis notation `(, ...)`. In that case, the number of arguments must equal or exceed the number of parameters; trailing arguments beyond the explicitly typed parameters suffer default argument promotion as described in the preceding paragraph. If the definition of the function is old-style, then the type of each parameter in the definition, after the definition parameter's type has undergone argument promotion.

These rules are especially complicated because they must cater to a mixture of old- and new-style functions. Mixtures are to be avoided if possible.

The order of evaluation of arguments is unspecified; take note that various compilers differ. However, the arguments and the function designator are completely evaluated, including all side effects, before the function is entered. Recursive calls to any function are permitted.

A.7.3.3 Structure References

A postfix expression followed by a dot followed by an identifier is a postfix expression. The first operand expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and its

type is the type of the member. The expression is an lvalue if the first expression is an lvalue, and if the type of the second expression is not an array type.

A postfix expression followed by an arrow (built from - and >) followed by an identifier is a postfix expression. The first operand expression must be a pointer to a structure or union, and the identifier must name a member of the structure or union. The result refers to the named member of the structure or union to which the pointer expression points, and the type is the type of the member; the result is an lvalue if the type is not an array type.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in [Par.A.8.3](#).

In the first edition of this book, it was already the rule that a member name in such an expression had to belong to the structure or union mentioned in the postfix expression; however, a note admitted that this rule was not firmly enforced. Recent compilers, and ANSI, do enforce it.

A.7.3.4 Postfix Incrementation

A postfix expression followed by a `++` or `--` operator is a postfix expression. The value of the expression is the value of the operand. After the value is noted, the operand is incremented `++` or decremented `--` by 1. The operand must be an lvalue; see the discussion of additive operators ([Par.A.7.7](#)) and assignment ([Par.A.7.17](#)) for further constraints on the operand and details of the operation. The result is not an lvalue.

A.7.4 Unary Operators

Expressions with unary operators group right-to-left.

unary-expression:

- postfix expression*
- ++unary expression*
- unary expression*
- unary-operator cast-expression*
- sizeof unary-expression*
- sizeof(type-name)*

unary operator: one of

`& * + - ~ !`

A.7.4.1 Prefix Incrementation Operators

A unary expression followed by a `++` or `--` operator is a unary expression. The operand is incremented `++` or decremented `--` by 1. The value of the expression is the value after the incrementation (decrementation). The operand must be an lvalue; see the discussion of additive operators ([Par.A.7.7](#)) and assignment ([Par.A.7.17](#)) for further constraints on the operands and details of the operation. The result is not an lvalue.

A.7.4.2 Address Operator

The unary operator `&` takes the address of its operand. The operand must be an lvalue referring neither to a bit-field nor to an object declared as `register`, or must be of function type. The result is a pointer to the object or function referred to by the lvalue. If the type of the operand is `T`, the type of the result is ``pointer to `T`.''

A.7.4.3 Indirection Operator

The unary `*` operator denotes indirection, and returns the object or function to which its operand points. It is an lvalue if the operand is a pointer to an object of arithmetic, structure, union, or pointer type. If the type of the expression is ``pointer to `T`,'' the type of the result is `T`.

A.7.4.4 Unary Plus Operator

The operand of the unary + operator must have arithmetic type, and the result is the value of the operand. An integral operand undergoes integral promotion. The type of the result is the type of the promoted operand.

The unary + is new with the ANSI standard. It was added for symmetry with the unary -.

A.7.4.5 Unary Minus Operator

The operand of the unary - operator must have arithmetic type, and the result is the negative of its operand. An integral operand undergoes integral promotion. The negative of an unsigned quantity is computed by subtracting the promoted value from the largest value of the promoted type and adding one; but negative zero is zero. The type of the result is the type of the promoted operand.

A.7.4.6 One's Complement Operator

The operand of the ~ operator must have integral type, and the result is the one's complement of its operand. The integral promotions are performed. If the operand is unsigned, the result is computed by subtracting the value from the largest value of the promoted type. If the operand is signed, the result is computed by converting the promoted operand to the corresponding unsigned type, applying ~, and converting back to the signed type. The type of the result is the type of the promoted operand.

A.7.4.7 Logical Negation Operator

The operand of the ! operator must have arithmetic type or be a pointer, and the result is 1 if the value of its operand compares equal to 0, and 0 otherwise. The type of the result is int.

A.7.4.8 Sizeof Operator

The sizeof operator yields the number of bytes required to store an object of the type of its operand. The operand is either an expression, which is not evaluated, or a parenthesized type name. When sizeof is applied to a char, the result is 1; when applied to an array, the result is the total number of bytes in the array. When applied to a structure or union, the result is the number of bytes in the object, including any padding required to make the object tile an array: the size of an array of n elements is n times the size of one element. The operator may not be applied to an operand of function type, or of incomplete type, or to a bit-field. The result is an unsigned integral constant; the particular type is implementation-defined. The standard header <stddef.h> (See [appendix B](#)) defines this type as size_t.

A.7.5 Casts

A unary expression preceded by the parenthesized name of a type causes conversion of the value of the expression to the named type.

cast-expression:
unary expression
(type-name) cast-expression

This construction is called a *cast*. The names are described in [Par.A.8.8](#). The effects of conversions are described in [Par.A.6](#). An expression with a cast is not an lvalue.

A.7.6 Multiplicative Operators

The multiplicative operators *, /, and % group left-to-right.

multiplicative-expression:
*multiplicative-expression * cast-expression*
multiplicative-expression / cast-expression
multiplicative-expression % cast-expression

The operands of `*` and `/` must have arithmetic type; the operands of `%` must have integral type. The usual arithmetic conversions are performed on the operands, and predict the type of the result.

The binary `*` operator denotes multiplication.

The binary `/` operator yields the quotient, and the `%` operator the remainder, of the division of the first operand by the second; if the second operand is 0, the result is undefined. Otherwise, it is always true that $(a/b)*b + a\%b$ is equal to a . If both operands are non-negative, then the remainder is non-negative and smaller than the divisor, if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

A.7.7 Additive Operators

The additive operators `+` and `-` group left-to-right. If the operands have arithmetic type, the usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

The result of the `+` operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is converted to an address offset by multiplying it by the size of the object to which the pointer points. The sum is a pointer of the same type as the original pointer, and points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array. If the sum pointer points outside the bounds of the array, except at the first location beyond the high end, the result is undefined.

The provision for pointers just beyond the end of an array is new. It legitimizes a common idiom for looping over the elements of an array.

The result of the `-` operator is the difference of the operands. A value of any integral type may be subtracted from a pointer, and then the same conversions and conditions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is a signed integral value representing the displacement between the pointed-to objects; pointers to successive objects differ by 1. The type of the result is defined as `ptrdiff_t` in the standard header `<stddef.h>`. The value is undefined unless the pointers point to objects within the same array; however, if `P` points to the last member of an array, then `(P+1)-P` has value 1.

A.7.8 Shift Operators

The shift operators `<<` and `>>` group left-to-right. For both operators, each operand must be integral, and is subject to integral the promotions. The type of the result is that of the promoted left operand. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type.

shift-expression:
additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits; in the absence of overflow, this is equivalent to multiplication by 2^{E2} . The value of `E1>>E2` is `E1` right-shifted `E2`

bit positions. The right shift is equivalent to division by 2^{E2} if `E1` is unsigned or it has a non-negative value; otherwise the result is implementation-defined.

A.7.9 Relational Operators

The relational operators group left-to-right, but this fact is not useful; `a<b<c` is parsed as `(a<b)<c`, and evaluates to either 0 or 1.

```
relational-expression:
  shift-expression
  relational-expression < shift-expression
  relational-expression > shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-expression
```

The operators `<` (less), `>` (greater), `<=` (less or equal) and `>=` (greater or equal) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed on arithmetic operands. Pointers to objects of the same type (ignoring any qualifiers) may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is defined only for parts of the same object; if two pointers point to the same simple object, they compare equal; if the pointers are to members of the same structure, pointers to objects declared later in the structure compare higher; if the pointers refer to members of an array, the comparison is equivalent to comparison of the corresponding subscripts. If `P` points to the last member of an array, then `P+1` compares higher than `P`, even though `P+1` points outside the array. Otherwise, pointer comparison is undefined.

These rules slightly liberalize the restrictions stated in the first edition, by permitting comparison of pointers to different members of a structure or union. They also legalize comparison with a pointer just off the end of an array.

A.7.10 Equality Operators

```
equality-expression:
  relational-expression
  equality-expression == relational-expression
  equality-expression != relational-expression
```

The `==` (equal to) and the `!=` (not equal to) operators are analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.)

The equality operators follow the same rules as the relational operators, but permit additional possibilities: a pointer may be compared to a constant integral expression with value 0, or to a pointer to `void`. See [Par.A.6.6](#).

A.7.11 Bitwise AND Operator

```
AND-expression:
  equality-expression
  AND-expression & equality-expression
```

The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

A.7.12 Bitwise Exclusive OR Operator

```
exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression ^ AND-expression
```

The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

A.7.13 Bitwise Inclusive OR Operator

inclusive-OR-expression:

exclusive-OR-expression

inclusive-OR-expression | *exclusive-OR-expression*

The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of the operands. The operator applies only to integral operands.

A.7.14 Logical AND Operator

logical-AND-expression:

inclusive-OR-expression

logical-AND-expression && *inclusive-OR-expression*

The && operator groups left-to-right. It returns 1 if both its operands compare unequal to zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to 0, the value of the expression is 0. Otherwise, the right operand is evaluated, and if it is equal to 0, the expression's value is 0, otherwise 1.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is int.

A.7.15 Logical OR Operator

logical-OR-expression:

logical-AND-expression

logical-OR-expression || *logical-AND-expression*

The || operator groups left-to-right. It returns 1 if either of its operands compare unequal to zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is unequal to 0, the value of the expression is 1. Otherwise, the right operand is evaluated, and if it is unequal to 0, the expression's value is 1, otherwise 0.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is int.

A.7.16 Conditional Operator

conditional-expression:

logical-OR-expression

logical-OR-expression ? *expression* : *conditional-expression*

The first expression is evaluated, including all side effects; if it compares unequal to 0, the result is the value of the second expression, otherwise that of the third expression. Only one of the second and third operands is evaluated. If the second and third operands are arithmetic, the usual arithmetic conversions are performed to bring them to a common type, and that type is the type of the result. If both are void, or structures or unions of the same type, or pointers to objects of the same type, the result has the common type. If one is a pointer and the other the constant 0, the 0 is converted to the pointer type, and the result has that type. If one is a pointer to void and the other is another pointer, the other pointer is converted to a pointer to void, and that is the type of the result.

In the type comparison for pointers, any type qualifiers ([Par.A.8.2](#)) in the type to which the pointer points are insignificant, but the result type inherits qualifiers from both arms of the conditional.

A.7.17 Assignment Expressions

There are several assignment operators; all group right-to-left.

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of

= *= /= %= += -= <<= >>= &= ^= |=

All require an lvalue as left operand, and the lvalue must be modifiable: it must not be an array, and must not have an incomplete type, or be a function. Also, its type must not be qualified with `const`; if it is a structure or union, it must not have any member or, recursively, submember qualified with `const`. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place.

In the simple assignment with `=`, the value of the expression replaces that of the object referred to by the lvalue. One of the following must be true: both operands have arithmetic type, in which case the right operand is converted to the type of the left by the assignment; or both operands are structures or unions of the same type; or one operand is a pointer and the other is a pointer to `void`, or the left operand is a pointer and the right operand is a constant expression with value 0; or both operands are pointers to functions or objects whose types are the same except for the possible absence of `const` or `volatile` in the right operand.

An expression of the form `E1 op= E2` is equivalent to `E1 = E1 op (E2)` except that `E1` is evaluated only once.

A.7.18 Comma Operator

expression:
assignment-expression
expression , assignment-expression

A pair of expressions separated by a comma is evaluated left-to-right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. All side effects from the evaluation of the left-operand are completed before beginning the evaluation of the right operand. In contexts where comma is given a special meaning, for example in lists of function arguments ([Par.A.7.3.2](#)) and lists of initializers ([Par.A.8.7](#)), the required syntactic unit is an assignment expression, so the comma operator appears only in a parenthetical grouping, for example,

`f(a, (t=3, t+2), c)`

has three arguments, the second of which has the value 5.

A.7.19 Constant Expressions

Syntactically, a constant expression is an expression restricted to a subset of operators:

constant-expression:
conditional-expression

Expressions that evaluate to a constant are required in several contexts: after `case`, as array bounds and bit-field lengths, as the value of an enumeration constant, in initializers, and in certain preprocessor expressions.

Constant expressions may not contain assignments, increment or decrement operators, function calls, or comma operators; except in an operand of `sizeof`. If the constant expression is required to be integral, its operands must consist of integer, enumeration, character, and floating constants; casts must specify an integral type, and any floating constants must be cast to integer. This necessarily rules out arrays, indirection, address-of, and structure member operations. (However, any operand is permitted for `sizeof`.)

More latitude is permitted for the constant expressions of initializers; the operands may be any type of constant, and the unary `&` operator may be applied to external or static objects, and to external and static arrays subscripted with a constant expression. The unary `&` operator can also be applied implicitly by appearance of unsubscripted arrays and functions. Initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Less latitude is allowed for the integral constant expressions after `#if`; `sizeof` expressions, enumeration constants, and casts are not permitted. See [Par.A.12.5](#).

A.8 Declarations

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations that reserve storage are called *definitions*. Declarations have the form

declaration:
declaration-specifiers init-declarator-list_{opt};

The declarators in the init-declarator list contain the identifiers being declared; the declaration-specifiers consist of a sequence of type and storage class specifiers.

declaration-specifiers:
storage-class-specifier declaration-specifiers_{opt}
type-specifier declaration-specifiers_{opt}
type-qualifier declaration-specifiers_{opt}

init-declarator-list:
init-declarator
init-declarator-list , init-declarator

init-declarator:
declarator
declarator = initializer

Declarators will be discussed later ([Par.A.8.5](#)); they contain the names being declared. A declaration must have at least one declarator, or its type specifier must declare a structure tag, a union tag, or the members of an enumeration; empty declarations are not permitted.

A.8.1 Storage Class Specifiers

The storage class specifiers are:

storage-class specifier:
auto
register

```
static
extern
typedef
```

The meaning of the storage classes were discussed in [Par.A.4.4](#).

The `auto` and `register` specifiers give the declared objects automatic storage class, and may be used only within functions. Such declarations also serve as definitions and cause storage to be reserved. A `register` declaration is equivalent to an `auto` declaration, but hints that the declared objects will be accessed frequently. Only a few objects are actually placed into registers, and only certain types are eligible; the restrictions are implementation-dependent. However, if an object is declared `register`, the unary `&` operator may not be applied to it, explicitly or implicitly.

The rule that it is illegal to calculate the address of an object declared `register`, but actually taken to be `auto`, is new.

The `static` specifier gives the declared objects static storage class, and may be used either inside or outside functions. Inside a function, this specifier causes storage to be allocated, and serves as a definition; for its effect outside a function, see [Par.A.11.2](#).

A declaration with `extern`, used inside a function, specifies that the storage for the declared objects is defined elsewhere; for its effects outside a function, see [Par.A.11.2](#).

The `typedef` specifier does not reserve storage and is called a storage class specifier only for syntactic convenience; it is discussed in [Par.A.8.9](#).

At most one storage class specifier may be given in a declaration. If none is given, these rules are used: objects declared inside a function are taken to be `auto`; functions declared within a function are taken to be `extern`; objects and functions declared outside a function are taken to be `static`, with external linkage. See [Pars. A.10-A.11](#).

A.8.2 Type Specifiers

The type-specifiers are

type specifier:

```
void
char
short
int
long
float
double
signed
unsigned
struct-or-union-specifier
enum-specifier
typedef-name
```

At most one of the words `long` or `short` may be specified together with `int`; the meaning is the same if `int` is not mentioned. The word `long` may be specified together with `double`. At most one of `signed` or `unsigned` may be specified together with `int` or any of its `short` or `long` varieties, or with `char`. Either may appear alone in which case `int` is understood. The `signed` specifier is useful for forcing `char` objects to carry a sign; it is permissible but redundant with other integral types.

Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be `int`.

Types may also be qualified, to indicate special properties of the objects being declared.

type-qualifier:

```
const
volatile
```

Type qualifiers may appear with any type specifier. A `const` object may be initialized, but not thereafter assigned to. There are no implementation-dependent semantics for `volatile` objects.

The `const` and `volatile` properties are new with the ANSI standard. The purpose of `const` is to announce objects that may be placed in read-only memory, and perhaps to increase opportunities for optimization. The purpose of `volatile` is to force an implementation to suppress optimization that could otherwise occur. For example, for a machine with memory-mapped input/output, a pointer to a device register might be declared as a pointer to `volatile`, in order to prevent the compiler from removing apparently redundant references through the pointer. Except that it should diagnose explicit attempts to change `const` objects, a compiler may ignore these qualifiers.

A.8.3 Structure and Union Declarations

A structure is an object consisting of a sequence of named members of various types. A union is an object that contains, at different times, any of several members of various types. Structure and union specifiers have the same form.

struct-or-union-specifier:

```
struct-or-union identifieropt{ struct-declaration-list }
struct-or-union identifier
```

struct-or-union:

```
struct
union
```

A `struct-declaration-list` is a sequence of declarations for the members of the structure or union:

struct-declaration-list:

```
struct declaration
struct-declaration-list struct declaration
```

struct-declaration: *specifier-qualifier-list struct-declarator-list;*

specifier-qualifier-list:

```
type-specifier specifier-qualifier-listopt
type-qualifier specifier-qualifier-listopt
```

struct-declarator-list:

```
struct-declarator
struct-declarator-list , struct-declarator
```

Usually, a `struct-declarator` is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *bit-field*; its length is set off from the declarator for the field name by a colon.

struct-declarator:

```
declarator   declaratoropt : constant-expression
```

A type specifier of the form

struct-or-union identifier { struct-declaration-list }

declares the identifier to be the *tag* of the structure or union specified by the list. A subsequent declaration in the same or an inner scope may refer to the same type by using the tag in a specifier without the list:

struct-or-union identifier

If a specifier with a tag but without a list appears when the tag is not declared, an *incomplete type* is specified. Objects with an incomplete structure or union type may be mentioned in contexts where their size is not needed, for example in declarations (not definitions), for specifying a pointer, or for creating a `typedef`, but not otherwise. The type becomes complete on occurrence of a subsequent specifier with that tag, and containing a declaration list. Even in specifiers with a list, the structure or union type being declared is incomplete within the list, and becomes complete only at the } terminating the specifier.

A structure may not contain a member of incomplete type. Therefore, it is impossible to declare a structure or union containing an instance of itself. However, besides giving a name to the structure or union type, tags allow definition of self-referential structures; a structure or union may contain a pointer to an instance of itself, because pointers to incomplete types may be declared.

A very special rule applies to declarations of the form

struct-or-union identifier;

that declare a structure or union, but have no declaration list and no declarators. Even if the identifier is a structure or union tag already declared in an outer scope ([Par.A.11.1](#)), this declaration makes the identifier the tag of a new, incompletely-typed structure or union in the current scope.

This recondite is new with ANSI. It is intended to deal with mutually-recursive structures declared in an inner scope, but whose tags might already be declared in the outer scope.

A structure or union specifier with a list but no tag creates a unique type; it can be referred to directly only in the declaration of which it is a part.

The names of members and tags do not conflict with each other or with ordinary variables. A member name may not appear twice in the same structure or union, but the same member name may be used in different structures or unions.

In the first edition of this book, the names of structure and union members were not associated with their parent. However, this association became common in compilers well before the ANSI standard.

A non-field member of a structure or union may have any object type. A field member (which need not have a declarator and thus may be unnamed) has type `int`, `unsigned int`, or `signed int`, and is interpreted as an object of integral type of the specified length in bits; whether an `int` field is treated as `signed` is implementation-dependent. Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction. When a field following another field will not fit into a partially-filled storage unit, it may be split between units, or the unit may be padded. An unnamed field with width 0 forces this padding, so that the next field will begin at the edge of the next allocation unit.

The ANSI standard makes fields even more implementation-dependent than did the first edition. It is advisable to read the language rules for storing bit-fields as "implementation-dependent" without qualification. Structures with bit-fields may be used as a portable way of attempting to reduce the storage required for a structure (with the probable cost of increasing the instruction space, and time,

needed to access the fields), or as a non-portable way to describe a storage layout known at the bit-level. In the second case, it is necessary to understand the rules of the local implementation.

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time. If a pointer to a union is cast to the type of a pointer to a member, the result refers to that member.

A simple example of a structure declaration is

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
}
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be a structure of the given sort, and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the `count` field of the structure to which `sp` points;

```
s.left
```

refers to the left subtree pointer of the structure `s`, and

```
s.right->tword[0]
```

refers to the first character of the `tword` member of the right subtree of `s`.

In general, a member of a union may not be inspected unless the value of the union has been assigned using the same member. However, one special guarantee simplifies the use of unions: if a union contains several structures that share a common initial sequence, and the union currently contains one of these structures, it is permitted to refer to the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
```

```
... sin(u.nf.floatnode) ...
```

A.8.4 Enumerations

Enumerations are unique types with values ranging over a set of named constants called enumerators. The form of an enumeration specifier borrows from that of structures and unions.

enum-specifier:

```
enum identifieropt { enumerator-list }
```

```
enum identifier
```

enumerator-list:

```
enumerator
```

```
enumerator-list , enumerator
```

enumerator:

```
identifier
```

```
identifier = constant-expression
```

The identifiers in an enumerator list are declared as constants of type `int`, and may appear wherever constants are required. If no enumerations with `=` appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with `=` gives the associated identifier the value specified; subsequent identifiers continue the progression from the assigned value.

Enumerator names in the same scope must all be distinct from each other and from ordinary variable names, but the values need not be distinct.

The role of the identifier in the enum-specifier is analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. The rules for enum-specifiers with and without tags and lists are the same as those for structure or union specifiers, except that incomplete enumeration types do not exist; the tag of an enum-specifier without an enumerator list must refer to an in-scope specifier with a list.

Enumerations are new since the first edition of this book, but have been part of the language for some years.

A.8.5 Declarators

Declarators have the syntax:

declarator:

```
pointeropt direct-declarator
```

direct-declarator:

```
identifier
```

```
(declarator)
```

```
direct-declarator [ constant-expressionopt ]
```

```
direct-declarator ( parameter-type-list )
```

```
direct-declarator ( identifier-listopt )
```

pointer:

```
* type-qualifier-listopt
```

```
* type-qualifier-listopt pointer
```

type-qualifier-list:

```
type-qualifier
```

```
type-qualifier-list type-qualifier
```

The structure of declarators resembles that of indirection, function, and array expressions; the grouping is the same.

A.8.6 Meaning of Declarators

A list of declarators appears after a sequence of type and storage class specifiers. Each declarator declares a unique main identifier, the one that appears as the first alternative of the production for *direct-declarator*. The storage class specifiers apply directly to this identifier, but its type depends on the form of its declarator. A declarator is read as an assertion that when its identifier appears in an expression of the same form as the declarator, it yields an object of the specified type.

Considering only the type parts of the declaration specifiers (Par. A.8.2) and a particular declarator, a declaration has the form `` $T\ D$," where T is a type and D is a declarator. The type attributed to the identifier in the various forms of declarator is described inductively using this notation.

In a declaration $T\ D$ where D is an unadorned identifier, the type of the identifier is T .

In a declaration $T\ D$ where D has the form

(D_1)

then the type of the identifier in D_1 is the same as that of D . The parentheses do not alter the type, but may change the binding of complex declarators.

A.8.6.1 Pointer Declarators

In a declaration $T\ D$ where D has the form

* *type-qualifier-list_{opt}* D_1

and the type of the identifier in the declaration $T\ D_1$ is ``*type-modifier T*," the type of the identifier of D is ``*type-modifier type-qualifier-list* pointer to T ." Qualifiers following * apply to pointer itself, rather than to the object to which the pointer points.

For example, consider the declaration

```
int *ap[ ];
```

Here, $ap[]$ plays the role of D_1 ; a declaration `` $int\ ap[]$ '' (below) would give ap the type ``array of int," the type-qualifier list is empty, and the type-modifier is ``array of." Hence the actual declaration gives ap the type ``array to pointers to int."

As other examples, the declarations

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

declare an integer i and a pointer to an integer pi . The value of the constant pointer cpi may not be changed; it will always point to the same location, although the value to which it refers may be altered. The integer ci is constant, and may not be changed (though it may be initialized, as here.) The type of pci is ``pointer to const int," and pci itself may be changed to point to another place, but the value to which it points may not be altered by assigning through pci .

A.8.6.2 Array Declarators

In a declaration $T\ D$ where D has the form

D_1 [*constant-expression_{opt}*]

and the type of the identifier in the declaration $T \ D_1$ is ``*type-modifier* T ," the type of the identifier of D is ``*type-modifier* array of T ." If the constant-expression is present, it must have integral type, and value greater than 0. If the constant expression specifying the bound is missing, the array has an incomplete type.

An array may be constructed from an arithmetic type, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array). Any type from which an array is constructed must be complete; it must not be an array of structure of incomplete type. This implies that for a multi-dimensional array, only the first dimension may be missing. The type of an object of incomplete array type is completed by another, complete, declaration for the object ([Par.A.10.2](#)), or by initializing it ([Par.A.8.7](#)). For example,

```
float fa[17], *afp[17];
```

declares an array of float numbers and an array of pointers to float numbers. Also,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank $3 \times 5 \times 7$. In complete detail, $x3d$ is an array of three items: each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions $x3d$, $x3d[i]$, $x3d[i][j]$, $x3d[i][j][k]$ may reasonably appear in an expression. The first three have type ``array," the last has type int. More specifically, $x3d[i][j]$ is an array of 7 integers, and $x3d[i]$ is an array of 5 arrays of 7 integers.

The array subscripting operation is defined so that $E1[E2]$ is identical to $*(E1+E2)$. Therefore, despite its asymmetric appearance, subscripting is a commutative operation. Because of the conversion rules that apply to + and to arrays ([Pars.A6.6](#), [A7.1](#), [A7.7](#)), if $E1$ is an array and $E2$ an integer, then $E1[E2]$ refers to the $E2$ -th member of $E1$.

In the example, $x3d[i][j][k]$ is equivalent to $*(x3d[i][j] + k)$. The first subexpression $x3d[i][j]$ is converted by [Par.A.7.1](#) to type ``pointer to array of integers," by [Par.A.7.7](#), the addition involves multiplication by the size of an integer. It follows from the rules that arrays are stored by rows (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array, but plays no other part in subscript calculations.

A.8.6.3 Function Declarators

In a new-style function declaration $T \ D$ where D has the form

D_1 (*parameter-type-list*)

and the type of the identifier in the declaration $T \ D_1$ is ``*type-modifier* T ," the type of the identifier of D is ``*type-modifier* function with arguments *parameter-type-list* returning T ."

The syntax of the parameters is

parameter-type-list:

parameter-list

parameter-list , ...

parameter-list:

parameter-declaration

parameter-list , *parameter-declaration*

parameter-declaration:

declaration-specifiers declarator

*declaration-specifiers abstract-declarator*_{opt}

In the new-style declaration, the parameter list specifies the types of the parameters. As a special case, the declarator for a new-style function with no parameters has a parameter list consisting solely of the keyword `void`. If the parameter list ends with an ellipsis ```, ...`'', then the function may accept more arguments than the number of parameters explicitly described, see [Par.A.7.3.2](#).

The types of parameters that are arrays or functions are altered to pointers, in accordance with the rules for parameter conversions; see [Par.A.10.1](#). The only storage class specifier permitted in a parameter's declaration is `register`, and this specifier is ignored unless the function declarator heads a function definition. Similarly, if the declarators in the parameter declarations contain identifiers and the function declarator does not head a function definition, the identifiers go out of scope immediately. Abstract declarators, which do not mention the identifiers, are discussed in [Par.A.8.8](#).

In an old-style function declaration `T D` where `D` has the form

`D1(identifier-listopt)`

and the type of the identifier in the declaration `T D1` is ``*type-modifier T*,'' the type of the identifier of `D` is ``*type-modifier* function of unspecified arguments returning `T`.'' The parameters (if present) have the form

identifier-list:
identifier
identifier-list , identifier

In the old-style declarator, the identifier list must be absent unless the declarator is used in the head of a function definition ([Par.A.10.1](#)). No information about the types of the parameters is supplied by the declaration.

For example, the declaration

```
int f(), *fpi(), (*pfi)();
```

declares a function `f` returning an integer, a function `fpi` returning a pointer to an integer, and a pointer `pfi` to a function returning an integer. In none of these are the parameter types specified; they are old-style.

In the new-style declaration

```
int strcpy(char *dest, const char *source), rand(void);
```

`strcpy` is a function returning `int`, with two arguments, the first a character pointer, and the second a pointer to constant characters. The parameter names are effectively comments. The second function `rand` takes no arguments and returns `int`.

Function declarators with parameter prototypes are, by far, the most important language change introduced by the ANSI standard. They offer an advantage over the ``old-style'' declarators of the first edition by providing error-detection and coercion of arguments across function calls, but at a cost: turmoil and confusion during their introduction, and the necessity of accomodating both forms. Some syntactic ugliness was required for the sake of compatibility, namely `void` as an explicit marker of new-style functions without parameters.

The ellipsis notation ```, ...`'' for variadic functions is also new, and, together with the macros in the standard header `<stdarg.h>`, formalizes a mechanism that was officially forbidden but unofficially condoned in the first edition.

These notations were adapted from the C++ language.

A.8.7 Initialization

When an object is declared, its init-declarator may specify an initial value for the identifier being declared. The initializer is preceded by `=`, and is either an expression, or a list of initializers nested in braces. A list may end with a comma, a nicety for neat formatting.

```
initializer:
  assignment-expression
  { initializer-list }
  { initializer-list , }
```



```
initializer-list:
  initializer
  initializer-list , initializer
```

All the expressions in the initializer for a static object or array must be constant expressions as described in [Par.A.7.19](#). The expressions in the initializer for an `auto` or `register` object or array must likewise be constant expressions if the initializer is a brace-enclosed list. However, if the initializer for an automatic object is a single expression, it need not be a constant expression, but must merely have appropriate type for assignment to the object.

The first edition did not countenance initialization of automatic structures, unions, or arrays. The ANSI standard allows it, but only by constant constructions unless the initializer can be expressed by a simple expression.

A static object not explicitly initialized is initialized as if it (or its members) were assigned the constant 0. The initial value of an automatic object not explicitly initialized is undefined.

The initializer for a pointer or an object of arithmetic type is a single expression, perhaps in braces. The expression is assigned to the object.

The initializer for a structure is either an expression of the same type, or a brace-enclosed list of initializers for its members in order. Unnamed bit-field members are ignored, and are not initialized. If there are fewer initializers in the list than members of the structure, the trailing members are initialized with 0. There may not be more initializers than members. Unnamed bit-field members are ignored, and are not initialized.

The initializer for an array is a brace-enclosed list of initializers for its members. If the array has unknown size, the number of initializers determines the size of the array, and its type becomes complete. If the array has fixed size, the number of initializers may not exceed the number of members of the array; if there are fewer, the trailing members are initialized with 0.

As a special case, a character array may be initialized by a string literal; successive characters of the string initialize successive members of the array. Similarly, a wide character literal ([Par.A.2.6](#)) may initialize an array of type `wchar_t`. If the array has unknown size, the number of characters in the string, including the terminating null character, determines its size; if its size is fixed, the number of characters in the string, not counting the terminating null character, must not exceed the size of the array.

The initializer for a union is either a single expression of the same type, or a brace-enclosed initializer for the first member of the union.

The first edition did not allow initialization of unions. The ``first-member'' rule is clumsy, but is hard to generalize without new syntax. Besides allowing unions to be explicitly initialized in at least a primitive way, this ANSI rule makes definite the semantics of static unions not explicitly initialized.

An *aggregate* is a structure or array. If an aggregate contains members of aggregate type, the initialization rules apply recursively. Braces may be elided in the initialization as follows: if the initializer for an aggregate's member that itself is an aggregate begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the subaggregate; it is erroneous for there to be more initializers than members. If, however, the initializer for a

subaggregate does not begin with a left brace, then only enough elements from the list are taken into account for the members of the subaggregate; any remaining members are left to initialize the next member of the aggregate of which the subaggregate is a part.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a 1-dimensional array with three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3 and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early, and therefore the elements of `y[3]` are initialized with 0. Precisely the same effect could have been achieved by

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace, but that for `y[0]` does not; therefore three elements from the list are used. Likewise the next three are taken successively for `y[1]` and for `y[2]`. Also,

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string; its size includes the terminating null character.

A.8.8 Type names

In several contexts (to specify type conversions explicitly with a cast, to declare parameter types in function declarators, and as argument of `sizeof`) it is necessary to supply the name of a data type. This is accomplished using a *type name*, which is syntactically a declaration for an object of that type omitting the name of the object.

type-name:
specifier-qualifier-list abstract-declarator_{opt}

abstract-declarator:
pointer
pointer_{opt} direct-abstract-declarator

direct-abstract-declarator:
(*abstract-declarator*)
direct-abstract-declarator_{opt} [constant-expression_{opt}]
direct-abstract-declarator_{opt} (parameter-type-list_{opt})

It is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

name respectively the types ``integer," ``pointer to integer," ``array of 3 pointers to integers," ``pointer to an unspecified number of integers," ``function of unspecified parameters returning pointer to integer," and ``array, of unspecified size, of pointers to functions with no parameters each returning an integer."

A.8.9 Typedef

Declarations whose storage class specifier is `typedef` do not declare objects; instead they define identifiers that name types. These identifiers are called `typedef` names.

typedef-name:
identifier

A `typedef` declaration attributes a type to each name among its declarators in the usual way (see [Par.A.8.6](#)). Thereafter, each such `typedef` name is syntactically equivalent to a type specifier keyword for the associated type.

For example, after

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

the constructions

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

are legal declarations. The type of `b` is `long`, that of `bp` is ``pointer to `long`," and that of `z` is the specified structure; `zp` is a pointer to such a structure.

`typedef` does not introduce new types, only synonyms for types that could be specified in another way. In the example, `b` has the same type as any `long` object.

`Typedef` names may be redeclared in an inner scope, but a non-empty set of type specifiers must be given. For example,

```
extern Blockno;
does not redeclare Blockno, but
```

```
extern int Blockno;
does.
```

A.8.10 Type Equivalence

Two type specifier lists are equivalent if they contain the same set of type specifiers, taking into account that some specifiers can be implied by others (for example, `long` alone implies `long int`). Structures, unions, and enumerations with different tags are distinct, and a tagless union, structure, or enumeration specifies a unique type.

Two types are the same if their abstract declarators ([Par.A.8.8](#)), after expanding any `typedef` types, and deleting any function parameter specifiers, are the same up to the equivalence of type specifier lists. Array sizes and function parameter types are significant.

A.9 Statements

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups.

statement:

labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

A.9.1 Labeled Statements

Statements may carry label prefixes.

labeled-statement:

identifier : *statement*
`case` *constant-expression* : *statement*
`default` : *statement*

A label consisting of an identifier declares the identifier. The only use of an identifier label is as a target of `goto`. The scope of the identifier is the current function. Because labels have their own name space, they do not interfere with other identifiers and cannot be redeclared. See [Par.A.11.1](#).

Case labels and default labels are used with the `switch` statement ([Par.A.9.4](#)). The constant expression of `case` must have integral type.

Labels themselves do not alter the flow of control.

A.9.2 Expression Statement

Most statements are expression statements, which have the form

expression-statement:
 $expression_{opt};$

Most expression statements are assignments or function calls. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement to place a label.

A.9.3 Compound Statement

So that several statements can be used where one is expected, the compound statement (also called ``block") is provided. The body of a function definition is a compound statement.

compound-statement:
 $\{ declaration-list_{opt} statement-list_{opt} \}$

declaration-list:
 $declaration$
 $declaration-list\ declaration$

statement-list:
statement
statement-list statement

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended within the block (see [Par.A.11.1](#)), after which it resumes its force. An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space ([Par.A.11](#)); identifiers in different name spaces are treated as distinct.

Initialization of automatic objects is performed each time the block is entered at the top, and proceeds in the order of the declarators. If a jump into the block is executed, these initializations are not performed. Initialization of `static` objects are performed only once, before the program begins execution.

A.9.4 Selection Statements

Selection statements choose one of several flows of control.

selection-statement:
`if (expression) statement`
`if (expression) statement else statement`
`switch (expression) statement`

In both forms of the `if` statement, the expression, which must have arithmetic or pointer type, is evaluated, including all side effects, and if it compares unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is 0. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else-less if` at the same block nesting level.

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression, which must have integral type. The substatement controlled by a `switch` is typically compound. Any statement within the substatement may be labeled with one or more `case` labels ([Par.A.9.1](#)). The controlling expression undergoes integral promotion ([Par.A.6.1](#)), and the case constants are converted to the promoted type. No two of these case constants associated with the same switch may have the same value after conversion. There may also be at most one `default` label associated with a switch. Switches may be nested; a `case` or `default` label is associated with the smallest switch that contains it.

When the `switch` statement is executed, its expression is evaluated, including all side effects, and compared with each case constant. If one of the case constants is equal to the value of the expression, control passes to the statement of the matched `case` label. If no case constant matches the expression, and if there is a `default` label, control passes to the labeled statement. If no case matches, and if there is no `default`, then none of the substatements of the `switch` is executed.

In the first edition of this book, the controlling expression of `switch`, and the case constants, were required to have `int` type.

A.9.5 Iteration Statements

Iteration statements specify looping.

iteration-statement:
`while (expression) statement`
`do statement while (expression);`
`for (expressionopt; expressionopt; expressionopt) statement`

In the `while` and `do` statements, the substatement is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must have arithmetic or pointer type. With `while`, the test, including all side effects from the expression, occurs before each execution of the statement; with `do`, the test follows each iteration.

In the `for` statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have arithmetic or pointer type; it is evaluated before each iteration, and if it becomes equal to 0, the `for` is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. If the substatement does not contain `continue`, a statement

`for (expression1 : expression2 : expression3) statement`

is equivalent to

```
expression1;
while (expression2) {
    statement
    expression3;
}
```

Any of the three expressions may be dropped. A missing second expression makes the implied test equivalent to testing a non-zero element.

A.9.6 Jump statements

Jump statements transfer control unconditionally.

jump-statement:

```
goto identifier;
continue;
break;
return expressionopt;
```

In the `goto` statement, the identifier must be a label ([Par.A.9.1](#)) located in the current function. Control transfers to the labeled statement.

A `continue` statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement. More precisely, within each of the statements

```
while (...) {           do {                   for (...) {
    ...
    contin: ;           contin: ;             contin: ;
} } while (...) ;       } while (...) ;       }
```

a `continue` not contained in a smaller iteration statement is the same as `goto contin`.

A `break` statement may appear only in an iteration statement or a `switch` statement, and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the `return` statement. When `return` is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears.

Flowing off the end of a function is equivalent to a `return` with no expression. In either case, the returned value is undefined.

A.10 External Declarations

The unit of input provided to the C compiler is called a translation unit; it consists of a sequence of external declarations, which are either declarations or function definitions.

translation-unit:

external-declaration

translation-unit *external-declaration*

external-declaration:

function-definition

declaration

The scope of external declarations persists to the end of the translation unit in which they are declared, just as the effect of declarations within the blocks persists to the end of the block. The syntax of external declarations is the same as that of all declarations, except that only at this level may the code for functions be given.

A.10.1 Function Definitions

Function definitions have the form

function-definition:

declaration-specifiers_{opt} *declarator* *declaration-list_{opt}* *compound-statement*

The only storage-class specifiers allowed among the declaration specifiers are `extern` or `static`; see [Par.A.11.2](#) for the distinction between them.

A function may return an arithmetic type, a structure, a union, a pointer, or `void`, but not a function or an array. The declarator in a function declaration must specify explicitly that the declared identifier has function type; that is, it must contain one of the forms (see [Par.A.8.6.3](#)).

direct-declarator (*parameter-type-list*)

direct-declarator (*identifier-list_{opt}*)

where the direct-declarator is an identifier or a parenthesized identifier. In particular, it must not achieve function type by means of a `typedef`.

In the first form, the definition is a new-style function, and its parameters, together with their types, are declared in its parameter type list; the declaration-list following the function's declarator must be absent. Unless the parameter type list consists solely of `void`, showing that the function takes no parameters, each declarator in the parameter type list must contain an identifier. If the parameter type list ends with `` , . . . '' then the function may be called with more arguments than parameters; the `va_arg` macro mechanism defined in the standard header `<stdarg.h>` and described in [Appendix B](#) must be used to refer to the extra arguments. Variadic functions must have at least one named parameter.

In the second form, the definition is old-style: the identifier list names the parameters, while the declaration list attributes types to them. If no declaration is given for a parameter, its type is taken to be `int`. The declaration list must declare only parameters named in the list, initialization is not permitted, and the only storage-class specifier possible is `register`.

In both styles of function definition, the parameters are understood to be declared just after the beginning of the compound statement constituting the function's body, and thus the same identifiers must not be redeclared there (although they may, like other identifiers, be redeclared in inner blocks). If a parameter is declared to have type ``array of *type*,'' the declaration is adjusted to read ``pointer to *type*''; similarly, if a parameter is declared to have type ``function

returning *type*," the declaration is adjusted to read ``pointer to function returning *type*.'' During the call to a function, the arguments are converted as necessary and assigned to the parameters; see [Par.A.7.3.2](#).

New-style function definitions are new with the ANSI standard. There is also a small change in the details of promotion; the first edition specified that the declarations of `float` parameters were adjusted to read `double`. The difference becomes noticeable when a pointer to a parameter is generated within a function.

A complete example of a new-style function definition is

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here `int` is the declaration specifier; `max(int a, int b, int c)` is the function's declarator, and `{ ... }` is the block giving the code for the function. The corresponding old-style definition would be

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
```

where now `int max(a, b, c)` is the declarator, and `int a, b, c;` is the declaration list for the parameters.

A.10.2 External Declarations

External declarations specify the characteristics of objects, functions and other identifiers. The term ``external'' refers to their location outside functions, and is not directly connected with the `extern` keyword; the storage class for an externally-declared object may be left empty, or it may be specified as `extern` or `static`.

Several external declarations for the same identifier may exist within the same translation unit if they agree in type and linkage, and if there is at most one definition for the identifier.

Two declarations for an object or function are deemed to agree in type under the rule discussed in [Par.A.8.10](#). In addition, if the declarations differ because one type is an incomplete structure, union, or enumeration type ([Par.A.8.3](#)) and the other is the corresponding completed type with the same tag, the types are taken to agree. Moreover, if one type is an incomplete array type ([Par.A.8.6.2](#)) and the other is a completed array type, the types, if otherwise identical, are also taken to agree. Finally, if one type specifies an old-style function, and the other an otherwise identical new-style function, with parameter declarations, the types are taken to agree.

If the first external declarator for a function or object includes the `static` specifier, the identifier has *internal linkage*; otherwise it has *external linkage*. Linkage is discussed in [Par.11.2](#).

An external declaration for an object is a definition if it has an initializer. An external object declaration that does not have an initializer, and does not contain the `extern` specifier, is a *tentative definition*. If a definition for an object appears in a translation unit, any tentative definitions are treated merely as redundant declarations. If no definition for the object appears in the translation unit, all its tentative definitions become a single definition with initializer 0.

Each object must have exactly one definition. For objects with internal linkage, this rule applies separately to each translation unit, because internally-linked objects are unique to a translation unit. For objects with external linkage, it applies to the entire program.

Although the one-definition rule is formulated somewhat differently in the first edition of this book, it is in effect identical to the one stated here. Some implementations relax it by generalizing the notion of tentative definition. In the alternate formulation, which is usual in UNIX systems and recognized as a common extension by the Standard, all the tentative definitions for an externally linked object, throughout all the translation units of the program, are considered together instead of in each translation unit separately. If a definition occurs somewhere in the program, then the tentative definitions become merely declarations, but if no definition appears, then all its tentative definitions become a definition with initializer 0.

A.11 Scope and Linkage

A program need not all be compiled at one time: the source text may be kept in several files containing translation units, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, the *lexical scope* of an identifier which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects and functions with external linkage, which determines the connections between identifiers in separately compiled translation units.

A.11.1 Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These classes are: objects, functions, `typedef` names, and `enum` constants; labels; tags of structures or unions, and enumerations; and members of each structure or union individually.

These rules differ in several ways from those described in the first edition of this manual. Labels did not previously have their own name space; tags of structures and unions each had a separate space, and in some implementations enumerations tags did as well; putting different kinds of tags into the same space is a new restriction. The most important departure from the first edition is that each structure or union creates a separate name space for its members, so that the same name may appear in several different structures. This rule has been common practice for several years.

The lexical scope of an object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a label is the whole of the function in which it appears. The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of a translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

A.11.2 Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit. All

declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

As discussed in [Par.A.10.2](#), the first external declaration for an identifier gives the identifier internal linkage if the `static` specifier is used, external linkage otherwise. If a declaration for an identifier within a block does not include the `extern` specifier, then the identifier has no linkage and is unique to the function. If it does include `extern`, and an external declaration for is active in the scope surrounding the block, then the identifier has the same linkage as the external declaration, and refers to the same object or function; but if no external declaration is visible, its linkage is external.

A.12 Preprocessing

A preprocessor performs macro substitution, conditional compilation, and inclusion of named files. Lines beginning with `#`, perhaps preceded by white space, communicate with this preprocessor. The syntax of these lines is independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the translation unit. Line boundaries are significant; each line is analyzed individually (but see [Par.A.12.2](#) for how to adjoin lines). To the preprocessor, a token is any language token, or a character sequence giving a file name as in the `#include` directive ([Par.A.12.4](#)); in addition, any character not otherwise defined is taken as a token. However, the effect of white spaces other than space and horizontal tab is undefined within preprocessor lines.

Preprocessing itself takes place in several logically successive phases that may, in a particular implementation, be condensed.

1. First, trigraph sequences as described in [Par.A.12.1](#) are replaced by their equivalents. Should the operating system environment require it, newline characters are introduced between the lines of the source file.
2. Each occurrence of a backslash character `\` followed by a newline is deleted, this splicing lines ([Par.A.12.2](#)).
3. The program is split into tokens separated by white-space characters; comments are replaced by a single space. Then preprocessing directives are obeyed, and macros ([Pars.A.12.3-A.12.10](#)) are expanded.
4. Escape sequences in character constants and string literals ([Pars. A.2.5.2, A.2.6](#)) are replaced by their equivalents; then adjacent string literals are concatenated.
5. The result is translated, then linked together with other programs and libraries, by collecting the necessary programs and data, and connecting external functions and object references to their definitions.

A.12.1 Trigraph Sequences

The character set of C source programs is contained within seven-bit ASCII, but is a superset of the ISO 646-1983 Invariant Code Set. In order to enable programs to be represented in the reduced set, all occurrences of the following trigraph sequences are replaced by the corresponding single character. This replacement occurs before any other processing.

<code>??=</code>	<code>#</code>	<code>??(</code>	<code>[</code>	<code>??<</code>	<code>{</code>
<code>??/</code>	<code>\</code>	<code>??)</code>	<code>]</code>	<code>??></code>	<code>}</code>
<code>??'</code>	<code>^</code>	<code>??!</code>	<code> </code>	<code>??-</code>	<code>~</code>

No other such replacements occur.

Trigraph sequences are new with the ANSI standard.

A.12.2 Line Splicing

Lines that end with the backslash character \ are folded by deleting the backslash and the following newline character. This occurs before division into tokens.

A.12.3 Macro Definition and Expansion

A control line of the form

```
# define identifier token-sequence
```

causes the preprocessor to replace subsequent instances of the identifier with the given sequence of tokens; leading and trailing white space around the token sequence is discarded. A second #define for the same identifier is erroneous unless the second token sequence is identical to the first, where all white space separations are taken to be equivalent.

A line of the form

```
# define identifier (identifier-list) token-sequence
```

where there is no space between the first identifier and the (, is a macro definition with parameters given by the identifier list. As with the first form, leading and trailing white space around the token sequence is discarded, and the macro may be redefined only with a definition in which the number and spelling of parameters, and the token sequence, is identical.

A control line of the form

```
# undef identifier
```

causes the identifier's preprocessor definition to be forgotten. It is not erroneous to apply #undef to an unknown identifier.

When a macro has been defined in the second form, subsequent textual instances of the macro identifier followed by optional white space, and then by (, a sequence of tokens separated by commas, and a) constitute a call of the macro. The arguments of the call are the comma-separated token sequences; commas that are quoted or protected by nested parentheses do not separate arguments. During collection, arguments are not macro-expanded. The number of arguments in the call must match the number of parameters in the definition. After the arguments are isolated, leading and trailing white space is removed from them. Then the token sequence resulting from each argument is substituted for each unquoted occurrence of the corresponding parameter's identifier in the replacement token sequence of the macro. Unless the parameter in the replacement sequence is preceded by #, or preceded or followed by ##, the argument tokens are examined for macro calls, and expanded as necessary, just before insertion.

Two special operators influence the replacement process. First, if an occurrence of a parameter in the replacement token sequence is immediately preceded by #, string quotes ("") are placed around the corresponding parameter, and then both the # and the parameter identifier are replaced by the quoted argument. A \ character is inserted before each " or \ character that appears surrounding, or inside, a string literal or character constant in the argument.

Second, if the definition token sequence for either kind of macro contains a ## operator, then just after replacement of the parameters, each ## is deleted, together with any white space on either side, so as to concatenate the adjacent tokens and form a new token. The effect is undefined if invalid tokens are produced, or if the result depends on the order of processing of the ## operators. Also, ## may not appear at the beginning or end of a replacement token sequence.

In both kinds of macro, the replacement token sequence is repeatedly rescanned for more defined identifiers. However, once a given identifier has been replaced in a given expansion, it is not replaced if it turns up again during rescanning; instead it is left unchanged.

Even if the final value of a macro expansion begins with #, it is not taken to be a preprocessing directive.

The details of the macro-expansion process are described more precisely in the ANSI standard than in the first edition. The most important change is the addition of the # and ## operators, which make quotation and concatenation admissible. Some of the new rules, especially those involving concatenation, are bizarre. (See example below.)

For example, this facility may be used for ``manifest-constants," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

The definition

```
#define ABSDIFF(a, b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

defines a macro to return the absolute value of the difference between its arguments. Unlike a function to do the same thing, the arguments and returned value may have any arithmetic type or even be pointers. Also, the arguments, which might have side effects, are evaluated twice, once for the test and once to produce the value.

Given the definition

```
#define tempfile(dir) #dir "%s"
```

the macro call `tempfile(/usr/tmp)` yields

```
" /usr/tmp" "%s"
```

which will subsequently be catenated into a single string. After

```
#define cat(x, y) x ## y
```

the call `cat(var, 123)` yields `var123`. However, the call `cat(cat(1,2),3)` is undefined: the presence of ## prevents the arguments of the outer call from being expanded. Thus it produces the token string

```
cat ( 1 , 2 )3
```

and `)3` (the catenation of the last token of the first argument with the first token of the second) is not a legal token. If a second level of macro definition is introduced,

```
#define xcat(x, y) cat(x,y)
```

things work more smoothly; `xcat(xcat(1, 2), 3)` does produce `123`, because the expansion of `xcat` itself does not involve the ## operator.

Likewise, `ABSDIFF(ABSDIFF(a,b),c)` produces the expected, fully-expanded result.

A.12.4 File Inclusion

A control line of the form

```
# include <filename>
```

causes the replacement of that line by the entire contents of the file *filename*. The characters in the name *filename* must not include > or newline, and the effect is undefined if it contains any of ", ', \, or /*. The named file is searched for in a sequence of implementation-defined places.

Similarly, a control line of the form

```
# include "filename"
```

searches first in association with the original source file (a deliberately implementation-dependent phrase), and if that search fails, then as in the first form. The effect of using ', \, or /* in the filename remains undefined, but > is permitted.

Finally, a directive of the form

include *token-sequence*

not matching one of the previous forms is interpreted by expanding the token sequence as for normal text; one of the two forms with <...> or "... " must result, and is then treated as previously described.

#include files may be nested.

A.12.5 Conditional Compilation

Parts of a program may be compiled conditionally, according to the following schematic syntax.

preprocessor-conditional:

if-line text elif-parts else-part_{opt} #endif

if-line:

if *constant-expression*
ifdef *identifier*
ifndef *identifier*

elif-parts:

elif-line text
elif-parts_{opt}

elif-line:

elif *constant-expression*

else-part:

else-line text

else-line:

#else

Each of the directives (if-line, elif-line, else-line, and #endif) appears alone on a line. The constant expressions in #if and subsequent #elif lines are evaluated in order until an expression with a non-zero value is found; text following a line with a zero value is discarded. The text following the successful directive line is treated normally. ``Text'' here refers to any material, including preprocessor lines, that is not part of the conditional structure; it may be empty. Once a successful #if or #elif line has been found and its text processed, succeeding #elif and #else lines, together with their text, are discarded. If all the expressions are zero, and there is an #else, the text following the #else is treated normally. Text controlled by inactive arms of the conditional is ignored except for checking the nesting of conditionals.

The constant expression in #if and #elif is subject to ordinary macro replacement. Moreover, any expressions of the form

defined *identifier*

or

defined (*identifier*)

are replaced, before scanning for macros, by `1L` if the identifier is defined in the preprocessor, and by `0L` if not. Any identifiers remaining after macro expansion are replaced by `0L`. Finally, each integer constant is considered to be suffixed with `L`, so that all arithmetic is taken to be long or unsigned long.

The resulting constant expression ([Par.A.7.19](#)) is restricted: it must be integral, and may not contain `sizeof`, a cast, or an enumeration constant.

The control lines

```
#ifdef identifier
#ifndef identifier
```

are equivalent to

```
# if defined identifier
# if ! defined identifier
```

respectively.

`#elif` is new since the first edition, although it has been available in some preprocessors. The `defined` preprocessor operator is also new.

A.12.6 Line Control

For the benefit of other preprocessors that generate C programs, a line in one of the forms

```
# line constant "filename"
# line constant
```

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the decimal integer constant and the current input file is named by the identifier. If the quoted filename is absent, the remembered name does not change. Macros in the line are expanded before it is interpreted.

A.12.7 Error Generation

A preprocessor line of the form

```
# error token-sequenceopt
```

causes the preprocessor to write a diagnostic message that includes the token sequence.

A.12.8 Pragmas

A control line of the form

```
# pragma token-sequenceopt
```

causes the preprocessor to perform an implementation-dependent action. An unrecognized pragma is ignored.

A.12.9 Null directive

A control line of the form

```
#
```

has no effect.

A.12.10 Predefined names

Several identifiers are predefined, and expand to produce special information. They, and also the preprocessor expansion operator `defined`, may not be undefined or redefined.

- `__LINE__` A decimal constant containing the current source line number.
 - `__FILE__` A string literal containing the name of the file being compiled.
 - `__DATE__` A string literal containing the date of compilation, in the form "Mmmm dd yyyy"
 - `__TIME__` A string literal containing the time of compilation, in the form "hh:mm:ss"
 - `__STDC__` The constant 1. It is intended that this identifier be defined to be 1 only in standard-conforming implementations.
- #error and #pragma are new with the ANSI standard; the predefined preprocessor macros are new, but some of them have been available in some implementations.

A.13 Grammar

Below is a recapitulation of the grammar that was given throughout the earlier part of this appendix. It has exactly the same content, but is in different order.

The grammar has undefined terminal symbols *integer-constant*, *character-constant*, *floating-constant*, *identifier*, *string*, and *enumeration-constant*; the typewriter style words and symbols are terminals given literally. This grammar can be transformed mechanically into input acceptable for an automatic parser-generator. Besides adding whatever syntactic marking is used to indicate alternatives in productions, it is necessary to expand the ``one of'' constructions, and (depending on the rules of the parser-generator) to duplicate each production with an *opt* symbol, once with the symbol and once without. With one further change, namely deleting the production *typedef-name*: *identifier* and making *typedef-name* a terminal symbol, this grammar is acceptable to the YACC parser-generator. It has only one conflict, generated by the `if-else` ambiguity.

translation-unit:

external-declaration

translation-unit *external-declaration*

external-declaration:

function-definition

declaration

function-definition:

declaration-specifiers_{opt} *declarator declaration-list_{opt}* *compound-statement*

declaration:

declaration-specifiers init-declarator-list_{opt} ;

declaration-list:

declaration

declaration-list declaration

declaration-specifiers:

storage-class-specifier declaration-specifiers_{opt}

type-specifier declaration-specifiers_{opt}

type-qualifier declaration-specifiers_{opt}

storage-class specifier: one of

 auto register static extern typedef

type specifier: one of

 void char short int long float double signed

 unsigned *struct-or-union-specifier enum-specifier typedef-name*

type-qualifier: one of
 const volatile

struct-or-union-specifier:
struct-or-union identifier_{opt} { *struct-declaration-list* }
struct-or-union identifier

struct-or-union: one of
 struct union

struct-declaration-list:
struct declaration
struct-declaration-list struct declaration

init-declarator-list:
init-declarator
init-declarator-list , init-declarator

init-declarator:
declarator
declarator = initializer

struct-declaration:
specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
type-specifier specifier-qualifier-list_{opt}
type-qualifier specifier-qualifier-list_{opt}

struct-declarator-list:
struct-declarator
struct-declarator-list , struct-declarator

struct-declarator:
declarator
declarator_{opt} : constant-expression

enum-specifier:
enum identifier_{opt} { *enumerator-list* }
enum identifier

enumerator-list:
enumerator
enumerator-list , enumerator

enumerator:
identifier
identifier = constant-expression

declarator:
pointer_{opt} direct-declarator

direct-declarator:
identifier
(declarator)
direct-declarator [constant-expression_{opt}]

direct-declarator (*parameter-type-list*)
direct-declarator (*identifier-list*_{opt})

pointer:

- * *type-qualifier-list*_{opt}
- * *type-qualifier-list*_{opt} *pointer*

type-qualifier-list:

- type-qualifier*
- type-qualifier-list* *type-qualifier*

parameter-type-list:

- parameter-list*
- parameter-list* , ...

parameter-list:

- parameter-declaration*
- parameter-list* , *parameter-declaration*

parameter-declaration:

- declaration-specifiers declarator*
- declaration-specifiers abstract-declarator*_{opt}

identifier-list:

- identifier*
- identifier-list* , *identifier*

initializer:

- assignment-expression*
- { *initializer-list* }
- { *initializer-list* , }

initializer-list:

- initializer*
- initializer-list* , *initializer*

type-name:

- specifier-qualifier-list abstract-declarator*_{opt}

abstract-declarator:

- pointer*
- pointer*_{opt} *direct-abstract-declarator*

direct-abstract-declarator:

- (*abstract-declarator*)
- direct-abstract-declarator*_{opt} [*constant-expression*_{opt}]
- direct-abstract-declarator*_{opt} (*parameter-type-list*_{opt})

typedef-name:

- identifier*

statement:

- labeled-statement*
- expression-statement*
- compound-statement*
- selection-statement*

iteration-statement

jump-statement

labeled-statement:

identifier : *statement*

case *constant-expression* : *statement*

default : *statement*

expression-statement:

expression_{opt} ;

compound-statement:

{ *declaration-list_{opt}* *statement-list_{opt}* }

statement-list:

statement

statement-list *statement*

selection-statement:

if (*expression*) *statement*

if (*expression*) *statement* else *statement*

switch (*expression*) *statement*

iteration-statement:

while (*expression*) *statement*

do *statement* while (*expression*);

for (*expression_{opt}*; *expression_{opt}*; *expression_{opt}*) *statement*

jump-statement:

goto *identifier*;

continue;

break;

return *expression_{opt}*;

expression:

assignment-expression

expression , *assignment-expression*

assignment-expression:

conditional-expression

unary-expression *assignment-operator* *assignment-expression*

assignment-operator: one of

= *= /= %= += -= <<= >>= &= ^= |=

conditional-expression:

logical-OR-expression

logical-OR-expression ? *expression* : *conditional-expression*

constant-expression:

conditional-expression

logical-OR-expression:

logical-AND-expression

logical-OR-expression || *logical-AND-expression*

logical-AND-expression:

- inclusive-OR-expression*
- logical-AND-expression && inclusive-OR-expression*

inclusive-OR-expression:

- exclusive-OR-expression*
- inclusive-OR-expression | exclusive-OR-expression*

exclusive-OR-expression:

- AND-expression*
- exclusive-OR-expression ^ AND-expression*

AND-expression:

- equality-expression*
- AND-expression & equality-expression*

equality-expression:

- relational-expression*
- equality-expression == relational-expression*
- equality-expression != relational-expression*

relational-expression:

- shift-expression*
- relational-expression < shift-expression*
- relational-expression > shift-expression*
- relational-expression <= shift-expression*
- relational-expression >= shift-expression*

shift-expression:

- additive-expression*
- shift-expression << additive-expression*
- shift-expression >> additive-expression*

additive-expression:

- multiplicative-expression*
- additive-expression + multiplicative-expression*
- additive-expression - multiplicative-expression*

multiplicative-expression:

- multiplicative-expression * cast-expression*
- multiplicative-expression / cast-expression*
- multiplicative-expression % cast-expression*

cast-expression:

- unary expression*
- (type-name) cast-expression*

unary-expression:

- postfix expression*
- ++unary expression*
- unary expression*
- unary-operator cast-expression*
- sizeof unary-expression*
- sizeof (type-name)*

unary operator: one of

& * + - ~ !

postfix-expression:

primary-expression

postfix-expression[expression]

postfix-expression(argument-expression-list_{opt})

postfix-expression.identifier

postfix-expression->+identifier

postfix-expression++

postfix-expression--

primary-expression:

identifier

constant

string

(expression)

argument-expression-list:

assignment-expression

assignment-expression-list , assignment-expression

constant:

integer-constant

character-constant

floating-constant

enumeration-constant

The following grammar for the preprocessor summarizes the structure of control lines, but is not suitable for mechanized parsing. It includes the symbol *text*, which means ordinary program text, non-conditional preprocessor control lines, or complete preprocessor conditional instructions.

control-line:

```
# define identifier token-sequence
# define identifier(identifier, ... , identifier) token-sequence
# undef identifier
# include <filename>
# include "filename"
# line constant "filename"
# line constant
# error token-sequenceopt
# pragma token-sequenceopt
#
preprocessor-conditional
```

preprocessor-conditional:

```
if-line text elif-parts else-partopt #endif
```

if-line:

```
# if constant-expression
# ifdef identifier
# ifndef identifier
```

elif-parts:
 elif-line text
 elif-parts_{opt}

elif-line:
 # *elif constant-expression*

else-part:
 else-line text

else-line:
 #else

Appendix B - Standard Library

This appendix is a summary of the library defined by the ANSI standard. The standard library is not part of the C language proper, but an environment that supports standard C will provide the function declarations and type and macro definitions of this library. We have omitted a few functions that are of limited utility or easily synthesized from others; we have omitted multi-byte characters; and we have omitted discussion of locale issues; that is, properties that depend on local language, nationality, or culture.

The functions, types and macros of the standard library are declared in standard *headers*:

```
<assert.h>   <float.h>    <math.h>      <stdarg.h>   <stdlib.h>
<cctype.h>   <limits.h>    <setjmp.h>   <stddef.h>   <string.h>
<errno.h>   <locale.h>    <signal.h>   <stdio.h>    <time.h>
```

A header can be accessed by

```
#include <header>
```

Headers may be included in any order and any number of times. A header must be included outside of any external declaration or definition and before any use of anything it declares. A header need not be a source file.

External identifiers that begin with an underscore are reserved for use by the library, as are all other identifiers that begin with an underscore and an upper-case letter or another underscore.

B.1 Input and Output: <stdio.h>

The input and output functions, types, and macros defined in <stdio.h> represent nearly one third of the library.

A *stream* is a source or destination of data that may be associated with a disk or other peripheral. The library supports text streams and binary streams, although on some systems, notably UNIX, these are identical. A text stream is a sequence of lines; each line has zero or more characters and is terminated by '\n'. An environment may need to convert a text stream to or from some other representation (such as mapping '\n' to carriage return and linefeed). A binary stream is a sequence of unprocessed bytes that record internal data, with the property that if it is written, then read back on the same system, it will compare equal.

A stream is connected to a file or device by *opening* it; the connection is broken by *closing* the stream. Opening a file returns a pointer to an object of type `FILE`, which records whatever information is necessary to control the stream. We will use ``file pointer'' and ``stream'' interchangeably when there is no ambiguity.

When a program begins execution, the three streams `stdin`, `stdout`, and `stderr` are already open.

B.1.1 File Operations

The following functions deal with operations on files. The type `size_t` is the unsigned integral type produced by the `sizeof` operator.

```
FILE *fopen(const char *filename, const char *mode)
```

`fopen` opens the named file, and returns a stream, or `NULL` if the attempt fails. Legal values for `mode` include:

- "r" open text file for reading
- "w" create text file for writing; discard previous contents if any
- "a" append; open or create text file for writing at end of file

"r+" open text file for update (i.e., reading and writing)
 "w+" create text file for update, discard previous contents if any
 "a+" append; open or create text file for update, writing at end

Update mode permits reading and writing the same file; `fflush` or a file-positioning function must be called between a read and a write or vice versa. If the mode includes `b` after the initial letter, as in "rb" or "w+b", that indicates a binary file. Filenames are limited to `FILENAME_MAX` characters. At most `FOPEN_MAX` files may be open at once.

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
  freopen opens the file with the specified mode and associates the stream with it. It returns stream, or NULL if an error occurs. freopen is normally used to change the files associated with stdin, stdout, or stderr.
```

```
int fflush(FILE *stream)
  On an output stream, fflush causes any buffered but unwritten data to be written; on an input stream, the effect is undefined. It returns EOF for a write error, and zero otherwise. fflush(NULL) flushes all output streams.
```

```
int fclose(FILE *stream)
  fclose flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. It returns EOF if any errors occurred, and zero otherwise.
```

```
int remove(const char *filename)
  remove removes the named file, so that a subsequent attempt to open it will fail. It returns non-zero if the attempt fails.
```

```
int rename(const char *oldname, const char *newname)
  rename changes the name of a file; it returns non-zero if the attempt fails.
```

```
FILE *tmpfile(void)
  tmpfile creates a temporary file of mode "wb+" that will be automatically removed when closed or when the program terminates normally. tmpfile returns a stream, or NULL if it could not create the file.
```

```
char *tmpnam(char s[L_tmpnam])
  tmpnam(NULL) creates a string that is not the name of an existing file, and returns a pointer to an internal static array. tmpnam(s) stores the string in s as well as returning it as the function value; s must have room for at least L_tmpnam characters. tmpnam generates a different name each time it is called; at most TMP_MAX different names are guaranteed during execution of the program. Note that tmpnam creates a name, not a file.
```

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
  setvbuf controls buffering for the stream; it must be called before reading, writing or any other operation. A mode of _IOFBF causes full buffering, _IOLBF line buffering of text files, and _IONBF no buffering. If buf is not NULL, it will be used as the buffer, otherwise a buffer will be allocated. size determines the buffer size. setvbuf returns non-zero for any error.
```

```
void setbuf(FILE *stream, char *buf)
  If buf is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to (void) setvbuf(stream, buf, _IOFBF, BUFSIZ).
```

B.1.2 Formatted Output

The `printf` functions provide formatted output conversion.

```
int fprintf(FILE *stream, const char *format, ...)
  fprintf converts and writes output to stream under the control of format. The return value is the number of characters written, or negative if an error occurred.
```

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of

the next successive argument to `fprintf`. Each conversion specification begins with the character % and ends with a conversion character. Between the % and the conversion character there may be, in order:

- Flags (in any order), which modify the specification:
 - -, which specifies left adjustment of the converted argument in its field.
 - +, which specifies that the number will always be printed with a sign.
 - space: if the first character is not a sign, a space will be prefixed.
 - 0: for numeric conversions, specifies padding to the field width with leading zeros.
 - #, which specifies an alternate output form. For o, the first digit will become zero. For x or X, 0x or 0X will be prefixed to a non-zero result. For e, E, f, g, and G, the output will always have a decimal point; for g and G, trailing zeros will not be removed.
- A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is 0 if the zero padding flag is present.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for e, E, or f conversions, or the number of significant digits for g or G conversion, or the number of digits to be printed for an integer (leading 0s will be added to make up the necessary width).
- A length modifier h, l (letter ell), or L. ``h'' indicates that the corresponding argument is to be printed as a short or unsigned short; ``l'' indicates that the argument is a long or unsigned long, ``L'' indicates that the argument is a long double.

Width or precision or both may be specified as *, in which case the value is computed by converting the next argument(s), which must be int.

The conversion characters and their meanings are shown in Table B.1. If the character after the % is not a conversion character, the behavior is undefined.

Table B.1 *Printf Conversions*

Character	Argument type; Printed As
d, i	int; signed decimal notation.
o	int; unsigned octal notation (without a leading zero).
x, X	unsigned int; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef for 0x or ABCDEF for 0X.
u	int; unsigned decimal notation.
c	int; single character, after conversion to unsigned char
s	char *; characters from the string are printed until a '\0' is reached or until the number of characters indicated by the precision have been printed.

f	double; decimal notation of the form [-]mmm.ddd, where the number of d's is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
e, E	double; decimal notation of the form [-]m.ddddde+/-xx or [-]m.ddddddE+/-xx, where the number of d's is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
g, G	double; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal point are not printed.
p	void *; print as a pointer (implementation-dependent representation).
n	int *; the number of characters written so far by this call to printf is <i>written into</i> the argument. No argument is converted.
%	no argument is converted; print a %

```
int printf(const char *format, ...)
    printf(...) is equivalent to fprintf(stdout, ...).
int sprintf(char *s, const char *format, ...)
    sprintf is the same as printf except that the output is written into the string s,
    terminated with '\0'. s must be big enough to hold the result. The return count does
    not include the '\0'.
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *stream, const char *format, va_list arg)
int vsprintf(char *s, const char *format, va_list arg)

The functions vprintf, vfprintf, and vsprintf are equivalent to the corresponding
printf functions, except that the variable argument list is replaced by arg, which has
been initialized by the va_start macro and perhaps va_arg calls. See the discussion of
<stdarg.h> in Section B.7.
```

B.1.3 Formatted Input

The scanf function deals with formatted input conversion.

```
int fscanf(FILE *stream, const char *format, ...)
fscanf reads from stream under control of format, and assigns converted values through
subsequent arguments, each of which must be a pointer. It returns when format is exhausted.
fscanf returns EOF if end of file or an error occurs before any conversion; otherwise it returns
the number of input items converted and assigned.
```

The format string usually contains conversion specifications, which are used to direct interpretation of input. The format string may contain:

- Blanks or tabs, which are not ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of a %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l, or L indicating the width of the target, and a conversion character.

A conversion specification determines the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by *, as in %*s, however, the input field is simply skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that scanf will read across line boundaries to find its input, since

newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer. The legal conversion characters are shown in Table B.2.

The conversion characters `d`, `i`, `n`, `o`, `u`, and `x` may be preceded by `h` if the argument is a pointer to `short` rather than `int`, or by `l` (letter ell) if the argument is a pointer to `long`. The conversion characters `e`, `f`, and `g` may be preceded by `l` if a pointer to `double` rather than `float` is in the argument list, and by `L` if a pointer to a `long double`.

Table B.2 Scanf Conversions

Character	Input Data; Argument type
<code>d</code>	decimal integer; <code>int *</code>
<code>i</code>	integer; <code>int *</code> . The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
<code>o</code>	octal integer (with or without leading zero); <code>int *</code> .
<code>u</code>	unsigned decimal integer; <code>unsigned int *</code> .
<code>x</code>	hexadecimal integer (with or without leading 0x or 0X); <code>int *</code> .
<code>c</code>	characters; <code>char *</code> . The next input characters are placed in the indicated array, up to the number given by the width field; the default is 1. No '\0' is added. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use <code>%1s</code> .
<code>s</code>	string of non-white space characters (not quoted); <code>char *</code> , pointing to an array of characters large enough to hold the string and a terminating '\0' that will be added.
<code>e</code> , <code>f</code> , <code>g</code>	floating-point number; <code>float *</code> . The input format for <code>float</code> 's is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an <code>E</code> or <code>e</code> followed by a possibly signed integer.
<code>p</code>	pointer value as printed by <code>printf("%p");</code> <code>void *</code> .
<code>n</code>	writes into the argument the number of characters read so far by this call; <code>int *</code> . No input is read. The converted item count is not incremented.
[...]	matches the longest non-empty string of input characters from the set between brackets; <code>char *</code> . A '\0' is added. []...] includes] in the set.
[^...]	matches the longest non-empty string of input characters <i>not</i> from the set between brackets; <code>char *</code> . A '\0' is added. [^]...] includes] in the set.
<code>%</code>	literal %; no assignment is made.

```
int scanf(const char *format, ...)
    scanf(...) is identical to fscanf(stdin, ...).
int sscanf(const char *s, const char *format, ...)
    sscanf(s, ...) is equivalent to scanf(...) except that the input characters are
    taken from the string s.
```

B.1.4 Character Input and Output Functions

```
int fgetc(FILE *stream)
    fgetc returns the next character of stream as an unsigned char (converted to an
    int), or EOF if end of file or error occurs.
char *fgets(char *s, int n, FILE *stream)
    fgets reads at most the next n-1 characters into the array s, stopping if a newline is
    encountered; the newline is included in the array, which is terminated by '\0'. fgets
    returns s, or NULL if end of file or error occurs.
```

```

int fputc(int c, FILE *stream)
    fputc writes the character c (converted to an unsigned char) on stream. It returns
    the character written, or EOF for error.
int fputs(const char *s, FILE *stream)
    fputs writes the string s (which need not contain \n) on stream; it returns non-
    negative, or EOF for an error.
int getc(FILE *stream)
    getc is equivalent to fgetc except that if it is a macro, it may evaluate stream more
    than once.
int getchar(void)
    getchar is equivalent to getc(stdin).
char *gets(char *s)
    gets reads the next input line into the array s; it replaces the terminating newline with
    '\0'. It returns s, or NULL if end of file or error occurs.
int putc(int c, FILE *stream)
    putc is equivalent to fputc except that if it is a macro, it may evaluate stream more
    than once.
int putchar(int c)
    putchar(c) is equivalent to putc(c,stdout).
int puts(const char *s)
    puts writes the string s and a newline to stdout. It returns EOF if an error occurs,
    non-negative otherwise.
int ungetc(int c, FILE *stream)
    ungetc pushes c (converted to an unsigned char) back onto stream, where it will be
    returned on the next read. Only one character of pushback per stream is guaranteed.
    EOF may not be pushed back. ungetc returns the character pushed back, or EOF for
    error.

```

B.1.5 Direct Input and Output Functions

```

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
    fread reads from stream into the array ptr at most nobj objects of size size. fread
    returns the number of objects read; this may be less than the number requested. feof
    and perror must be used to determine status.
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)
    fwrite writes, from the array ptr, nobj objects of size size on stream. It returns the
    number of objects written, which is less than nobj on error.

```

B.1.6 File Positioning Functions

```

int fseek(FILE *stream, long offset, int origin)
    fseek sets the file position for stream; a subsequent read or write will access data
    beginning at the new position. For a binary file, the position is set to offset characters
    from origin, which may be SEEK_SET (beginning), SEEK_CUR (current position), or
    SEEK_END (end of file). For a text stream, offset must be zero, or a value returned by
    ftell (in which case origin must be SEEK_SET). fseek returns non-zero on error.
long ftell(FILE *stream)
    ftell returns the current file position for stream, or -1 on error.
void rewind(FILE *stream)
    rewind(fp) is equivalent to fseek(fp, 0L, SEEK_SET); clearerr(fp).
int fgetpos(FILE *stream, fpos_t *ptr)
    fgetpos records the current position in stream in *ptr, for subsequent use by
    fsetpos. The type fpos_t is suitable for recording such values. fgetpos returns non-
    zero on error.
int fsetpos(FILE *stream, const fpos_t *ptr)

```

`fsetpos` positions `stream` at the position recorded by `fgetpos` in `*ptr`. `fsetpos` returns non-zero on error.

B.1.7 Error Functions

Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression `errno` (declared in `<errno.h>`) may contain an error number that gives further information about the most recent error.

```
void clearerr(FILE *stream)
    clearerr clears the end of file and error indicators for stream.
int feof(FILE *stream)
    feof returns non-zero if the end of file indicator for stream is set.
int ferror(FILE *stream)
    ferror returns non-zero if the error indicator for stream is set.
void perror(const char *s)
    perror(s) prints s and an implementation-defined error message corresponding to the
    integer in errno, as if by
        fprintf(stderr, "%s: %s\n", s, "error message");
```

See `strerror` in [Section B.3](#).

B.2 Character Class Tests: <ctype.h>

The header `<ctype.h>` declares functions for testing characters. For each function, the argument list is an `int`, whose value must be EOF or representable as an `unsigned char`, and the return value is an `int`. The functions return non-zero (true) if the argument `c` satisfies the condition described, and zero if not.

<code>isalnum(c)</code>	<code>isalpha(c)</code> or <code>isdigit(c)</code> is true
<code>isalpha(c)</code>	<code>isupper(c)</code> or <code>islower(c)</code> is true
<code>iscntrl(c)</code>	control character
<code>isdigit(c)</code>	decimal digit
<code>isgraph(c)</code>	printing character except space
<code>islower(c)</code>	lower-case letter
<code>isprint(c)</code>	printing character including space
<code>ispunct(c)</code>	printing character except space or letter or digit
<code>isspace(c)</code>	space, formfeed, newline, carriage return, tab, vertical tab
<code>isupper(c)</code>	upper-case letter
<code>isxdigit(c)</code>	hexadecimal digit

In the seven-bit ASCII character set, the printing characters are `0x20` (' ') to `0x7E` ('-'); the control characters are `0 NUL` to `0x1F` (US), and `0x7F` (DEL).

In addition, there are two functions that convert the case of letters:

```
int tolower(c) convert c to lower case
int toupper(c) convert c to upper case
```

If `c` is an upper-case letter, `tolower(c)` returns the corresponding lower-case letter, `toupper(c)` returns the corresponding upper-case letter; otherwise it returns `c`.

B.3 String Functions: <string.h>

There are two groups of string functions defined in the header `<string.h>`. The first have names beginning with `str`; the second have names beginning with `mem`. Except for `memmove`, the behavior is undefined if copying takes place between overlapping objects. Comparison functions treat arguments as `unsigned char` arrays.

In the following table, variables `s` and `t` are of type `char *`; `cs` and `ct` are of type `const char *`; `n` is of type `size_t`; and `c` is an `int` converted to `char`.

<code>char *strcpy(s,ct)</code>	copy string <code>ct</code> to string <code>s</code> , including '\0'; return <code>s</code> .
<code>char *strncpy(s,ct,n)</code>	copy at most <code>n</code> characters of string <code>ct</code> to <code>s</code> ; return <code>s</code> . Pad with '\0' if <code>ct</code> has fewer than <code>n</code> characters.
<code>char *strcat(s,ct)</code>	concatenate string <code>ct</code> to end of string <code>s</code> ; return <code>s</code> .
<code>char *strncat(s,ct,n)</code>	concatenate at most <code>n</code> characters of string <code>ct</code> to string <code>s</code> , terminate <code>s</code> with '\0'; return <code>s</code> .
<code>int strcmp(cs,ct)</code>	compare string <code>cs</code> to string <code>ct</code> , return <0 if <code>cs</code> < <code>ct</code> , 0 if <code>cs</code> = <code>ct</code> , or >0 if <code>cs</code> > <code>ct</code> .
<code>int strncmp(cs,ct,n)</code>	compare at most <code>n</code> characters of string <code>cs</code> to string <code>ct</code> ; return <0 if <code>cs</code> < <code>ct</code> , 0 if <code>cs</code> = <code>ct</code> , or >0 if <code>cs</code> > <code>ct</code> .
<code>char *strchr(cs,c)</code>	return pointer to first occurrence of <code>c</code> in <code>cs</code> or <code>NULL</code> if not present.
<code>char * strrchr(cs,c)</code>	return pointer to last occurrence of <code>c</code> in <code>cs</code> or <code>NULL</code> if not present.
<code>size_t strspn(cs,ct)</code>	return length of prefix of <code>cs</code> consisting of characters in <code>ct</code> .
<code>size_t strcspn(cs,ct)</code>	return length of prefix of <code>cs</code> consisting of characters <i>not</i> in <code>ct</code> .
<code>char *strpbrk(cs,ct)</code>	return pointer to first occurrence in string <code>cs</code> of any character string <code>ct</code> , or <code>NULL</code> if not present.
<code>char *strstr(cs,ct)</code>	return pointer to first occurrence of string <code>ct</code> in <code>cs</code> , or <code>NULL</code> if not present.
<code>size_t strlen(cs)</code>	return length of <code>cs</code> .
<code>char *strerror(n)</code>	return pointer to implementation-defined string corresponding to error <code>n</code> .
<code>char * strtok(s,ct)</code>	<code>strtok</code> searches <code>s</code> for tokens delimited by characters from <code>ct</code> ; see below.

A sequence of calls of `strtok(s,ct)` splits `s` into tokens, each delimited by a character from `ct`. The first call in a sequence has a non-`NULL` `s`, it finds the first token in `s` consisting of characters not in `ct`; it terminates that by overwriting the next character of `s` with '\0' and returns a pointer to the token. Each subsequent call, indicated by a `NULL` value of `s`, returns the next such token, searching from just past the end of the previous one. `strtok` returns `NULL` when no further token is found. The string `ct` may be different on each call.

The `mem...` functions are meant for manipulating objects as character arrays; the intent is an interface to efficient routines. In the following table, `s` and `t` are of type `void *`; `cs` and `ct` are of type `const void *`; `n` is of type `size_t`; and `c` is an `int` converted to an `unsigned char`.

<code>void *memcpy(s,ct,n)</code>	copy <code>n</code> characters from <code>ct</code> to <code>s</code> , and return <code>s</code> .
<code>void *memmove(s,ct,n)</code>	same as <code>memcpy</code> except that it works even if the objects overlap.
<code>int memcmp(cs,ct,n)</code>	compare the first <code>n</code> characters of <code>cs</code> with <code>ct</code> ; return as with <code>strcmp</code> .
<code>void *memchr(cs,c,n)</code>	return pointer to first occurrence of character <code>c</code> in <code>cs</code> , or <code>NULL</code> if not present among the first <code>n</code> characters.
<code>void *memset(s,c,n)</code>	place character <code>c</code> into first <code>n</code> characters of <code>s</code> , return <code>s</code> .

B.4 Mathematical Functions: <math.h>

The header <math.h> declares mathematical functions and macros.

The macros `EDOM` and `ERANGE` (found in <errno.h>) are non-zero integral constants that are used to signal domain and range errors for the functions; `HUGE_VAL` is a positive double value. A *domain error* occurs if an argument is outside the domain over which the function is defined. On a domain error, `errno` is set to `EDOM`; the return value is implementation-defined. A *range error* occurs if the result of the function cannot be represented as a `double`. If the

result overflows, the function returns `HUGE_VAL` with the right sign, and `errno` is set to `ERANGE`. If the result underflows, the function returns zero; whether `errno` is set to `ERANGE` is implementation-defined.

In the following table, `x` and `y` are of type `double`, `n` is an `int`, and all functions return `double`. Angles for trigonometric functions are expressed in radians.

<code>sin(x)</code>	sine of x
<code>cos(x)</code>	cosine of x
<code>tan(x)</code>	tangent of x
<code>asin(x)</code>	$\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$, x in $[-1, 1]$.
<code>acos(x)</code>	$\cos^{-1}(x)$ in range $[0, \pi]$, x in $[-1, 1]$.
<code>atan(x)</code>	$\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$.
<code>atan2(y, x)</code>	$\tan^{-1}(y/x)$ in range $[-\pi, \pi]$.
<code>sinh(x)</code>	hyperbolic sine of x
<code>cosh(x)</code>	hyperbolic cosine of x
<code>tanh(x)</code>	hyperbolic tangent of x
<code>exp(x)</code>	exponential function e^x
<code>log(x)</code>	natural logarithm $\ln(x)$, $x > 0$.
<code>log10(x)</code>	base 10 logarithm $\log_{10}(x)$, $x > 0$.
<code>pow(x, y)</code>	x^y . A domain error occurs if $x=0$ and $y \leq 0$, or if $x < 0$ and y is not an integer.
<code>sqrt(x)</code>	square root of x , $x \geq 0$.
<code>ceil(x)</code>	smallest integer not less than x , as a <code>double</code> .
<code>floor(x)</code>	largest integer not greater than x , as a <code>double</code> .
<code>fabs(x)</code>	absolute value $ x $
<code>ldexp(x, n)</code>	$x \cdot 2^n$
<code>frexp(x, int *ip)</code>	splits x into a normalized fraction in the interval $[1/2, 1)$ which is returned, and a power of 2, which is stored in <code>*exp</code> . If x is zero, both parts of the result are zero.
<code>modf(x, double *ip)</code>	splits x into integral and fractional parts, each with the same sign as x . It stores the integral part in <code>*ip</code> , and returns the fractional part.
<code>fmod(x, y)</code>	floating-point remainder of x/y , with the same sign as x . If y is zero, the result is implementation-defined.

B.5 Utility Functions: <stdlib.h>

The header `<stdlib.h>` declares functions for number conversion, storage allocation, and similar tasks.

```
double atof(const char *s)
    atof converts s to double; it is equivalent to strtod(s, (char**)NULL).

int atoi(const char *s)
    converts s to int; it is equivalent to (int)strtol(s, (char**)NULL, 10).

long atol(const char *s)
    converts s to long; it is equivalent to strtol(s, (char**)NULL, 10).

double strtod(const char *s, char **endp)
    strtod converts the prefix of s to double, ignoring leading white space; it stores a pointer to any unconverted suffix in *endp unless endp is NULL. If the answer would overflow, HUGE_VAL is returned with the proper sign; if the answer would underflow, zero is returned. In either case errno is set to ERANGE.

long strtol(const char *s, char **endp, int base)
    strtol converts the prefix of s to long, ignoring leading white space; it stores a pointer to any unconverted suffix in *endp unless endp is NULL. If base is between 2 and 36, conversion is done assuming that the input is written in that base. If base is zero, the base is 8, 10, or 16; leading 0 implies octal and leading 0x or 0X hexadecimal.
```

Letters in either case represent digits from 10 to base-1; a leading 0x or 0X is permitted in base 16. If the answer would overflow, LONG_MAX or LONG_MIN is returned, depending on the sign of the result, and errno is set to ERANGE.

```
unsigned long strtoul(const char *s, char **endp, int base)
    strtoul is the same as strtol except that the result is unsigned long and the error
    value is ULONG_MAX.

int rand(void)
    rand returns a pseudo-random integer in the range 0 to RAND_MAX, which is at least
    32767.

void srand(unsigned int seed)
    srand uses seed as the seed for a new sequence of pseudo-random numbers. The
    initial seed is 1.

void *calloc(size_t nobj, size_t size)
    calloc returns a pointer to space for an array of nobj objects, each of size size, or
    NULL if the request cannot be satisfied. The space is initialized to zero bytes.

void *malloc(size_t size)
    malloc returns a pointer to space for an object of size size, or NULL if the request
    cannot be satisfied. The space is uninitialized.

void *realloc(void *p, size_t size)
    realloc changes the size of the object pointed to by p to size. The contents will be
    unchanged up to the minimum of the old and new sizes. If the new size is larger, the
    new space is uninitialized. realloc returns a pointer to the new space, or NULL if the
    request cannot be satisfied, in which case *p is unchanged.

void free(void *p)
    free deallocates the space pointed to by p; it does nothing if p is NULL. p must be a
    pointer to space previously allocated by calloc, malloc, or realloc.

void abort(void)
    abort causes the program to terminate abnormally, as if by raise(SIGABRT).

void exit(int status)
    exit causes normal program termination. atexit functions are called in reverse order
    of registration, open files are flushed, open streams are closed, and control is returned
    to the environment. How status is returned to the environment is implementation-
    dependent, but zero is taken as successful termination. The values EXIT_SUCCESS and
    EXIT_FAILURE may also be used.

int atexit(void (*fcn)(void))
    atexit registers the function fcn to be called when the program terminates normally;
    it returns non-zero if the registration cannot be made.

int system(const char *s)
    system passes the string s to the environment for execution. If s is NULL, system
    returns non-zero if there is a command processor. If s is not NULL, the return value is
    implementation-dependent.

char *getenv(const char *name)
    getenv returns the environment string associated with name, or NULL if no string exists.
    Details are implementation-dependent.

void *bsearch(const void *key, const void *base,
              size_t n, size_t size,
              int (*cmp)(const void *keyval, const void *datum))
    bsearch searches base[0]...base[n-1] for an item that matches *key. The function
    cmp must return negative if its first argument (the search key) is less than its second (a
    table entry), zero if equal, and positive if greater. Items in the array base must be in
    ascending order. bsearch returns a pointer to a matching item, or NULL if none exists.

void qsort(void *base, size_t n, size_t size,
```

```

int (*cmp)(const void *, const void *)
qsort sorts into ascending order an array base[0]...base[n-1] of objects of size
size. The comparison function cmp is as in bsearch.
int abs(int n)
    abs returns the absolute value of its int argument.
long labs(long n)
    labs returns the absolute value of its long argument.
div_t div(int num, int denom)
    div computes the quotient and remainder of num/denom. The results are stored in the
    int members quot and rem of a structure of type div_t.
ldiv_t ldiv(long num, long denom)
    ldiv computes the quotient and remainder of num/denom. The results are stored in the
    long members quot and rem of a structure of type ldiv_t.

```

B.6 Diagnostics: <assert.h>

The assert macro is used to add diagnostics to programs:

```
void assert(int expression)
```

If *expression* is zero when

```
assert(expression)
```

is executed, the assert macro will print on stderr a message, such as

```
Assertion failed: expression, file filename, line nnn
```

It then calls abort to terminate execution. The source filename and line number come from the preprocessor macros __FILE__ and __LINE__.

If NDEBUG is defined at the time <assert.h> is included, the assert macro is ignored.

B.7 Variable Argument Lists: <stdarg.h>

The header <stdarg.h> provides facilities for stepping through a list of function arguments of unknown number and type.

Suppose *lastarg* is the last named parameter of a function *f* with a variable number of arguments. Then declare within *f* a variable of type *va_list* that will point to each argument in turn:

```
va_list ap;
```

ap must be initialized once with the macro *va_start* before any unnamed argument is accessed:

```
va_start(va_list ap, lastarg);
```

Thereafter, each execution of the macro *va_arg* will produce a value that has the type and value of the next unnamed argument, and will also modify *ap* so the next use of *va_arg* returns the next argument:

```
type va_arg(va_list ap, type);
```

The macro

```
void va_end(va_list ap);
```

must be called once after the arguments have been processed but before *f* is exited.

B.8 Non-local Jumps: <setjmp.h>

The declarations in <setjmp.h> provide a way to avoid the normal function call and return sequence, typically to permit an immediate return from a deeply nested function call.

```
int setjmp(jmp_buf env)
```

The macro `setjmp` saves state information in `env` for use by `longjmp`. The return is zero from a direct call of `setjmp`, and non-zero from a subsequent call of `longjmp`. A call to `setjmp` can only occur in certain contexts, basically the test of `if`, `switch`, and loops, and only in simple relational expressions.

```
if (setjmp(env) == 0)
    /* get here on direct call */
else
    /* get here by calling longjmp */
void longjmp(jmp_buf env, int val)
```

`longjmp` restores the state saved by the most recent call to `setjmp`, using the information saved in `env`, and execution resumes as if the `setjmp` function had just executed and returned the non-zero value `val`. The function containing the `setjmp` must not have terminated. Accessible objects have the values they had at the time `longjmp` was called, except that non-volatile automatic variables in the function calling `setjmp` become undefined if they were changed after the `setjmp` call.

B.9 Signals: <signal.h>

The header <signal.h> provides facilities for handling exceptional conditions that arise during execution, such as an interrupt signal from an external source or an error in execution.

```
void (*signal(int sig, void (*handler)(int)))(int)
```

`signal` determines how subsequent signals will be handled. If `handler` is `SIG_DFL`, the implementation-defined default behavior is used, if it is `SIG_IGN`, the signal is ignored; otherwise, the function pointed to by `handler` will be called, with the argument of the type of signal. Valid signals include

<code>SIGABRT</code>	abnormal termination, e.g., from <code>abort</code>
<code>SIGFPE</code>	arithmetic error, e.g., zero divide or overflow
<code>SIGILL</code>	illegal function image, e.g., illegal instruction
<code>SIGINT</code>	interactive attention, e.g., interrupt
<code>SIGSEGV</code>	illegal storage access, e.g., access outside memory limits
<code>SIGTERM</code>	termination request sent to this program

`signal` returns the previous value of `handler` for the specific signal, or `SIG_ERR` if an error occurs.

When a signal `sig` subsequently occurs, the signal is restored to its default behavior; then the signal-handler function is called, as if by `(*handler)(sig)`. If the handler returns, execution will resume where it was when the signal occurred.

The initial state of signals is implementation-defined.

```
int raise(int sig)
raise sends the signal sig to the program; it returns non-zero if unsuccessful.
```

B.10 Date and Time Functions: <time.h>

The header <time.h> declares types and functions for manipulating date and time. Some functions process *local time*, which may differ from calendar time, for example because of time

zone. `clock_t` and `time_t` are arithmetic types representing times, and `struct tm` holds the components of a calendar time:

int <code>tm_sec</code> ;	seconds after the minute (0,61)
int <code>tm_min</code> ;	minutes after the hour (0,59)
int <code>tm_hour</code> ;	hours since midnight (0,23)
int <code>tm_mday</code> ;	day of the month (1,31)
int <code>tm_mon</code> ;	months <i>since January</i> (0,11)
int <code>tm_year</code> ;	years since 1900
int <code>tm_wday</code> ;	days since Sunday (0,6)
int <code>tm_yday</code> ;	days since January 1 (0,365)
int <code>tm_isdst</code> ;	Daylight Saving Time flag

`tm_isdst` is positive if Daylight Saving Time is in effect, zero if not, and negative if the information is not available.

```
clock_t clock(void)
    clock returns the processor time used by the program since the beginning of execution,
    or -1 if unavailable. clock() / CLK_PER_SEC is a time in seconds.

time_t time(time_t *tp)
    time returns the current calendar time or -1 if the time is not available. If tp is not
    NULL, the return value is also assigned to *tp.

double difftime(time_t time2, time_t time1)
    difftime returns time2-time1 expressed in seconds.

time_t mktime(struct tm *tp)
    mktime converts the local time in the structure *tp into calendar time in the same
    representation used by time. The components will have values in the ranges shown.
    mktime returns the calendar time or -1 if it cannot be represented.
```

The next four functions return pointers to static objects that may be overwritten by other calls.

```
char *asctime(const struct tm *tp)
    asctime</tt> converts the time in the structure *tp into a string of
    the form

        Sun Jan  3 15:14:13 1988\n\0
char *ctime(const time_t *tp)
    ctime converts the calendar time *tp to local time; it is equivalent
    to

        asctime(localtime(tp))
struct tm *gmtime(const time_t *tp)
    gmtime converts the calendar time *tp into Coordinated Universal Time
    (UTC). It returns NULL if UTC is not available. The name gmtime has
    historical significance.
struct tm *localtime(const time_t *tp)
    localtime converts the calendar time *tp into local time.

size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)
    strftime formats date and time information from *tp into s according
    to fmt, which is analogous to a printf format. Ordinary characters
    (including the terminating '\0') are copied into s. Each %c is
    replaced as described below, using values appropriate for the local
    environment. No more than smax characters are placed into s. strftime
    returns the number of characters, excluding the '\0', or zero if more
    than smax characters were produced.

    %a  abbreviated weekday name.
    %A  full weekday name.
    %b  abbreviated month name.
    %B  full month name.
    %c  local date and time representation.
    %d  day of the month (01-31).
    %H  hour (24-hour clock) (00-23).
```

%I	hour (12-hour clock) (01-12).
%j	day of the year (001-366).
%m	month (01-12).
%M	minute (00-59).
%p	local equivalent of AM or PM.
%S	second (00-61).
%U	week number of the year (Sunday as 1st day of week) (00-53).
%w	weekday (0-6, Sunday is 0).
%W	week number of the year (Monday as 1st day of week) (00-53).
%x	local date representation.
%X	local time representation.
%Y	year without century (00-99).
% <u>Y</u>	year with century.
%Z	time zone name, if any.
%%	%

B.11 Implementation-defined Limits: <limits.h> and <float.h>

The header <limits.h> defines constants for the sizes of integral types. The values below are acceptable minimum magnitudes; larger values may be used.

CHAR_BIT	8	bits in a char
CHAR_MAX	UCHAR_MAX or SCHAR_MAX	maximum value of char
CHAR_MIN	0 or SCHAR_MIN	maximum value of char
INT_MAX	32767	maximum value of int
INT_MIN	-32767	minimum value of int
LONG_MAX	2147483647	maximum value of long
LONG_MIN	-2147483647	minimum value of long
SCHAR_MAX	+127	maximum value of signed char
SCHAR_MIN	-127	minimum value of signed char
SHRT_MAX	+32767	maximum value of short
SHRT_MIN	-32767	minimum value of short
UCHAR_MAX	255	maximum value of unsigned char
UINT_MAX	65535	maximum value of unsigned int
ULONG_MAX	4294967295	maximum value of unsigned long
USHRT_MAX	65535	maximum value of unsigned short

The names in the table below, a subset of <float.h>, are constants related to floating-point arithmetic. When a value is given, it represents the minimum magnitude for the corresponding quantity. Each implementation defines appropriate values.

FLT_RADIX	2	radix of exponent representation, e.g., 2, 16
FLT_ROUNDS		floating-point rounding mode for addition
FLT_DIG	6	decimal digits of precision
FLT_EPSILON	1E-5	smallest number x such that $1.0+x \neq 1.0$
FLT_MANT_DIG		number of base FLT_RADIX in mantissa
FLT_MAX	1E+37	maximum floating-point number
FLT_MAX_EXP		maximum n such that FLT_RADIX^{n-1} is representable
FLT_MIN	1E-37	minimum normalized floating-point number
FLT_MIN_EXP		minimum n such that 10^n is a normalized number
DBL_DIG	10	decimal digits of precision
DBL_EPSILON	1E-9	smallest number x such that $1.0+x \neq 1.0$
DBL_MANT_DIG		number of base FLT_RADIX in mantissa

DBL_MAX	1E+37	maximum double floating-point number
DBL_MAX_EXP		maximum n such that FLT_RADIX^{n-1} is representable
DBL_MIN	1E-37	minimum normalized double floating-point number
DBL_MIN_EXP		minimum n such that 10^n is a normalized number

Appendix C - Summary of Changes

Since the publication of the first edition of this book, the definition of the C language has undergone changes. Almost all were extensions of the original language, and were carefully designed to remain compatible with existing practice; some repaired ambiguities in the original description; and some represent modifications that change existing practice. Many of the new facilities were announced in the documents accompanying compilers available from AT&T, and have subsequently been adopted by other suppliers of C compilers. More recently, the ANSI committee standardizing the language incorporated most of the changes, and also introduced other significant modifications. Their report was in part participated by some commercial compilers even before issuance of the formal C standard.

This Appendix summarizes the differences between the language defined by the first edition of this book, and that expected to be defined by the final standard. It treats only the language itself, not its environment and library; although these are an important part of the standard, there is little to compare with, because the first edition did not attempt to prescribe an environment or library.

- Preprocessing is more carefully defined in the Standard than in the first edition, and is extended: it is explicitly token based; there are new operators for concatenation of tokens (##), and creation of strings (#); there are new control lines like #elif and #pragma; redeclaration of macros by the same token sequence is explicitly permitted; parameters inside strings are no longer replaced. Splicing of lines by \ is permitted everywhere, not just in strings and macro definitions. See [Par.A.12](#).
- The minimum significance of all internal identifiers increased to 31 characters; the smallest mandated significance of identifiers with external linkage remains 6 monocase letters. (Many implementations provide more.)
- Trigraph sequences introduced by ?? allow representation of characters lacking in some character sets. Escapes for #\^[]{}|~ are defined, see [Par.A.12.1](#). Observe that the introduction of trigraphs may change the meaning of strings containing the sequence ??.
- New keywords (void, const, volatile, signed, enum) are introduced. The stillborn entry keyword is withdrawn.
- New escape sequences, for use within character constants and string literals, are defined. The effect of following \ by a character not part of an approved escape sequence is undefined. See [Par.A.2.5.2](#).
- Everyone's favorite trivial change: 8 and 9 are not octal digits.
- The standard introduces a larger set of suffixes to make the type of constants explicit: U or L for integers, F or L for floating. It also refines the rules for the type of unsuffixed constants ([Par.A.2.5](#)).
- Adjacent string literals are concatenated.
- There is a notation for wide-character string literals and character constants; see [Par.A.2.6](#).
- Characters as well as other types, may be explicitly declared to carry, or not to carry, a sign by using the keywords signed or unsigned. The locution long float as a

synonym for `double` is withdrawn, but `long double` may be used to declare an extra-precision floating quantity.

- For some time, type `unsigned char` has been available. The standard introduces the `signed` keyword to make signedness explicit for `char` and other integral objects.
- The `void` type has been available in most implementations for some years. The Standard introduces the use of the `void *` type as a generic pointer type; previously `char *` played this role. At the same time, explicit rules are enacted against mixing pointers and integers, and pointers of different type, without the use of casts.
- The Standard places explicit minima on the ranges of the arithmetic types, and mandates headers (`<limits.h>` and `<float.h>`) giving the characteristics of each particular implementation.
- Enumerations are new since the first edition of this book.
- The Standard adopts from C++ the notion of type qualifier, for example `const`. See [Par.A.8.2](#).
- Strings are no longer modifiable, and so may be placed in read-only memory.
- The ``usual arithmetic conversions'' are changed, essentially from ``for integers, `unsigned` always wins; for floating point, always use `double`'' to ``promote to the smallest capacious-enough type.'' See [Par.A.6.5](#).
- The old assignment operators like `=+` are truly gone. Also, assignment operators are now single tokens; in the first edition, they were pairs, and could be separated by white space.
- A compiler's license to treat mathematically associative operators as computationally associative is revoked.
- A unary `+` operator is introduced for symmetry with unary `-`.
- A pointer to a function may be used as a function designator without an explicit `*` operator. See [Par.A.7.3.2](#).
- Structures may be assigned, passed to functions, and returned by functions.
- Applying the address-of operator to arrays is permitted, and the result is a pointer to the array.
- The `sizeof` operator, in the first edition, yielded type `int`; subsequently, many implementations made it `unsigned`. The Standard makes its type explicitly implementation-dependent, but requires the type, `size_t`, to be defined in a standard header (`<stddef.h>`). A similar change occurs in the type (`ptrdiff_t`) of the difference between pointers. See [Par.A.7.4.8](#) and [Par.A.7.7](#).
- The address-of operator `&` may not be applied to an object declared `register`, even if the implementation chooses not to keep the object in a register.
- The type of a shift expression is that of the left operand; the right operand can't promote the result. See [Par.A.7.8](#).
- The Standard legalizes the creation of a pointer just beyond the end of an array, and allows arithmetic and relations on it; see [Par.A.7.7](#).

- The Standard introduces (borrowing from C++) the notion of a function prototype declaration that incorporates the types of the parameters, and includes an explicit recognition of variadic functions together with an approved way of dealing with them. See Pars. [A.7.3.2](#), [A.8.6.3](#), [B.7](#). The older style is still accepted, with restrictions.
- Empty declarations, which have no declarators and don't declare at least a structure, union, or enumeration, are forbidden by the Standard. On the other hand, a declaration with just a structure or union tag redeclares that tag even if it was declared in an outer scope.
- External data declarations without any specifiers or qualifiers (just a naked declarator) are forbidden.
- Some implementations, when presented with an `extern` declaration in an inner block, would export the declaration to the rest of the file. The Standard makes it clear that the scope of such a declaration is just the block.
- The scope of parameters is injected into a function's compound statement, so that variable declarations at the top level of the function cannot hide the parameters.
- The name spaces of identifiers are somewhat different. The Standard puts all tags in a single name space, and also introduces a separate name space for labels; see [Par.A.11.1](#). Also, member names are associated with the structure or union of which they are a part. (This has been common practice from some time.)
- Unions may be initialized; the initializer refers to the first member.
- Automatic structures, unions, and arrays may be initialized, albeit in a restricted way.
- Character arrays with an explicit size may be initialized by a string literal with exactly that many characters (the `\0` is quietly squeezed out).
- The controlling expression, and the case labels, of a switch may have any integral type.

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either expressed or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, Bell Telephone Laboratories assumes no responsibility for their use by the recipient. Further, Bell Laboratories assumes no obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

UNIX PROGRAMMER'S MANUAL

Fifth Edition

K. Thompson
D. M. Ritchie

June, 1974

Copyright © 1972, 1973, 1974
Bell Telephone Laboratories, Incorporated

Copyright © 1972, 1973, 1974
Bell Telephone Laboratories, Incorporated

This manual was set by a Graphic Systems phototypesetter driven by the *troff* formatting program operating under the UNIX system. The text of the manual was prepared using the *ed* text editor.

June 1974

PREFACE to the Fifth Edition

The number of UNIX installations is now above 50, and many more are expected. None of these has exactly the same complement of hardware or software. Therefore, at any particular installation, it is quite possible that this manual will give inappropriate information.

The authors are grateful to L. L. Cherry, L. A. Dimino, R. C. Haight, S. C. Johnson, B. W. Kernighan, M. E. Lesk, and E. N. Pinson for their contributions to the system software, and to L. E. McMahon for software and for his contributions to this manual. We are particularly appreciative of the invaluable technical, editorial, and administrative efforts of J. F. Ossanna, M. D. McIlroy, and R. Morris. They all contributed greatly to the stock of UNIX software and to this manual. Their inventiveness, thoughtful criticism, and ungrudging support increased immeasurably not only whatever success the UNIX system enjoys, but also our own enjoyment in its creation.

INTRODUCTION TO THIS MANUAL

This manual gives descriptions of the publicly available features of UNIX. It provides neither a general overview (see "The UNIX Time-sharing System" for that) nor details of the implementation of the system (which remain to be disclosed).

Within the area it surveys, this manual attempts to be as complete and timely as possible. A conscious decision was made to describe each program in exactly the state it was in at the time its manual section was prepared. In particular, the desire to describe something as it should be, not as it is, was resisted. Inevitably, this means that many sections will soon be out of date.

This manual is divided into eight sections:

- I. Commands
- II. System calls
- III. Subroutines
- IV. Special files
- V. File formats
- VI. User-maintained programs
- VII. Miscellaneous
- VIII. Maintenance

Commands are programs intended to be invoked directly by the user, in contradistinction to subroutines, which are intended to be called by the user's programs. Commands generally reside in directory */bin* (for *binary* programs). This directory is searched automatically by the command line interpreter. Some programs also reside in */usr/bin*, to save space in */bin*. Some programs classified as commands are located elsewhere; this fact is indicated in the appropriate sections.

System calls are entries into the UNIX supervisor. In assembly language, they are coded with the use of the opcode *sys*, a synonym for the *trap* instruction. In this edition, the C language interface routines to the system calls have been incorporated in section II.

A small assortment of subroutines is available; they are described in section III. The binary form of most of them is kept in the system library */lib/liba.a*. The subroutines available from C and from Fortran are also included; they reside in */lib/libc.a* and */lib/libf.a* respectively.

The special files section IV discusses the characteristics of each system "file" which actually refers to an I/O device. The names in this section refer to the DEC device names for the hardware, instead of the names of the special files themselves.

The file formats section V documents the structure of particular kinds of files; for example, the form of the output of the loader and assembler is given. Excluded are files used by only one command, for example the assembler's intermediate files.

User-maintained programs (section VI) are not considered part of the UNIX system, and the principal reason for listing them is to indicate their existence without necessarily giving a complete description. The author should be consulted for information.

The miscellaneous section (VII) gathers odds and ends.

Section VIII discusses commands which are not intended for use by the ordinary user, in some cases because they disclose information in which he is presumably not interested, and in others because they perform privileged functions.

Each section consists of a number of independent entries of a page or so each. The name of the entry is in the upper corners of its pages, its preparation date in the upper middle. Entries within each section are alphabetized. The page numbers of each entry start at 1. (The earlier

hope for frequent, partial updates of the manual is clearly in vain, but in any event it is not feasible to maintain consecutive page numbering in a document like this.)

All entries are based on a common format, not all of whose subsections will always appear.

The *name* section repeats the entry name and gives a very short description of its purpose.

The *synopsis* summarizes the use of the program being described. A few conventions are used, particularly in the Commands section:

Boldface words are considered literals, and are typed just as they appear.

Square brackets ([]) around an argument indicate that the argument is optional. When an argument is given as "name", it always refers to a file name.

Ellipses "..." are used to show that the previous argument-prototype may be repeated.

A final convention is used by the commands themselves. An argument beginning with a minus sign “-” is often taken to mean some sort of flag argument even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with “-”.

The *description* section discusses in detail the subject at hand.

The *files* section gives the names of files which are built into the program.

A *see also* section gives pointers to related information.

A *diagnostics* section discusses the diagnostic indications which may be produced. Messages which are intended to be self-explanatory are not listed.

The *bugs* section gives known bugs and sometimes deficiencies. Occasionally also the suggested fix is described.

At the beginning of this document is a table of contents, organized by section and alphabetically within each section. There is also a permuted index derived from the table of contents. Within each index entry, the title of the writeup to which it refers is followed by the appropriate section number in parentheses. This fact is important because there is considerable name duplication among the sections, arising principally from commands which exist only to exercise a particular system call.

This manual was prepared using the UNIX text editor *ed* and the formatting program *troff*.

HOW TO GET STARTED

This section provides the basic information you need to get started on UNIX: how to log in and log out, how to communicate through your terminal, and how to run a program.

Logging in. You must call UNIX from an appropriate terminal. UNIX supports ASCII terminals typified by the TTY 37, the GE Terminet 300, the Memorex 1240, and various graphical terminals. You must also have a valid user name, which may be obtained, together with the telephone number, from the system administrators. The same telephone number serves terminals operating at all the standard speeds. After a data connection is established, the login procedure depends on what kind of terminal you are using.

300-baud terminals: Such terminals include the GE Terminet 300, most display terminals, Execuport, TI, and certain Anderson-Jacobson terminals. These terminals generally have a speed switch which should be set at "300" (or "30" for 30 characters per second) and a half/full duplex switch which should be set at full-duplex. (Note that this switch will often have to be changed since many other systems require half-duplex). When a connection is established, the system types "login:"; you type your user name, followed by the "return" key. If you have a password, the system asks for it and turns off the printer on the terminal so the password will not appear. After you have logged in, the "return", "new line", or "linefeed" keys will give exactly the same results.

TTY 37 terminal: When you have established a data connection, the system types out a few garbage characters (the "login:" message at the wrong speed). Depress the "break" (or "interrupt") key; this is a speed-independent signal to UNIX that a 150-baud terminal is in use. The system then will type "login:" this time at the correct speed; you respond with your user name. From the TTY 37 terminal, and any other which has the "new-line" function (combined carriage return and linefeed), terminate each line you type with the "new-line" key (*not* the "return" key).

For all these terminals, it is important that you type your name in lower-case if possible; if you type upper-case letters, UNIX will assume that your terminal cannot generate lower-case letters and will translate all subsequent upper-case letters to lower case.

The evidence that you have successfully logged in is that the Shell program will type a "%" to you. (The Shell is described below under "How to run a program.")

For more information, consult *getty* (VIII), which discusses the login sequence in more detail, and *tty* (IV), which discusses typewriter I/O.

Logging out. There are three ways to log out:

You can simply hang up the phone.

You can log out by typing an end-of-file indication (EOT character, control "d") to the Shell. The Shell will terminate and the "login:" message will appear again.

You can also log in directly as another user by giving a *login* command (I).

How to communicate through your terminal. When you type to UNIX, a gnome deep in the system is gathering your characters and saving them in a secret place. The characters will not be given to a program until you type a return (or new-line), as described above in *Logging in*.

UNIX typewriter I/O is full-duplex. It has full read-ahead, which means that you can type at any time, even while a program is typing at you. Of course, if you type during output, the output will have the input characters interspersed. However, whatever you type will be saved up and interpreted in correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded unless the system is in trouble. When the read-ahead limit is exceeded, the system throws away all the saved characters.

On a typewriter input line, the character "@" kills all the characters typed before it, so typing mistakes can be repaired on a single line. Also, the character "#" erases the last character typed. Successive uses of "#" erase characters back to, but not beyond, the beginning of the line. "@" and "#" can be transmitted to a program by preceding them with "\". (So, to erase "\", you need two "#'s).

The ASCII "delete" (a.k.a. "rubout") character is not passed to programs but instead generates an *interrupt signal*. This signal generally causes whatever program you are running to terminate. It is typically used to stop a long printout that you don't want. However, programs can arrange either to ignore this signal altogether, or to be notified when it happens (instead of being terminated). The editor, for example, catches interrupts and stops what it is doing, instead of terminating, so that an interrupt can be used to halt an editor printout without losing the file being edited.

The *quit* signal is generated by typing the ASCII FS character. It not only causes a running program to terminate but also generates a file with the core image of the terminated process. Quit is useful for debugging.

Besides adapting to the speed of the terminal, UNIX tries to be intelligent about whether you have a terminal with the new-line function or whether it must be simulated with carriage-return and line-feed. In the latter case, all input carriage returns are turned to new-line characters (the standard line delimiter) and both a carriage return and a line feed are echoed to the terminal. If you get into the wrong mode, the *stty* command (I) will rescue you.

Tab characters are used freely in UNIX source programs. If your terminal does not have the tab function, you can arrange to have them turned into spaces during output, and echoed as spaces during input. The system assumes that tabs are set every eight columns. Again, the *stty* command (I) will set or reset this mode. Also, there is a file which, if printed on TTY 37 or TermiNet 300 terminals, will set the tab stops correctly (*tabs* (VII)).

Section *tty* (IV) discusses typewriter I/O more fully. Section *k1* (IV) discusses the console typewriter.

How to run a program; The Shell. When you have successfully logged into UNIX, a program called the Shell is listening to your terminal. The Shell reads typed-in lines, splits them up into a command name and arguments, and executes the command. A command is simply an executable program. The Shell looks first in your current directory (see next section) for a program with the given name, and if none is there, then in a system directory. There is nothing special about system-provided commands except that they are kept in a directory where the Shell can find them.

The command name is always the first word on an input line; it and its arguments are separated from one another by spaces.

When a program terminates, the Shell will ordinarily regain control and type a "%" at you to indicate that it is ready for another command.

The Shell has many other capabilities, which are described in detail in section *sh*(I).

The current directory. UNIX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he also created a directory for you (ordinarily with the same name as your user name). When you log in, any file name you type is by default in this directory. Since you are the owner of this directory, you have full permissions to read, write, alter, or destroy its contents. Permissions to have your will with other directories and files will have been granted or denied to you by their owners. As a matter of observed fact, few UNIX users protect their files from destruction, let alone perusal, by other users.

To change the current directory (but not the set of permissions you were endowed with at login) use *chdir* (I).

Path names. To refer to files not in the current directory, you must use a path name. Full path names begin with "/", the name of the root directory of the whole file system. After the slash comes the name of each directory containing the next sub-directory (followed by a "/") until finally the file name is reached. E.g.: */usr/lem/filex* refers to the file *filex* in the directory *lem*; *lem* is itself a subdirectory of *usr*; *usr* springs directly from the root directory.

If your current directory has subdirectories, the path names of files therein begin with the name of the subdirectory (no prefixed "/").

Without important exception, a path name may be used anywhere a file name is required.

Important commands which modify the contents of files are *cp* (I), *mv* (I), and *rm* (I), which respectively copy, move (i.e. rename) and remove files. To find out the status of files or directories, use *ls* (I). See *mkdir* (I) for making directories; *rmdir* (I) for destroying them.

For a fuller discussion of the file system, see "The UNIX Time-Sharing System," by the present authors, to appear in the Communications of the ACM; a version is also available from the same source as this manual. It may also be useful to glance through section II of this manual, which discusses system calls, even if you don't intend to deal with the system at that level.

Writing a program. To enter the text of a source program into a UNIX file, use *ed* (I). The three principal languages in UNIX are assembly language (see *as* (I)), Fortran (see *fc* (I)), and C (see *cc* (I)). After the program text has been entered through the editor and written on a file, you can give the file to the appropriate language processor as an argument. The output of the language processor will be left on a file in the current directory named "a.out". (If the output is precious, use *mv* to move it to a less exposed name soon.) If you wrote in assembly language, you will probably need to load the program with library subroutines; see *ld* (I). The other two language processors call the loader automatically.

When you have finally gone through this entire process without provoking any diagnostics, the resulting program can be run by giving its name to the Shell in response to the "%" prompt.

The next command you will need is *db* (I). As a debugger, *db* is better than average for assembly-language programs, marginally useful for C programs (when completed, *cdb* (I) will be a boon), and virtually useless for Fortran.

Your programs can receive arguments from the command line just as system programs do. See *exec* (II).

Text processing. Almost all text is entered through the editor. The commands most often used to write text on a terminal are: *cat*, *pr*, *roff*, *nroff*, and *troff*, all in section I.

The *cat* command simply dumps ASCII text on the terminal, with no processing at all. The *pr* command paginates the text, supplies headings, and has a facility for multi-column output. *Troff* and *nroff* are elaborate text formatting programs, and require careful forethought in entering both the text and the formatting commands into the input file. *Troff* drives a Graphic Systems phototypesetter; it was used to produce this manual. *Nroff* produces output on a typewriter terminal. *Roff* (I) is a somewhat less elaborate text formatting program, and requires somewhat less forethought.

Surprises. Certain commands provide inter-user communication. Even if you do not plan to use them, it would be well to learn something about them, because someone else may aim them at you.

To communicate with another user currently logged in, *write* (I) is used; *mail* (I) will leave a message whose presence will be announced to another user when he next logs in. The write-ups in the manual also suggest how to respond to the two commands if you are a target.

When you log in, a message-of-the-day may greet you before the first "%".

TABLE OF CONTENTS

I. COMMANDS

ar	archive and library maintainer
as	assembler
cat	concatenate and print
cc	C compiler
cdb	C debugger
chdir	change working directory
chmod	change mode
chown	change owner
cmp	compare two files
comm	print lines common to two files
cp	copy
cref	make cross reference listing
date	print and set the date
db	debug
dc	desk calculator
dd	convert and copy a file
diff	differential file comparator
dsw	delete interactively
du	summarize disk usage
echo	echo arguments
ed	text editor
eqn	typeset mathematics
exit	terminate command file
fc	Fortran compiler
fed	edit form letter memory
find	find files
form	form letter generator
goto	command transfer
grep	search a file for a pattern
if	conditional command
kill	terminate a process
ld	link editor
ln	make a link
login	sign onto UNIX
lpr	spool for line printer
ls	list contents of directory
mail	send mail to another user
man	run off section of UNIX manual
mesg	permit or deny messages
mkdir	make a directory
mv	move or rename a file
neqn	typeset mathematics on terminal
nice	run a command at low priority
nm	print name list
nohup	run a command immune to hangups
nroff	format text
od	octal dump
opr	off line print
passwd	set login password
pfe	print floating exception

II. SYSTEM CALLS

intro	introduction to system calls
break	change core allocation
chdir	change working directory
chmod	change mode of file
chown	change owner
close	close a file
creat	create a new file
csw	read console switches
dup	duplicate an open file descriptor
exec	execute a file
exit	terminate process
fork	spawn new process
fstat	get status of open file
getgid	get group identifications
getuid	get user identifications
gtty	get typewriter status
indir	indirect system call
kill	send signal to a process
link	link to a file
mknod	make a directory or a special file

mount	mount file system
nice	set program priority
open	open for reading or writing
pipe	create an interprocess channel
profil	execution time profile
read	read from file
seek	move read/write pointer
setgid	set process group ID
setuid	set process user ID
signal	catch or ignore signals
sleep	stop execution for interval
stat	get file status
stime	set time
stty	set mode of typewriter
sync	update super-block
time	get date and time
times	get process times
umount	dismount file system
unlink	remove directory entry
wait	wait for process to terminate
write	write on a file

III. SUBROUTINES

alloc	core allocator
atan	arc tangent function
atof	ASCII to floating
crypt	password encoding
ctime	convert date and time to ASCII
ecvt	output conversion
exp	exponential function
floor	floor and ceiling functions
fptrap	floating point interpreter
gamma	log gamma function
getarg	get command arguments from Fortran
getc	buffered input
getchar	read character
getpw	get name from UID
hmul	high-order product
hypot	calculate hypotenuse
ierror	catch Fortran errors
ldiv	long division
locv	long output conversion
log	natural logarithm
monitor	prepare execution profile
nargs	argument count
nlist	get entries from name list
perror	system error messages
pow	floating exponentiation
printf	formatted print
putc	buffered output
putchar	write character
qsort	quicker sort
rand	random number generator

reset	execute non-local goto
setfil	specify Fortran file name
sin	sine and cosine functions
sqrt	square root function
ttyn	return name of current typewriter
vt	display (vt01) interface

IV. SPECIAL FILES

cat	phototypesetter interface
dc	DC-11 communications interface
dh	DH-11 communications multiplexer
dn	DN-11 ACU interface
dp	DP-11 201 data-phone interface
kl	KL-11 or DL-11 asynchronous interface
lp	line printer
mem	core memory
pc	PC-11 paper tape reader/punch
rf	RF11/RS11 fixed-head disk file
rk	RK-11/RK03 (or RK05) disk
rp	RP-11/RP03 moving-head disk
tc	TC-11/TU56 DECtape
tiu	Spider interface
tm	TM-11/TU-10 magtape interface
tty	general typewriter interface
vs	voice synthesizer interface
vt	11/20 (vt01) interface

V. FILE FORMATS

a.out	assembler and link editor output
ar	archive (library) file format
core	format of core image file
dir	format of directories
dump	incremental dump tape format
fs	format of file system volume
mtab	mounted file system table
passwd	password file
speak.m	voice synthesizer vocabulary
tp	DEC/mag tape formats
ttys	typewriter initialization data
utmp	user information
wtmp	user login history

VI. USER MAINTAINED PROGRAMS

apl	APL interpreter
azel	satellite predictions
bas	basic
bj	the game of black jack
cal	print calendar
catsim	phototypesetter simulator
chess	the game of chess
col	filter reverse line feeds

cubic	three dimensional tic-tac-toe
factor	discover prime factors of a number
graf	draw graph on GSI terminal
gsi	interpret funny characters on GSI terminal
hyphen	find hyphenated words
ibm	submit off-line job to HO IBM 370
m6	general purpose macroprocessor
maze	generate a maze problem
moo	guessing game
npr	print file on Spider line-printer
plog	make a graph on the gsi terminal
plot	make a graph
ptx	permuted index
sfs	structured file scanner
sky	obtain ephemerides
sno	Snobol interpreter
speak	word to voice translator
spline	interpolate smooth curve
tmg	compiler-compiler
ttt	the game of tic-tac-toe
wump	the game of hunt-the-wumpus
yacc	yet another compiler-compiler

VII. MISCELLANEOUS

ascii	map of ASCII character set
greek	graphics for extended TTY-37 type-box
tabs	set tab stops
tmheader	TM cover sheet
vs	voice synthesizer code

VIII. SYSTEM MAINTENANCE

20boot	install new 11/20 system
ac	login accounting
boot procedures	UNIX startup
check	file system consistency check
clri	clear i-node
df	disk free
dpd	data phone daemon
dump	incremental file system dump
getty	set typewriter mode
glob	generate command arguments
init	process control initialization
lpd	line printer daemon
mkfs	construct a file system
mknod	build special file
mount	mount file system
msh	mini Shell
reloc	relocate object files
restor	incremental file system restore
sa	Shell accounting
su	become privileged user
sync	update the super block

umount dismount file system
update periodically update the super block

PERMUTED INDEX

20boot(VIII) install new	11/20 system
vt(IV)	11/20 (vt01) interface
dp(IV) DP-11	201 data-phone interface
ibm(VI) submit off-line job to HO IBM	20boot(VIII) install new 11/20 system
ac(VIII) login	370
sa(VIII) Shell	accounting
dn(IV) DN-11	accounting
shift(I)	ACU interface
break(II) change core	ac(VIII) login accounting
alloc(III) core	adjust Shell arguments
yacc(VI) yet	allocation
mail(I) send mail to	allocator
write(I) write to	alloc(III) core allocator
apl(VI)	another compiler-compiler
atan(III)	another user
ar(I)	another user
ar(V)	a.out(V) assembler and link editor output
nargs(III)	APL interpreter
getarg(III) get command	apl(VI) APL interpreter
echo(I) echo	arc tangent function
glob(VIII) generate command	archive and library maintainer
shift(I) adjust Shell	archive (library) file format
ascii(VII) map of	argument count
atof(III)	arguments from Fortran
ctime(III) convert date and time to	arguments
a.out(V)	arguments
as(I)	ar(I) archive and library maintainer
kl(IV) KL-11 or DL-11	ar(V) archive (library) file format
nice(I) run a command	ASCII character set
wait(I)	ASCII to floating
bas(VI)	ASCII
su(VIII)	ascii(VII) map of ASCII character set
strip(I) remove symbols and relocation	as(I) assembler
bj(VI) the game of	assembler and link editor output
sync(VIII) update the super	assembler
update(VIII) periodically update the super	asynchronous interface
	at low priority
	atan(III) arc tangent function
	atof(III) ASCII to floating
	await completion of process
	azel(VI) satellite predictions
	basic
	bas(VI) basic
	become privileged user
	bits
	bj(VI) the game of black jack
	black jack
	block
	block
	boot procedures(VIII) UNIX startup

restor(VIII)	incremental	file system restore
	mtab(V)	file system table
	fs(V)	file system volume
mkfs(VIII)	construct a	file system
	mount(II)	file system
	mount(VIII)	file system
	umount(II)	dismount
	umount(VIII)	dismount
chmod(II)	change mode of	file
	close(II)	close a
core(V)	format of core image	file
	creat(II)	create a new
	dd(I)	convert and copy a
	exec(II)	execute a
exit(I)	terminate command	file
fstat(II)	get status of open	file
	link(II)	link to a
mknod(II)	make a directory or a special	file
	mknod(VIII)	build special
	mv(I)	move or rename a
	passwd(V)	password
	pr(I)	print
	read(II)	read from
rf(IV)	RF11/RS11 fixed-head disk	file
	cmp(I)	compare two
comm(I)	print lines common to two	files
	find(I)	find
	size(I)	size of an object
reloc(VIII)	relocate object	files
	rm(I)	remove (unlink)
	sort(I)	sort or merge
	sum(I)	sum
uniq(I)	report repeated lines in a	file
	write(II)	write on a
	col(VI)	filter reverse line feeds
	find(I)	find files
	hyphen(VI)	find hyphenated words
	typo(I)	find possible typos
	spell(I)	find spelling errors
	find(I)	find files
	tee(I)	pipe
rf(IV)	RF11/RS11	fixed-head disk file
	pfe(I)	print
	pow(III)	floating exception
	fptrap(III)	floating exponentiation
atof(III)	ASCII to	floating point interpreter
	floor(III)	floating
		floor and ceiling functions
		floor(III) floor and ceiling functions
	fork(II)	spawn new process
	form(I)	form letter generator
fed(I)	edit	form letter memory
core(V)	format of core image file	
dir(V)	format of directories	
fs(V)	format of file system volume	

sh(I) shell	(command interpreter)
goto(I)	command transfer
if(I) conditional	command
time(I) time a	command
comm(I) print lines	comm(I) print lines common to two files
dc(IV) DC-11	common to two files
'dh(IV) DH-11	communications interface
diff(I) differential file	communications multiplexer
cmp(I)	comparator
cc(I) C	compare two files
tmg(VI)	compiler
yacc(VI) yet another	compiler-compiler
fc(I) Fortran	compiler-compiler
wait(I) await	compiler
cat(I)	completion of process
if(I)	concatenate and print
check(VIII) file system	conditional command
csw(II) read	consistency check
mkfs(VIII)	console switches
ls(I) list	construct a file system
init(VIII) process	contents of directory
ecvt(III) output	control initialization
locv(III) long output	conversion
dd(I)	conversion
ctime(III)	convert and copy a file
dd(I) convert and	convert date and time to ASCII
cp(I)	copy a file
break(II) change	copy
alloc(III)	core allocation
core(V) format of	core allocator
mem(IV)	core image file
sin(III) sine and	core memory
nargs(III) argument	core(V) format of core image file
wc(I) word	cosine functions
tmheader(VII) TM	count
creat(II)	count
pipe(II)	cover sheet
cref(I) make	cp(I) copy
ttyn(III) return name of	create a new file
spline(VI) interpolate smooth	create an interprocess channel
dpd(VIII) data phone	creat(II) create a new file
lpd(VIII) line printer	cref(I) make cross reference listing
dpd(VIII)	cross reference listing
dp(IV) DP-11 201	crypt(III) password encoding
prof(I) display profile	csw(II) read console switches
	ctime(III) convert date and time to ASCII
	cubic(VI) three dimensional tic-tac-toe
	current typewriter
	curve
	daemon
	daemon
	data phone daemon
	data-phone interface
	data

ttys(V) typewriter initialization	data
ctime(III) convert	date and time to ASCII
time(II) get	date and time
date(I) print and set the	date
dc(IV)	date(I) print and set the date
db(I) debug	db(I) debug
dc(IV) DC-11 communications interface	DC-11 communications interface
dc(I) desk calculator	dc(I) desk calculator
dc(IV) DC-11 communications interface	dc(IV) DC-11 communications interface
dd(I) convert and copy a file	dd(I) convert and copy a file
db(I) debug	debug
cdb(I) C	debugger
tp(V)	DEC/mag tape formats
tp(I) manipulate	DECtape and magtape
tc(IV) TC-11/TU56	DECtape
dsw(I)	delete interactively
mesg(I) permit or	deny messages
dup(II) duplicate an open file	descriptor
dc(I)	desk calculator
dh(IV)	df(VIII) disk free
dh(IV)	DH-11 communications multiplexer
diff(I)	dh(IV) DH-11 communications multiplexer
cubic(VI) three	differential file comparator
dir(V) format of	diff(I) differential file comparator
unlink(II) remove	dimensional tic-tac-toe
pwd(I) working	directories
mknod(II) make a	directory entry
chdir(I) change working	directory name
chdir(II) change working	directory or a special file
ls(I) list contents of	directory
mkdir(I) make a	directory
rmdir(I) remove	directory
factor(VI)	dir(V) format of directories
rf(IV) RF11/RS11 fixed-head	discover prime factors of a number
df(VIII)	disk file
du(I) summarize	disk free
rk(IV) RK-11/RK03 (or RK05)	disk usage
rp(IV) RP-11/RP03 moving-head	disk
umount(II)	disk
umount(VIII)	dismount file system
prof(I)	dismount file system
vt(III)	display profile data
ldiv(III) long	display (vt01) interface
kl(IV) KL-11 or	division
dn(IV)	DL-11 asynchronous interface
dp(IV)	DN-11 ACU interface
graf(VI)	dn(IV) DN-11 ACU interface
dp(IV)	DP-11 201 data-phone interface
dpd(VIII) data phone daemon	dpd(VIII) data phone daemon
dp(IV)	dp(IV) DP-11 201 data-phone interface
graf(VI)	draw graph on GSI terminal
dsw(I)	dsw(I) delete interactively

dump(V) incremental	du(I) summarize disk usage
dump(VIII) incremental file system	dump tape format
od(I) octal	dump
	dump
	dump(V) incremental dump tape format
	dump(VIII) incremental file system dump
	dup(II) duplicate an open file descriptor
	duplicate an open file descriptor
dup(II)	echo arguments
echo(I)	echo(I) echo arguments
	ecvt(III) output conversion
	ed(I) text editor
fed(I)	edit form letter memory
a.out(V) assembler and link	editor output
ed(I) text	editor
ld(I) link	editor
	encoding
crypt(III) password	entries from name list
nlist(III) get	entry
unlink(II) remove directory	ephemerides
sky(VI) obtain	eqn(I) typeset mathematics
	error messages
perror(III) system	errors
ierror(III) catch Fortran	errors
spell(I) find spelling	exception
pfe(I) print floating	exec(II) execute a file
	execute a file
exec(II)	execute non-local goto
reset(III)	execution for an interval
sleep(I) suspend	execution for interval
sleep(II) stop	execution profile
monitor(III) prepare	execution time profile
profil(II)	exit(I) terminate command file
	exit(II) terminate process
	exp(III) exponential function
	exponential function
exp(III)	exponentiation
pow(III) floating	extended TTY-37 type-box
greek(VII) graphics for	factors of a number
factor(VI) discover prime	factor(VI) discover prime factors of a number
	fc(I) Fortran compiler
	fed(I) edit form letter memory
	feeds
col(VI) filter reverse line	file comparator
diff(I) differential	file descriptor
dup(II) duplicate an open	file for a pattern
grep(I) search a	file format
ar(V) archive (library)	file into pieces
split(I) split a	file name
setfil(III) specify Fortran	file on Spider line-printer
npr(VI) print	file scanner
sfs(VI) structured	file status
stat(II) get	file system consistency check
check(VIII)	file system dump
dump(VIII) incremental	

restor(VIII)	incremental	file system restore
	mtab(V)	file system table
	fs(V)	file system volume
mkfs(VIII)	construct a	file system
	mount(II)	file system
	mount(VIII)	file system
	umount(II)	file system
	umount(VIII)	file system
chmod(II)	change mode of	file system
	close(II)	file
core(V)	format of core image	file
	creat(II)	file
	dd(I)	file
	exec(II)	file
exit(I)	terminate command	file
fstat(II)	get status of open	file
	link(II)	file
mknod(II)	make a directory or a special	file
	mknod(VIII)	file
	mv(I)	file
	passwd(V)	password
	pr(I)	print
rf(IV)	RF11/RS11 fixed-head disk	file
	cmp(I)	files
comm(I)	print lines common to two	files
	find(I)	files
	size(I)	file
reloc(VIII)	relocate object	files
	rm(I)	files
	sort(I)	files
	sum(I)	file
uniq(I)	report repeated lines in a	file
	write(II)	file
	col(VI)	filter reverse line feeds
	find(I)	find files
	hyphen(VI)	find hyphenated words
	typo(I)	find possible typos
	spell(I)	find spelling errors
	find(I)	find files
	tee(I)	fitting
rf(IV)	RF11/RS11	fixed-head disk file
	pfe(I)	floating exception
	pow(III)	floating exponentiation
	fptrap(III)	floating point interpreter
atof(III)	ASCII to	floating
	floor(III)	floor and ceiling functions
	form(I)	floor(II) floor and ceiling functions
fed(I)	edit	fork(II) spawn new process
core(V)		form letter generator
dir(V)		form letter memory
fs(V)		format of core image file
		format of directories
		format of file system volume

nroff(I)	format text
roff(I)	format text
troff(I)	format text
ar(V) archive (library) file	
dump(V) incremental dump tape	format
tp(V) DEC/mag tape	formats
printf(III)	formatted print
fc(I)	form(I) form letter generator
ierror(III) catch	Fortran compiler
setfil(III) specify	Fortran errors
getarg(III) get command arguments from	Fortran file name
df(VIII) disk	Fortran
read(II) read	fptrap(III) floating point interpreter
getarg(III) get command arguments	free
nlist(III) get entries	from file
getpw(III) get name	from Fortran
atan(III) arc tangent	from name list
exp(III) exponential	from UID
gamma(III) log gamma	fstat(II) get status of open file
floor(III) floor and ceiling	fs(V) format of file system volume
sqrt(III) square root	function
sin(III) sine and cosine	function
gsi(VI) interpret	functions
bj(VI) the	funny characters on GSI terminal
chess(VI) the	game of black jack
wump(VI) the	game of chess
ttt(VI) the	game of hunt-the-wumpus
moo(VI) guessing	game of tic-tac-toe
gamma(III) log	game
m6(VI)	gamma function
tty(IV)	gamma(III) log gamma function
maze(VI)	general purpose macroprocessor
glob(VIII)	general typewriter interface
form(I) form letter	generate a maze problem
rand(III) random number	generate command arguments
getarg(III)	generator
time(II)	generator
nlist(III)	get command arguments from Fortran
stat(II)	get date and time
getgid(II)	get entries from name list
getpw(III)	get file status
times(II)	get group identifications
fstat(II)	get name from UID
tty(I)	get process times
gtty(II)	get status of open file
getuid(II)	get typewriter name
	get typewriter status
	get user identifications
	getarg(III) get command arguments from Fortran
	getchar(III) read character
	getc(III) buffered input

	getgid(II) get group identifications
	getpw(III) get name from UID
	getty(VIII) set typewriter mode
	getuid(II) get user identifications
	glob(VIII) generate command arguments
	goto(I) command transfer
reset(III) execute non-local	goto
	graf(VI) draw graph on GSI terminal
	plog(VI) make a
	greek(VII)
	plot(VI) make a
	getgid(II) get
	setgid(II) set process
	graf(VI) draw graph on
gsi(VI) interpret funny characters on	GSI terminal
	plog(VI) make a graph on the
	moo(VI)
nohup(I) run a command immune to	hangups
	hmul(III)
	wtmp(V) user login
ibm(VI) submit off-line job to	high-order product
	wump(VI) the game of
	hyphen(VI) find
	hypot(III) calculate
ibm(VI) submit off-line job to HO	HO IBM 370
	getgid(II) get group
	getuid(II) get user
	setgid(II) set process group
	setuid(II) set process user
	signal(II) catch or
	core(V) format of core
	nohup(I) run a command
	uniq(I) report repeated lines
	dump(V)
	dump(VIII)
	restor(VIII)
	ptx(VI) permuted
	indir(II)
	utmp(V) user
	ttys(V) typewriter
	init(VIII) process control
	identifications
	identifications
	ID
	ID
	ierror(III) catch Fortran errors
	if(I) conditional command
	ignore signals
	image file
	immune to hangups
	in a file
	incremental dump tape format
	incremental file system dump
	incremental file system restore
	index
	indirect system call
	indir(II) indirect system call
	information
	initialization data
	initialization
	init(VIII) process control initialization

clri(VIII)	clear	i-node
getc(III)	buffered	input
	20boot(VIII)	install new 11/20 system
dsw(I)	delete	interactively
	tss(I)	interface to MH-TSS
cat(IV)	phototypesetter	interface
dc(IV)	DC-11 communications	interface
	dn(IV)	DN-11 ACU
dp(IV)	DP-11 201 data-phone	interface
kl(IV)	KL-11 or DL-11 asynchronous	interface
	tiu(IV)	Spider
tm(IV)	TM-11/TU-10 magtape	interface
	tty(IV)	general typewriter
	vs(IV)	voice synthesizer
	vt(II)	display (vt01)
	vt(IV)	11/20 (vt01)
		spline(VI)
		gsi(VI)
	apl(VI)	APL
fptrap(III)	floating point	interpreter
sh(I)	shell (command	interpreter
	sno(VI)	Snobol
	pipe(II)	create an
sleep(I)	suspend execution for an	interprocess channel
	sleep(II)	stop execution for
	split(I)	interval
		into pieces
	intro(II)	introduction to system calls
		intro(II) introduction to system calls
bj(VI)	the game of black	jack
ibm(VI)	submit off-line	job to HO IBM 370
		kill(I) terminate a process
		kill(II) send signal to a process
kl(IV)		KL-11 or DL-11 asynchronous interface
		kl(IV) KL-11 or DL-11 asynchronous interface
	form(I)	ld(I) link editor
fed(I)	edit form	ldiv(III) long division
	ar(V)	letter generator
	ar(I)	letter memory
col(VI)	filter reverse	(library) file format
	lpd(VIII)	library maintainer
	lp(IV)	line feeds
	lpr(I)	line printer daemon
	opr(I)	line printer
npr(VI)	print file on Spider	line printer
	comm(I)	line print
	uniq(I)	line-printer
a.out(V)	assembler and	lines common to two files
	ld(I)	lines in a file
	link(II)	link editor output
In(I)	make a	link editor
ls(I)		link to a file
		link(II) link to a file
		link
		list contents of directory

cref(I) make cross reference	listing
nlist(III) get entries from name	list
nm(I) print name	list
	ln(I) make a link
	locv(III) long output conversion
gamma(III)	log gamma function
log(III) natural	logarithm
	log(III) natural logarithm
ac(VIII)	login accounting
wtmp(V) user	login history
passwd(I) set	login password
	login(I) sign onto UNIX
ldiv(III)	long division
locv(III)	long output conversion
nice(I) run a command at	low priority
	lpd(VIII) line printer daemon
m6(VI) general purpose	lp(IV) line printer
tm(IV) TM-11/TU-10	lpr(I) spool for line printer
tp(I) manipulate DECTape and	ls(I) list contents of directory
mail(I) send	m6(VI) general purpose macroprocessor
	macroprocessor
ar(I) archive and library	magtape interface
	magtape
	mail to another user
mknod(II)	mail(I) send mail to another user
mkdir(I)	maintainer
plog(VI)	make a directory or a special file
plot(VI)	make a directory
ln(I)	make a graph on the gsi terminal
cref(I)	make a graph
	make a link
tp(I)	make cross reference listing
man(I) run off section of UNIX	man(I) run off section of UNIX manual
ascii(VII)	manipulate DECTape and magtape
neqn(I) typeset	manual
eqn(I) typeset	map of ASCII character set
maze(VI) generate a	mathematics on terminal
	mathematics
fed(I) edit form letter	maze problem
mem(IV) core	maze(VI) generate a maze problem
sort(I) sort or	mem(IV) core memory
	memory
	memory
	merge files
mesg(I) permit or deny	mesg(I) permit or deny messages
perror(III) system error	messages
tss(I) interface to	messages
msh(VIII)	MH-TSS
	mini Shell
	mkdir(I) make a directory
	mkfs(VIII) construct a file system
	mknod(II) make a directory or a special file
	mknod(VIII) build special file
chmod(II) change	mode of file

stty(II) set	mode of typewriter
chmod(I) change	mode
getty(VIII) set	typewriter mode
	monitor(III) prepare execution profile
	moo(VI) guessing game
mount(II)	mount file system
mount(VIII)	mount file system
mtab(V)	mounted file system table
	mount(II) mount file system
	mount(VIII) mount file system
mv(I)	move or rename a file
seek(II)	move read/write pointer
rp(IV) RP-11/RP03	moving-head disk
	msh(VIII) mini Shell
	mtab(V) mounted file system table
dh(IV) DH-11 communications	multiplexer
getpw(III) get	name from UID
nlist(III) get entries from	name list
	nm(I) print
	name list
	ttyn(III) return
	name of current typewriter
pwd(I) working directory	name
setfil(III) specify Fortran file	name
tty(I) get typewriter	name
	nargs(III) argument count
log(III)	natural logarithm
20boot(VIII) install	neqn(I) typeset mathematics on terminal
creat(II) create a	new 11/20 system
fork(II) spawn	new file
	new process
	nice(I) run a command at low priority
	nice(II) set program priority
	nlist(III) get entries from name list
	nm(I) print name list
	nohup(I) run a command immune to hangups
reset(III) execute	non-local goto
	npr(VI) print file on Spider line-printer
	nroff(I) format text
	number generator
rand(III) random	number
factor(VI) discover prime factors of a	object file
	size(I) size of an
	reloc(VIII) relocate
	sky(VI) obtain ephemerides
	od(I) octal dump
	od(I) octal dump
	opr(I) off line print
man(I) run	off section of UNIX manual
ibm(VI) submit	off-line job to HO IBM 370
login(I) sign	onto UNIX
dup(II) duplicate an	open file descriptor
fstat(II) get status of	open file
	open(II) open for reading or writing
	open(II) open for reading or writing
	opr(I) off line print

stty(I)	set typewriter
rk(IV)	RK-11/RK03
ecvt(III)	
locv(III)	long
a.out(V)	assembler and link editor
putc(III)	buffered
chown(I)	change
chown(II)	change
pc(IV)	PC-11
crypt(III)	
passwd(V)	
passwd(I)	set login
grep(I)	search a file for a
pc(IV)	
update(VIII)	
mesg(I)	
ptx(VI)	
dpd(VIII)	data
cat(IV)	
catsim(VI)	
split(I)	split a file into
tee(I)	
fptrap(III)	floating
seek(II)	move read/write
typo(I)	find
azel(VI)	satellite
monitor(III)	
factor(VI)	discover
date(I)	
cal(VI)	
npr(VI)	
pr(I)	
pfe(I)	
comm(I)	
nm(I)	
cat(I)	concatenate and
lpd(VIII)	line
lp(IV)	line
lpr(I)	spool for line
opr(I)	off line
printf(III)	formatted
nice(I)	run a command at low
nice(II)	set program
options	
(or RK05) disk	
output conversion	
output conversion	
output	
output	
owner	
owner	
paper tape reader/punch	
passwd(I)	set login password
passwd(V)	password file
password encoding	
password file	
password	
pattern	
PC-11 paper tape reader/punch	
pc(IV)	PC-11 paper tape reader/punch
periodically update the super block	
permit or deny messages	
permuted index	
perror(III)	system error messages
pfe(I)	print floating exception
phone daemon	
phototypesetter interface	
phototypesetter simulator	
pieces	
pipe fitting	
pipe(II)	create an interprocess channel
plog(VI)	make a graph on the gsi terminal
plot(VI)	make a graph
point interpreter	
pointer	
possible typos	
pow(III)	floating exponentiation
predictions	
prepare execution profile	
pr(I)	print file
prime factors of a number	
print and set the date	
print calendar	
print file on Spider line-printer	
print file	
print floating exception	
print lines common to two files	
print name list	
print	
printer daemon	
printer	
printer	
printf(III)	formatted print
print	
print	
priority	
priority	

su(VIII)	become privileged user
maze(VI)	generate a maze
boot	problem
init(VIII)	procedures(VIII) UNIX startup
setgid(II)	process control initialization
set	process group ID
ps(I)	process status
times(II)	process times
get	process to terminate
wait(II)	process user ID
wait for	process
'setuid(II)	process
set	process
exit(II)	process
terminate	process
fork(II)	process
spawn new	process
kill(I)	process
terminate a	process
kill(II)	process
send signal to a	process
wait(I)	process
await completion of	process
hmul(III)	product
high-order	prof(I) display profile data
prof(I)	profile data
display	profile
monitor(III)	prepare execution
profil(II)	profile
execution time	profil(II) execution time profile
nice(II)	program priority
set	ps(I) process status
m6(VI)	general
qsort(III)	ptx(VI) permuted index
general	purpose macroprocessor
qsort(III)	putchar(III) write character
rand(III)	putc(III) buffered output
getchar(III)	pwd(I) working directory name
csw(II)	qsort(III) quicker sort
read(II)	quicker sort
pc(IV)	PC-11 paper tape
open(II)	rand(III) random number generator
open for	random number generator
seek(II)	read character
move	read console switches
cref(I)	read from file
make cross	reader/punch
reloc(VIII)	read(II) read from file
strip(I)	reading or writing
remove symbols and	read/write pointer
unlink(II)	reference listing
rmmdir(I)	relocate object files
strip(I)	relocation bits
rm(I)	reloc(VIII) relocate object files
mv(I)	remove directory entry
move or	remove directory
uniq(I)	remove symbols and relocation bits
report	remove (unlink) files
uniq(I)	rename a file
restor(VIII)	repeated lines in a file
incremental file system	report repeated lines in a file
reset(III)	reset(III) execute non-local goto
restore	restore
restor(VIII)	restor(VIII) incremental file system restore
incremental file system	return name of current typewriter
ttypn(III)	reverse line feeds
col(VI)	filter

	rew(I) rewind tape
rew(I)	rewind tape
rf(IV)	RF11/RS11 fixed-head disk file
rk(IV) RK-11/RK03 (or rk(IV)	rk(IV) RF11/RS11 fixed-head disk file RK05) disk
	RK-11/RK03 (or RK05) disk
	rk(IV) RK-11/RK03 (or RK05) disk
	rmdir(I) remove directory
	rm(I) remove (unlink) files
	roff(I) format text
sqrt(III) square	root function
rp(IV)	RP-11/RP03 moving-head disk
	rp(IV) RP-11/RP03 moving-head disk
nice(I)	run a command at low priority
nohup(I)	run a command immune to hangups
man(I)	run off section of UNIX manual
azel(VI)	satellite predictions
sfs(VI) structured file	sa(VIII) Shell accounting
grep(I)	scanner
man(I) run off	search a file for a pattern
	section of UNIX manual
	seek(II) move read/write pointer
mail(I)	send mail to another user
kill(II)	send signal to a process
passwd(I)	set login password
stty(II)	set mode of typewriter
setgid(II)	set process group ID
setuid(II)	set process user ID
nice(II)	set program priority
tabs(VII)	set tab stops
date(I) print and	set the date
stime(II)	set time
getty(VIII)	set typewriter mode
stty(I)	set typewriter options
ascii(VII) map of ASCII character	set
	setfil(III) specify Fortran file name
	setgid(II) set process group ID
	setuid(II) set process user ID
	sfs(VI) structured file scanner
tmheader(VII) TM cover	sheet
	Shell accounting
sa(VIII)	Shell arguments
shift(I) adjust	shell (command interpreter)
	sh(I)
msh(VIII) mini	Shell
	sh(I) shell (command interpreter)
	shift(I) adjust Shell arguments
login(I)	sign onto UNIX
kill(II) send	signa' to a process
signal(II) catch or ignore	signal(II) catch or ignore signals
catsim(VI) phototypesetter	signals
	simulator
sin(III)	sine and cosine functions
size(I)	sin(III) sine and cosine functions
	size of an object file

		size(I) size of an object file
		sky(VI) obtain ephemerides
		sleep(I) suspend execution for an interval
		sleep(II) stop execution for interval
	spline(VI)	smooth curve
	interpolate	
	sno(VI)	Snobol interpreter
		sno(VI) Snobol interpreter
	sort(I)	sort or merge files
		sort(I) sort or merge files
	qsort(III)	sort
	quicker	spawn new process
	fork(II)	speak.m(V) voice synthesizer vocabulary
		speak(VI) word to voice translator
	mknod(II)	special file
	make a directory or a	special file
	mknod(VIII)	specify Fortran file name
	build	spell(I) find spelling errors
		spelling errors
	setfil(I")	Spider interface
		Spider line-printer
	spell(I)	spline(VI) interpolate smooth curve
	find	split(I) split a file into pieces
		split(I) split a file into pieces
	tiu(IV)	lpr(I) spool for line printer
		sqrt(III) square root function
	npr(VI)	square root function
	print file on	startup
		stat(II) get file status
	split(I)	status of open file
		status
	lpr(I)	status
		status
	sqrt(III)	stime(II) set time
		stop execution for interval
	boot procedures(VIII)	stops
	UNIX	strip(I) remove symbols and relocation bits
		sfs(VI) structured file scanner
	fstat(II)	stty(I) set typewriter options
	get	stty(II) set mode of typewriter
	gtty(II)	ibm(VI) submit off-line job to HO IBM 370
	get typewriter	sum(I) sum file
		sum(I) sum file
	ps(I)	du(I) summarize disk usage
	process	sync(VIII) update the
	stat(II)	super block
	get file	super block
		super-block
	sleep(II)	sleep(I) suspend execution for an interval
		su(VIII) become privileged user
	tabs(VII)	switches
	set tab	strip(I) remove symbols and relocation bits
		sync(II) update super-block
	sfs(VI)	sync(VIII) update the super block
		vs(VII) voice synthesizer code
	csw(II)	vs(IV) voice synthesizer interface
	read console	
	strip(I)	
	remove	

speak.m(V)	voice	synthesizer vocabulary
indir(II)	indirect	system call
intro(II)	introduction to	system calls
	check(VIII) file	system consistency check
dump(VIII)	incremental file	system dump
	perror(III)	system error messages
restor(VIII)	incremental file	system restore
	mtab(V) mounted file	system table
	fs(V) format of file	system volume
20boot(VIII)	install new 11/20	system
	mkfs(VIII) construct a file	system
	mount(II) mount file	system
	mount(VIII) mount file	system
	umount(II) dismount file	system
	umount(VIII) dismount file	system
	who(I) who is on the	system
	tabs(VII) set	tab stops
mtab(V)	mounted file system	table
		tabs(VII) set tab stops
	atan(III) arc	tangent function
dump(V)	incremental dump	tape format
	tp(V) DEC/mag	tape formats
	pc(IV) PC-11 paper	tape reader/punch
	rew(I) rewind	tape
	tc(IV)	TC-11/TU56 DECTape
		tc(IV) TC-11/TU56 DECTape
		tee(I) pipe fitting
graf(VI)	draw graph on GSI	terminal
gsi(VI)	interpret funny characters on GSI	terminal
	neqn(I) typeset mathematics on	terminal
plog(VI)	make a graph on the gsi	terminal
	kill(I)	terminate a process
	exit(I)	terminate command file
	exit(II)	terminate process
wait(II)	wait for process to	terminate
	ed(I)	text editor
	nroff(I) format	text
	roff(I) format	text
	troff(I) format	text
	cubic(VI)	three dimensional tic-tac-toe
cubic(VI)	three dimensional	tic-tac-toe
	ttt(VI) the game of	tic-tac-toe
	time(I)	time a command
	profil(II) execution	time profile
ctime(III)	convert date and	time to ASCII
	stime(II) set	time(I) time a command
	times(II) get process	time(II) get date and time
	time(II) get date and	times(II) get process times
		time
		times
		time
		time
		tiu(IV) Spider interface
tmheader(VII)		TM cover sheet
tm(IV)		TM-11/TU-10 magtape interface

goto(I) command	tmg(VI) compiler-compiler
speak(VI) word to voice	tmheader(VII) TM cover sheet
tr(I)	tm(IV) TM-11/TU-10 magtape interface
	tp(I) manipulate DECtape and magtape
	tp(V) DEC/mag tape formats
greek(VII) graphics for extended	transfer
	translator
	transliterate
	tr(I) transliterate
	troff(I) format text
	tss(I) interface to MH-TSS
	ttt(VI) the game of tic-tac-toe
	TTY-37 type-box
	tty(I) get typewriter name
	tty(IV) general typewriter interface
	ttyn(III) return name of current typewriter
	ttys(V) typewriter initialization data
	two files
	two files
	type-box
	typeset mathematics on terminal
	typeset mathematics
	typewriter initialization data
	typewriter interface
	typewriter mode
	typewriter name
	typewriter options
	typewriter status
	typewriter
	typewriter
	typo(I) find possible typos
	typos
	UID
	umount(II) dismount file system
	umount(VIII) dismount file system
	uniq(I) report repeated lines in a file
man(I) run off section of	UNIX manual
boot procedures(VIII)	UNIX startup
login(I) sign onto	UNIX
rm(I) remove	(unlink) files
	unlink(II) remove directory entry
	update super-block
	sync(II)
	sync(VIII)
update(VIII) periodically	update the super block
	update the super block
	update(VIII) periodically update the super block
du(I) summarize disk	usage
getuid(II) get	user identifications
setuid(II) set process	user ID
	user information
	wtmp(V)
mail(I) send mail to another	user login history
su(VIII) become privileged	user
write(I) write to another	user
	utmp(V) user information

speak.m(V)	voice synthesizer	vocabulary
	vs(VII)	voice synthesizer code
	vs(IV)	voice synthesizer interface
	speak.m(V)	voice synthesizer vocabulary
	speak(VI)	voice translator
	word to	volume
fs(V)	format of file system	vs(IV) voice synthesizer interface
		vs(VII) voice synthesizer code
	vt(III)	(vt01) interface
	display	(vt01) interface
	vt(IV)	vt(III) display (vt01) interface
	11/20	vt(IV) 11/20 (vt01) interface
		wait(II) wait for process to terminate
		wait(I) await completion of process
		wait(II) wait for process to terminate
	wc(I)	wc(I) word count
	who(I)	who is on the system
	wc(I)	who(I) who is on the system
	word count	word count
	speak(VI)	word to voice translator
hyphen(VI)	find hyphenated	words
	pwd(I)	working directory name
	chdir(I)	working directory
	change	working directory
	chdir(II)	putchar(III) write character
	change	write(II) write on a file
		write(I) write to another user
		write(I) write to another user
		write(II) write on a file
open(II)	open for reading or	writing
		wtmp(V) user login history
		wump(VI) the game of hunt-the-wumpus
	yacc(VI)	yacc(VI) yet another compiler-compiler
		yacc(VI) yet another compiler-compiler

NAME

ar — archive and library maintainer

SYNOPSIS

ar key *afile* *name* ...

DESCRIPTION

Ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the loader. It can be used, though, for any similar purpose.

Key is one character from the set *drtux*, optionally concatenated with *v*. *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d means delete the named files from the archive file.

r means replace the named files in the archive file. If the archive file does not exist, **r** will create it. If the named files are not in the archive file, they are appended.

t prints a table of contents of the archive file. If no names are given, all files in the archive are tabbed. If names are given, only those files are tabbed.

u is similar to **r** except that only those files that have been modified are replaced. If no names are given, all files in the archive that have been modified will be replaced by the modified version.

x will extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v means verbose. Under the verbose option, *ar* gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. The following abbreviations are used:

c copy

a append

d delete

r replace

x extract

FILES

/tmp/vtm? temporary

SEE ALSO

ld (I), *archive* (V)

BUGS

Option **tv** should be implemented as a table with more information.

There should be a way to specify the placement of a new file in an archive. Currently, it is placed at the end.

Since *ar* has not been rewritten to deal properly with the new file system modes, extracted files have mode 666.

For the same reason, names are only maintained to 8 characters.

NAME
as — assembler

SYNOPSIS
as [-] name ...

DESCRIPTION
As assembles the concatenation of the named files. If the optional first argument - is used, all undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file a.out. It is executable if no errors occurred during the assembly, and if there were no unresolved external references.

FILES
/etc/as2 pass 2 of the assembler
/tmp/atm[1-4]? temporary
a.out object

SEE ALSO
ld(I), nm(I), db(I), a.out(V), 'UNIX Assembler Manual'.

DIAGNOSTICS
When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

-) Parentheses error
-] Parentheses error
- < String not terminated properly
- * Indirection used illegally
- . Illegal assignment to ‘.’
- A Error in address
- B Branch instruction is odd or too remote
- E Error in expression
- F Error in local ('f' or 'b') type symbol
- G Garbage (unknown) character
- I End of file inside an if
- M Multiply defined symbol as label
- O Word quantity assembled at odd address
- P ‘.’ different in pass 1 and 2
- R Relocation error
- U Undefined symbol
- X Syntax error

BUGS
Symbol table overflow is not checked. x errors can cause incorrect line numbers in following diagnostics.

NAME

cat — concatenate and print

SYNOPSIS

cat file ...

DESCRIPTION

Cat reads each file in sequence and writes it on the standard output. Thus:

cat file

is about the easiest way to print a file. Also:

cat file1 file2 >file3

is about the easiest way to concatenate files.

If no input file is given *cat* reads from the standard input file.

If the argument **-** is encountered, *cat* reads from the standard input file.

SEE ALSO

pr(I), cp(I)

DIAGNOSTICS

none; if a file cannot be found it is ignored.

BUGS

cat x y >x and **cat** x y >y cause strange results.

NAME

cc — C compiler

SYNOPSIS

cc [-c] [-p] [-O] [-S] [-P] file ...

DESCRIPTION

Cc is the UNIX C compiler. It accepts three types of arguments:

Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

The following flags are interpreted by *cc*. See *ld (I)* for load-time flags.

- c** Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.
- p** Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls the *monitor* subroutine (III) at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof (I)*.
- O** Invoke the experimental object-code optimizer.
- S** Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.
- P** Run only the macro preprocessor on the named C programs, and leave the output on corresponding files suffixed '.i'.

Other arguments are taken to be either loader flag arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out**.

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/c[01]	compiler
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	runtime startoff for monitoring.
/lib/libc.a	builtin functions, etc.
/lib/liba.a	system library

SEE ALSO

"Programming in C— a tutorial," C Reference Manual, *monitor (III)*, *prof (I)*, *cdb (I)*, *ld (I)*.

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Of these, the most mystifying are from the assembler, in particular "m," which means a multiply-defined external symbol (function or data).

BUGS

NAME

cdb – C debugger

SYNOPSIS

cdb [core [a.out]]

DESCRIPTION

Cdb is a debugger for use with C programs. The first argument is a core-image file; if not given, “core” is used. The second argument is the object program (containing a symbol table); if not given “a.out” is used. An acceptable core and object file must both be present.

Commands to *cdb* consist of an address followed by a single command character. If no address is given the last-printed address is used. An address may be followed by a comma and a number, in which case the command applies to the appropriate number of successive addresses.

Addresses are expressions composed of names, decimal numbers, and octal numbers (which begin with “0”) and separated by “+” and “−”. Evaluation proceeds left-to-right. The construction “name[expression]” assumes that *name* is a pointer to an integer and is equivalent to the contents of the named cell plus twice the expression.

The command characters are:

- / print the addressed words in octal.
- = print the value of the addressed expression.
- ' print the addressed bytes as characters.
- " take the address as a pointer to a sequence of characters, and print the characters up to a null byte.
- & If there is any symbol which has the same value as the address, print the symbol's name.
- \$ print a stack trace of the terminated program. The calls are listed in the order made; the actual arguments to each routine are given in octal.

SEE ALSO

cc (I), db (I), C Reference Manual

BUGS

It's still very feeble, even compared with *db (I)*. The stack trace is also pretty vulnerable to corruption, and often doesn't work.

NAME

chdir — change working directory

SYNOPSIS

chdir *directory*

DESCRIPTION

Directory becomes the new working directory. The process must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *chdir* would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell.

SEE ALSO

sh(I)

BUGS

NAME

chmod - change mode

SYNOPSIS

chmod octal file ...

DESCRIPTION

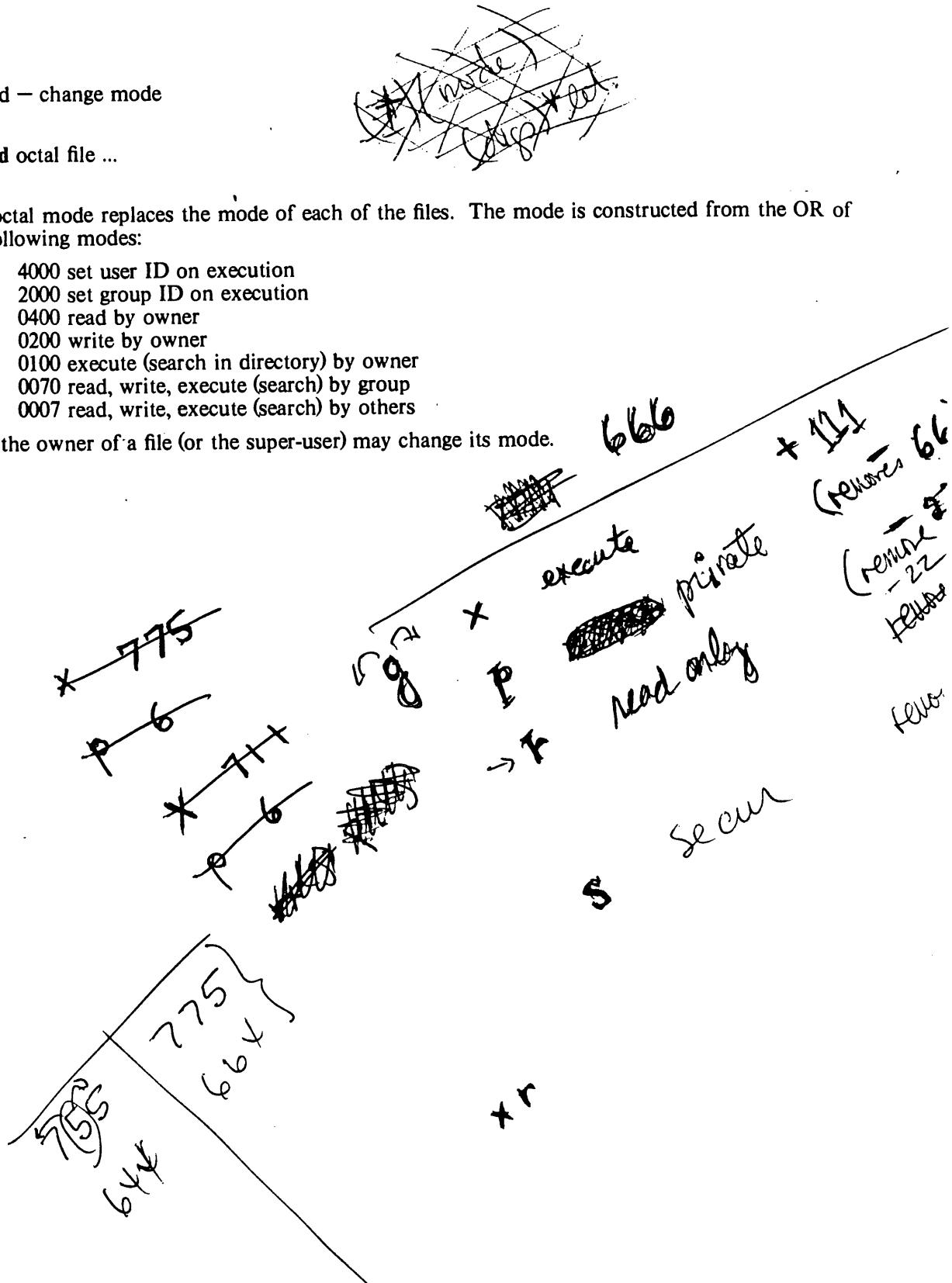
The octal mode replaces the mode of each of the files. The mode is constructed from the OR of the following modes:

- 4000 set user ID on execution
- 2000 set group ID on execution
- 0400 read by owner
- 0200 write by owner
- 0100 execute (search in directory) by owner
- 0070 read, write, execute (search) by group
- 0007 read, write, execute (search) by others

Only the owner of a file (or the super-user) may change its mode.

SEE ALSO

ls(I)

BUGS

NAME

chown — change owner

SYNOPSIS

chown owner file ...

DESCRIPTION

Owner becomes the new owner of the files. The owner may be either a decimal UID or a login name found in the password file.

Only the owner of a file (or the super-user) is allowed to change the owner. Unless it is done by the super-user, the set-user-ID permission bit is turned off as the owner of a file is changed.

FILES

/etc/passwd

BUGS

NAME

cmp — compare two files

SYNOPSIS

cmp file1 file2

DESCRIPTION

The two files are compared for identical contents. Discrepancies are noted by giving the offset and the differing words, all in octal.

SEE ALSO

diff (I), comm (I)

BUGS

If the shorter of the two files is of odd length, *cmp* acts as if a null byte had been appended to it. The *offset* is only a single-precision number.

NAME

comm — print lines common to two files

SYNOPSIS

comm [- [123]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2*, which should be in sort, and produces a three column output: lines only in *file1*; lines only in *file2*; and lines in both files.

The output is written on the standard output.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm -12** prints only the lines common to the two files; **comm -23** prints only lines in the first file but not in the second; **comm -123** is a no-op.

SEE ALSO

cmp (I), **diff (I)**

BUGS

NAME

cp — copy

SYNOPSIS

cp [-t] file1 file2

DESCRIPTION

The first file is copied onto the second. The mode and owner of the target file are preserved if it already existed; the mode of the source file is used otherwise.

If *file2* is a directory, then the target file is a file in that directory with the file-name of *file1*.

No one is quite sure what the flag **-t** does.

SEE ALSO

cat (I), pr (I), mv (I)

BUGS

Copying a file onto itself destroys its contents.

NAME

cref — make cross reference listing

SYNOPSIS

cref [**-acilostux123**] name ...

DESCRIPTION

Cref makes a cross reference listing of program files in assembler or C format. The files named as arguments in the command line are searched for symbols in the appropriate syntax.

The output report is in four columns:

(1)	(2)	(3)	(4)
symbol file	see	text as it appears in file	
		below	

Cref uses either an *ignore* file or an *only* file. If the **-i** option is given, it will take the next available argument to be an *ignore* file name; if the **-o** option is given, the next available argument will be taken as an *only* file name. *Ignore* and *only* files should be lists of symbols separated by new lines. If an *ignore* file is given, all the symbols in that file will be ignored in columns (1) and (3) of the output. If an *only* file is given, only symbols appearing in that file will appear in column (1). Only one of the options **-i** or **-o** may be used. The default setting is **-i**. Assembler predefined symbols or C keywords are ignored.

The **-s** option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, the current function name. The **-l** option causes the line number within the file to be put in column 3.

The **-t** option causes the next available argument to be used as the name of the intermediate temporary file (instead of /tmp/crt??). The file is created and is not removed at the end of the process.

Options:

- a** assembler format (default)
- c** C format input
- i** use *ignore* file (see above)
- l** put line number in col. 3 (instead of current symbol)
- o** use *only* file (see above)
- s** current symbol in col. 3 (default)
- t** user supplied temporary file
- u** print only symbols that occur exactly once
- x** print only C external symbols
- 1** sort output on column 1 (default)
- 2** sort output on column 2
- 3** sort output on column 3

FILES

/tmp/crt??	temporaries
/usr/lib/aign	default assembler <i>ignore</i> file
/usr/lib/cign	default C <i>ignore</i> file
/usr/bin/crpost	post processor
/usr/bin/upost	post processor for -u option
/bin/sort	used to sort temporaries

SEE ALSO

as (I), cc (I)

BUGS

NAME

date — print and set the date

SYNOPSIS

date [mmddhhmm[yy]]

DESCRIPTION

If no argument is given, the current date is printed to the second. If an argument is given, the current date is set. The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

DIAGNOSTICS

'bad conversion' if the argument is syntactically incorrect.

BUGS

NAME

db — debug

SYNOPSIS

db [core [namelist]] [-]

DESCRIPTION

Unlike many debugging packages (including DEC's ODT, on which *db* is loosely based), *db* is not loaded as part of the core image which it is used to examine; instead it examines files. Typically, the file will be either a core image produced after a fault or the binary output of the assembler. *Core* is the file being debugged; if omitted *core* is assumed. *Namelist* is a file containing a symbol table. If it is omitted, the symbol table is obtained from the file being debugged, or if not there from *a.out*. If no appropriate name list file can be found, *db* can still be used but some of its symbolic facilities become unavailable.

For the meaning of the optional third argument, see the last paragraph below.

The format for most *db* requests is an address followed by a one character command. Addresses are expressions built up as follows:

1. A name has the value assigned to it when the input file was assembled. It may be relocatable or not depending on the use of the name during the assembly.
2. An octal number is an absolute quantity with the appropriate value.
3. A decimal number immediately followed by '.' is an absolute quantity with the appropriate value.
4. An octal number immediately followed by **r** is a relocatable quantity with the appropriate value.
5. The symbol **.** indicates the current pointer of *db*. The current pointer is set by many *db* requests.
6. A * before an expression forms an expression whose value is the number in the word addressed by the first expression. A * alone is equivalent to '*.'
7. Expressions separated by + or blank are expressions with value equal to the sum of the components. At most one of the components may be relocatable.
8. Expressions separated by - form an expression with value equal to the difference to the components. If the right component is relocatable, the left component must be relocatable.
9. Expressions are evaluated left to right.

Names for registers are built in:

r0 ... r5
sp
pc
fr0 ... fr5

These may be examined. Their values are deduced from the contents of the stack in a core image file. They are meaningless in a file that is not a core image.

If no address is given for a command, the current address (also specified by ".") is assumed. In general, "." points to the last word or byte printed by *db*.

There are *db* commands for examining locations interpreted as numbers, machine instructions, ASCII characters, and addresses. For numbers and characters, either bytes or words may be examined. The following commands are used to examine the specified file.

- / The addressed word is printed in octal.
- \ The addressed byte is printed in octal.

- " The addressed word is printed as two ASCII characters.
- ' The addressed byte is printed as an ASCII character.
- ` The addressed word is printed in decimal.
- ? The addressed word is interpreted as a machine instruction and a symbolic form of the instruction, including symbolic addresses, is printed. Often, the result will appear exactly as it was written in the source program.
- & The addressed word is interpreted as a symbolic address and is printed as the name of the symbol whose value is closest to the addressed word, possibly followed by a signed offset.
- <nl>(i. e., the character "new line") This command advances the current location counter "." and prints the resulting location in the mode last specified by one of the above requests.
- ^ This character decrements "." and prints the resulting location in the mode last selected one of the above requests. It is a converse to <nl>.
- % Exit.

Odd addresses to word-oriented commands are rounded down. The incrementing and decrementing of "." done by the <nl> and ^ requests is by one or two depending on whether the last command was word or byte oriented.

The address portion of any of the above commands may be followed by a comma and then by an expression. In this case that number of sequential words or bytes specified by the expression is printed. "." is advanced so that it points at the last thing printed.

There are two commands to interpret the value of expressions.

- = When preceded by an expression, the value of the expression is typed in octal. When not preceded by an expression, the value of "." is indicated. This command does not change the value of ".".
- : An attempt is made to print the given expression as a symbolic address. If the expression is relocatable, that symbol is found whose value is nearest that of the expression, and the symbol is typed, followed by a sign and the appropriate offset. If the value of the expression is absolute, a symbol with exactly the indicated value is sought and printed if found; if no matching symbol is discovered, the octal value of the expression is given.

The following command may be used to patch the file being debugged.

- ! This command must be preceded by an expression. The value of the expression is stored at the location addressed by the current value of ".". The opcodes do not appear in the symbol table, so the user must assemble them by hand.

The following command is used after a fault has caused a core image file to be produced.

- \$ causes the fault type and the contents of the general registers and several other registers to be printed both in octal and symbolic format. The values are as they were at the time of the fault.

For some purposes, it is important to know how addresses typed by the user correspond with locations in the file being debugged. The mapping algorithm employed by *db* is non-trivial for two reasons: First, in an *a.out* file, there is a 20(8) byte header which will not appear when the file is loaded into core for execution. Therefore, apparent location 0 should correspond with actual file offset 20. Second, addresses in core images do not correspond with the addresses used by the program because in a core image there is a header containing the system's per-process data for the dumped process, and also because the stack is stored contiguously with the text and data part of the core image rather than at the highest possible locations. *Db* obeys the following rules:

If exactly one argument is given, and if it appears to be an *a.out* file, the 20-byte header is skipped during addressing, i.e., 20 is added to all addresses typed. As a consequence, the header can be examined beginning at location -20.

If exactly one argument is given and if the file does not appear to be an **a.out** file, no mapping is done.

If zero or two arguments are given, the mapping appropriate to a core image file is employed. This means that locations above the program break and below the stack effectively do not exist (and are not, in fact, recorded in the core file). Locations above the user's stack pointer are mapped, in looking at the core file, to the place where they are really stored. The per-process data kept by the system, which is stored in the first part of the core file, cannot currently be examined (except by \$).

If one wants to examine a file which has an associated name list, but is not a core image file, the last argument “—” can be used (actually the only purpose of the last argument is to make the number of arguments not equal to two). This feature is used most frequently in examining the memory file /dev/mem.

SEE ALSO

as (I), core (V), a.out (V), od (I)

DIAGNOSTICS

“File not found” if the first argument cannot be read; otherwise “?”.
)

BUGS

There should be some way to examine the registers and other per-process data in a core image; also there should be some way of specifying double-precision addresses. It does not know yet about shared text segments.

NAME

dc — desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

Dc is an arbitrary precision integer arithmetic package. The overall structure of *dc* is a stacking (reverse Polish) calculator. The following constructions are recognized by the calculator:

number The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number.

+

-

%

The top two values on the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place.

sx The top of the stack is popped and stored into a register named *x*, where *x* may be any character.

lx The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged.

f All values on the stack and in registers are printed.

q exits the program. If executing a string, the recursion level is popped by two.

x treats the top element of the stack as a character string and executes it as a string of *dc* commands.

[...] puts the bracketed ascii string onto the top of the stack.

<x

=x

>x

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation.

v replaces the top element on the stack by its square root.

! interprets the rest of the line as a UNIX command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input.

o The top value on the stack is popped and used as the number radix for further output.

z The stack level is pushed onto the stack.

? A line of input is taken from the input source (usually the console) and executed.

new-line ignored except as the name of a register or to end the response to a ?.

space ignored except as the name of a register or to terminate a number.

If a file name is given, input is taken from that file until end-of-file, then input is taken from the console. An example which prints the first ten values of n! is

[la1+dsa*pla10>xlsx

0sal

lxx

FILES

/etc/msh to implement '!'

DIAGNOSTICS

- (x) ? for unrecognized character x.
- (x) ? for not enough elements on the stack to do what was asked by command x.
- 'Out of space' when the free list is exhausted (too many digits).
- 'Out of headers' for too many numbers being kept around.
- 'Out of pushdown' for too many items on the stack.
- 'Nesting Depth' for too many levels of nested execution.

BUGS

NAME

dd — convert and copy a file

SYNOPSIS

dd [option=value] ...

DESCRIPTION

Dd copies its specified input file to its specified output with possible conversions. The input and output block size may be specified to take advantage of raw physical I/O.

<i>option</i>	<i>values</i>
<i>if</i> =	input file name; standard input is default
<i>of</i> =	output file name; standard output is default
<i>ibs</i> =	input block size (default 512)
<i>obs</i> =	output block size (default 512)
<i>bs</i> =	set both input and output block size, superseding <i>ibs</i> and <i>obs</i> ; also, if no conversion is specified, it is particularly efficient since no copy need be done
<i>cbs=n</i>	conversion buffer size
<i>skip=n</i>	skip <i>n</i> input records before starting copy
<i>count=n</i>	copy only <i>n</i> input records
<i>conv=ascii</i>	convert EBCDIC to ASCII
<i>ebcdic</i>	convert ASCII to EBCDIC
<i>lcase</i>	map alphabetics to lower case
<i>ucase</i>	map alphabetics to upper case
<i>swab</i>	swap every pair of bytes
<i>noerror</i>	do not stop processing on an error
<i>sync</i>	pad every input record to <i>ibs</i>
... , ...	several comma-separated conversions

Where sizes are specified, a number of bytes is expected. A number may end with **k**, **b** or **w** to specify multiplication by 1024, 512, or 2 respectively. Also a pair of numbers may be separated by **x** to indicate a product.

Cbs is used only if *ascii* or *ebcdic* conversion is specified. In the former case *cbs* characters are placed into the conversion buffer, converted to ASCII, and trailing blanks trimmed and new-line added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output record of size *cbs*.

After completion, *dd* reports the number of whole and partial input and output blocks.

For example, to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file *x*:

```
dd if=/dev/rmt0 of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw magtape. *Dd* is quite suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

SEE ALSO

cp (I)

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. It is not clear how this relates to real life.

NAME

diff — differential file comparator

SYNOPSIS

diff [—] *name1* *name2*

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. The normal output contains lines of these forms:

n1 a n3,n4
n1,n2 d n3
n1,n2 c n3,n4

These lines resemble *ed* commands to convert file *name1* into file *name2*. The numbers after the letters pertain to file *name2*. In fact, by exchanging ‘a’ for ‘d’ and reading backward one may ascertain equally how to convert file *name2* into *name1*. As in *ed*, identical pairs where *n1 = n2* or *n3 = n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by ‘*’, then all the lines that are affected in the second file flagged by ‘.’.

Under the — option, the output of *diff* is a script of *a*, *c* and *d* commands for the editor *ed*, which will change the contents of the first file into the contents of the second. In this connection, the following shell program may help maintain multiple versions of a file. Only an ancestral file (\$1) and a chain of version-to-version *ed* scripts (\$2,\$3,...) made by *diff* need be on hand. A ‘latest version’ appears on the standard output.

```
(cat $2 ... $9; echo "1,$p") | ed - $1
```

Diff does an optimal and unfailing job of detecting the file differences, and also reports these differences side-by-side. However, *diff* uses a quadratic algorithm that usually slows to a crawl on 2000-line files.

SEE ALSO

cmp (I), **comm** (I), **ed** (I)

DIAGNOSTICS

‘can’t open input’

‘arg count’

‘jackpot’ — To speed things up, the program uses hashing. You have stumbled on a case where there is a minuscule chance that this has resulted in an unnecessarily long list of differences being published. It’s a curio that we’d like to see.

BUGS

Editing scripts produced under the — option are naive about creating lines consisting of a single ‘.’.

NAME

dsw — delete interactively

SYNOPSIS

dsw [directory]

DESCRIPTION

For each file in the given directory ('.' if not specified) *dsw* types its name. If *y* is typed, the file is deleted; if *x*, *dsw* exits; if new-line, the file is not deleted; if anything else, *dsw* asks again.

SEE ALSO

rm(I)

BUGS

The name *dsw* is a carryover from the ancient past. Its etymology is amusing.

NAME

du — summarize disk usage

SYNOPSIS

du [-s] [-a] [name ...]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each specified directory or file *name*. If *name* is missing, '.' is used.

The optional argument **-s** causes only the grand total to be given. The optional argument **-a** causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

A file which has two links to it is only counted once.

BUGS

Non-directories given as arguments (not under **-a** option) are not listed.

Removable file systems do not work correctly since i-numbers may be repeated while the corresponding files are distinct. *Du* should maintain an i-number list per root directory encountered.

NAME

echo — echo arguments

SYNOPSIS

echo [arg ...]

DESCRIPTION

Echo writes all its arguments in order as a line on the standard output file. It is mainly useful for producing diagnostics in command files.

BUGS

NAME

ed — text editor

SYNOPSIS

ed [—] [name]

DESCRIPTION

Ed is the standard text editor.

If a *name* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited. The optional — simulates an *os* command (see below) which suppresses the printing of character counts by *e*, *r*, and *w* commands.

Ed operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

Commands to *ed* have a simple and regular structure: zero or more *addresses* followed by a single character *command*, possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Every command which requires addresses has default addresses, so that the addresses can often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period '.' alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation. A regular expression is an expression which specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. The regular expressions allowed by *ed* are constructed as follows:

1. An ordinary character (not one of those discussed below) is a regular expression and matches that character.
2. A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
3. A currency symbol '\$' at the end of a regular expression matches the null character at the end of a line.
4. A period '.' matches any character but a new-line character.
5. A regular expression followed by an asterisk '*' matches any number of adjacent occurrences (including zero) of the regular expression it follows.
6. A string of characters enclosed in square brackets '[']' matches any character in the string but no others. If, however, the first character of the string is a circumflex '^' the regular expression matches any character but new-line and the characters in the string.
7. The concatenation of regular expressions is a regular expression which matches the concatenation of the strings matched by the components of the regular expression.
8. The null regular expression standing alone is equivalent to the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see *s* below) to specify a portion of a line which is to be replaced.

If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by '\'. This also applies to the character bounding the regular expression (often '/') and to '\' itself.

Addresses are constructed as follows. To understand addressing in *ed* it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line unaffected by a command; however, the exact effect on the current line by each command is dis-

cussed under the description of the command.

1. The character '.' addresses the current line.
2. The character '^' addresses the line immediately before the current line.
3. The character '\$' addresses the last line of the buffer.
4. A decimal number n addresses the n -th line of the buffer.
5. 'x' addresses the line associated (marked) with the mark name character x which must be a printable character. Lines are marked with the k command described below.
6. A regular expression enclosed in slashes '/' addresses the first line found by searching toward the end of the buffer and stopping at the first line containing a string matching the regular expression. If necessary the search wraps around to the beginning of the buffer.
7. A regular expression enclosed in queries '?' addresses the first line found by searching toward the beginning of the buffer and stopping at the first line found containing a string matching the regular expression. If necessary the search wraps around to the end of the buffer.
8. An address followed by a plus sign '+' or a minus sign '-' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma ','. They may also be separated by a semicolon ;. In this case the current line '.' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches ('/,'?''). The second address of any two-address sequence must correspond to a line following the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, any command may be suffixed by 'p' (for 'print'). In that case, the current line is printed after the command is complete.

(.)a
<text>

The append command reads the given text and appends it after the addressed line. '.' is left on the last line input, if there were any, otherwise at the addressed line. Address '0' is legal for this command; text is placed at the beginning of the buffer.

(...)c
<text>

The change command deletes the addressed lines, then accepts input text which replaces these lines. '.' is left at the last line input; if there were none, it is left at the first line not changed.

(...)d

The delete command deletes the addressed lines from the buffer. The line originally after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

e filename

The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. '.' is set to the last line of the buffer. The number of characters read is typed. 'filename' is remembered for possible use as a default file name in a subsequent *r* or *w* command.

f filename

The filename command prints the currently remembered file name. If 'filename' is given, the currently remembered file name is changed to 'filename'.

(1,\$)g/regular expression/command list

In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with '.' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with '\'. *A*, *i*, and *c* commands and associated input are permitted; the '.' terminating input mode may be omitted if it would be on the last line of the command list. The (global) commands, *g*, and *v*, are not permitted in the command list.

(.)i

<text>

This command inserts the given text before the addressed line. '.' is left at the last line input; if there were none, at the addressed line. This command differs from the *a* command only in the placement of the text.

(.)kx

The mark command associates or marks the addressed line with the single character mark name *x*. The ten most recent mark names are remembered. The current mark names may be printed with the *n* command.

(.,.)l

The list command will print the addressed lines in a way that is unambiguous. Non-graphic characters are printed in octal, prefixed characters are overstruck with a circumflex, and long lines are folded.

(.,.)ma

The move command will reposition the addressed lines after the line addressed by *a*. The last of the moved lines becomes the current line.

n

The *n* command will print the current mark names.

os**ov**

After *os* character counts printed by *e*, *r*, and *w* are suppressed. *Ov* turns them back on.

(.,.)p

The print command prints the addressed lines. '.' is left at the last line printed. The *p* command *may* be placed on the same line after any command.

q

The quit command causes *ed* to exit. No automatic write of a file is done.

(\$)r filename

The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). The remembered file name is not changed unless 'filename' is the very first file name mentioned. Address '0' is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. '.' is left at the last line read in from the file.

(...)s/regular expression/replacement/ or,
(...)s/regular expression/replacement/g

The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator 'g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of '/' to delimit the regular expression and the replacement. '.' is left at the last line substituted.

An ampersand '&' appearing in the replacement is replaced by the regular expression that was matched. The special meaning of '&' in this context may be suppressed by preceding it by '\'.

(1,\$)v/regular expression/command list

This command is the same as the global command except that the command list is executed with '.' initially set to every line *except* those matching the regular expression.

(1,\$)w filename

The write command writes the addressed lines onto the given file. If the file does not exist, it is created mode 666 (readable and writeable by everyone). The remembered file name is *not* changed unless 'filename' is the very first file name mentioned. If no file name is given, the remembered file name, if any, is used (see e and f commands). '.' is unchanged. If the command is successful, the number of characters written is typed.

(\$) =

The line number of the addressed line is typed. '.' is unchanged by this command.

!UNIX command

The remainder of the line after the '!' is sent to UNIX to be interpreted as a command. '.' is unchanged. The entire shell syntax is not recognized. See msh(VII) for the restrictions.

(.+1)<newline>

An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1'; it is useful for stepping through text.

If an interrupt signal (ASCII DEL) is sent, ed will print a '?' and return to its command level.

If invoked with the command name '--', (see init(VII)) ed will sign on with the message 'Editing system' and print '*' as the command level prompt character.

Ed has size limitations on the maximum number of lines that can be edited, on the maximum number of characters in a line, in a global's command list, in a remembered file name, and in the size of the temporary file. The current sizes are: 4000 lines per file, 512 characters per line, 256 characters per global command list, 64 characters per file name, and 64K characters in the temporary file (see BUGS).

FILES

/tmp/etm?, temporary
/etc/msh, to implement the '!' command.

DIAGNOSTICS

?" for errors in commands; 'TMP' for temporary file overflow.

SEE ALSO

A Tutorial Introduction to the ED Text Editor (internal memorandum)

BUGS

The temporary file can grow to no more than 64K bytes.

NAME

eqn — typeset mathematics

SYNOPSIS

eqn [file] ...

DESCRIPTION

Eqn is a troff (I) preprocessor for typesetting mathematics on the Graphics Systems phototypesetter. Usage is almost always

eqn file ... | troff

If no files are specified, *eqn* reads from the standard input. A line beginning with ".EQ" marks the start of an equation; the end of an equation is marked by a line beginning with ".EN". Neither of these lines is altered or defined by *eqn*, so you can define them yourself to get centering, numbering, etc. All other lines are treated as comments, and passed through untouched.

Spaces, tabs, newlines, braces, double quotes, tilde and circumflex are the only delimiters. Braces "{}" are used for grouping. Use tildes "~~" to get extra spaces in an equation.

Subscripts and superscripts are produced with the keywords **sub** and **sup**. Thus $x \sub i$ makes x_i , $a \sub i \sup 2$ produces a_i^2 , and $e \sup \{x \sup 2 + y \sup 2\}$ gives $e^{x^2+y^2}$. Fractions are made with **over**. $a \over b$ is $\frac{a}{b}$ and $1 \over \sqrt{ax \sup 2 + bx + c}$ is $\frac{1}{\sqrt{ax^2+bx+c}}$. **sqrt** makes square roots.

The keywords **from** and **to** introduce lower and upper limits on arbitrary things: $\lim_{n \rightarrow \infty} \sum_0^n x_i$ is made with *lim from {n-> inf} sum from 0 to n x sub i*. Left and right brackets, braces, etc., of the right height are made with **left** and **right**: **left [x sup 2 + y sup 2 over alpha right] ~ = 1** produces $\left[x^2 + \frac{y^2}{\alpha} \right] = 1$. The **right** clause is optional.

Vertical piles of things are made with **pile**, **lpile**, **cpile**, and **rpile**: **pile {a above b above c}** produces $\begin{array}{c} a \\ b \\ c \end{array}$. There can be an arbitrary number of elements in a pile. **lpile** left-justifies, **pile** and **cpile** center, with different vertical spacing, and **rpile** right justifies.

Diacritical marks are made with **dot**, **dotdot**, **hat**, **bar**: $x \dot = f(t)$ **bar** is $\dot{x} = \overline{f(t)}$. Default sizes and fonts can be changed with **size n** and various of **roman**, **italic**, and **bold**.

Keywords like **sum** (Σ) **int** (\int) **inf** (∞) and shorthands like $>=$ (\geq) $->$ (\rightarrow), $!=$ (\neq), are recognized. Spell out Greek letters in the desired case, as in *alpha*, *GAMMA*. Mathematical words like *sin*, *cos*, *log* are made Roman automatically. Troff (I) four-character escapes like **\(bs** (@) can be used anywhere. Strings enclosed in double quotes "..." are passed through untouched.

SEE ALSO

A System for Typesetting Mathematics (Computer Science Technical Report #17, Bell Laboratories, 1974.)

TROFF Users' Manual (internal memorandum)

TROFF Made Trivial (internal memorandum)

troff (I), **neqn** (I)

BUGS

Undoubtedly. Watch out for small or large point sizes — it's tuned too well for size 10. Be cautious if inserting horizontal or vertical motions, and of backslashes in general.

NAME

exit — terminate command file

SYNOPSIS

exit

DESCRIPTION

Exit performs a **seek** to the end of its standard input file. Thus, if it is invoked inside a file of commands, upon return from *exit* the shell will discover an end-of-file and terminate.

SEE ALSO

if (I), goto (I), sh (I)

BUGS

NAME

fc — fortran compiler

SYNOPSIS

fc [-c] sfile1.f ... ofile1 ...

DESCRIPTION

Fc is the UNIX Fortran compiler. It accepts three types of arguments:

Arguments whose names end with '.f' are assumed to be Fortran source program units; they are compiled, and the object program is left on the file sfile1.o (i.e. the file whose name is that of the source with '.o' substituted for '.f').

Other arguments (except for **-c**) are assumed to be either loader flags, or object programs, typically produced by an earlier *fc* run, or perhaps libraries of Fortran-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out**.

The **-c** argument suppresses the loading phase, as does any syntax error in any of the routines being compiled.

The following is a list of differences between *fc* and ANSI standard Fortran (also see the BUGS section):

1. Arbitrary combination of types is allowed in expressions. Not all combinations are expected to be supported at runtime. All of the normal conversions involving integer, real, double precision and complex are allowed.
2. DEC's **implicit** statement is recognized. E.g.: **implicit integer /i-n/**
3. The types doublecomplex, logical*1, integer*1, integer*2 and real*8 (double precision) are supported.
4. & as the first character of a line signals a continuation card.
5. c as the first character of a line signals a comment.
6. All keywords are recognized in lower case.
7. The notion of 'column 7' is not implemented.
8. G-format input is free form— leading blanks are ignored, the first blank after the start of the number terminates the field.
9. A comma in any numeric or logical input field terminates the field.
10. There is no carriage control on output.
11. A sequence of *n* characters in double quotes "" is equivalent to *n h* followed by those characters.
12. In **data** statements, a hollerith string may initialize an array or a sequence of array elements.
13. The number of storage units requested by a binary **read** must be identical to the number contained in the record being read.
14. If the first character in an input file is "#", a preprocessor identical to the C preprocessor is called, which implements "#define" and "#include" preprocessor statements. (See the C reference manual for details.) The preprocessor does not recognize Hollerith strings written with *n h*.

In I/O statements, only unit numbers 0-19 are supported. Unit number *n* refers to file **fortn**; (e.g. unit 9 is file 'fort09'). For input, the file must exist; for output, it will be created. Unit 5 is permanently associated with the standard input file; unit 6 with the standard output file. Also see **setfil** (III) for a way to associate unit numbers with named files.

FILES

file.f	input file
a.out	loaded output
f.tmp[123]	temporary (deleted)
/usr/fort/fcl	compiler proper
/lib/fr0.o	runtime startoff
/lib/filib.a	interpreter library
/lib/libf.a	builtin functions, etc.
/lib/liba.a	system library

SEE ALSO

ANSI standard, *ld* (I) for loader flags

Also see the writeups on the precious few non-standard Fortran subroutines, *ierror* and *setfil* (III)

DIAGNOSTICS

Compile-time diagnostics are given in English, accompanied if possible with the offending line number and source line with an underscore where the error occurred. Runtime diagnostics are given by number as follows:

- 1 invalid log argument
- 2 bad arg count to amod
- 3 bad arg count to atan2
- 4 excessive argument to cabs
- 5 exp too large in cexp
- 6 bad arg count to cmplx
- 7 bad arg count to dim
- 8 excessive argument to exp
- 9 bad arg count to idim
- 10 bad arg count to isign
- 11 bad arg count to mod
- 12 bad arg count to sign
- 13 illegal argument to sqrt
- 14 assigned/computed goto out of range
- 15 subscript out of range
- 16 real**real overflow
- 17 (negative real)**real
- 100 illegal I/O unit number
- 101 inconsistent use of I/O unit
- 102 cannot create output file
- 103 cannot open input file
- 104 EOF on input file
- 105 illegal character in format
- 106 format does not begin with (
- 107 no conversion in format but non-empty list
- 108 excessive parenthesis depth in format
- 109 illegal format specification
- 110 illegal character in input field
- 111 end of format in hollerith specification
- 112 bad argument to setfil
- 120 bad argument to ierror
- 999 unimplemented input conversion

Any of these errors can be caught by the program; see *ierror* (III).

BUGS

The following is a list of those features not yet implemented:

arithmetic statement functions

scale factors on input

Backspace statement.

NAME

fed — edit associative memory for form letter

SYNOPSIS

fed

DESCRIPTION

Fed is used to edit a form letter associative memory file, **form.m**, which consists of named strings. Commands consist of single letters followed by a list of string names separated by a single space and ending with a new line. The conventions of the Shell with respect to '*' and '?' hold for all commands but **m**. The commands are:

e name ...

Fed writes the string whose name is *name* onto a temporary file and executes *ed*. On exit from the *ed* the temporary file is copied back into the associative memory. Each argument is operated on separately. Be sure to give an *ed w* command (without a filename) to rewrite *fed's* temporary file before quitting out of *ed*.

d [name ...]

deletes a string and its name from the memory. When called with no arguments **d** operates in a verbose mode typing each string name and deleting only if a **y** is typed. A **q** response returns to *fed's* command level. Any other response does nothing.

m name1 name2 ...

(move) changes the name of *name1* to *name2* and removes previous string *name2* if one exists. Several pairs of arguments may be given. Literal strings are expected for the names.

n [name ...]

(names) lists the string names in the memory. If called with the optional arguments, it just lists those requested.

p name ...

prints the contents of the strings with names given by the arguments.

q

returns to the system.

c [**p**] [**f**]

checks the associative memory file for consistency and reports the number of free headers and blocks. The optional arguments do the following:

p causes any unaccounted-for string to be printed.

f fixes broken memories by adding unaccounted-for headers to free storage and removing references to released headers from associative memory.

FILES

/tmp/ftmp? temporary
form.m associative memory

SEE ALSO

form(I), ed(I), sh(I)

WARNING

It is legal but unwise to have string names with blanks, ':' or '?' in them.

BUGS

NAME

find — find files

SYNOPSIS

find pathname expression

DESCRIPTION

Find recursively descends the directory hierarchy from *pathname* seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where *+n* means more than *n*, *-n* means less than *n* and *n* means exactly *n*.

- name filename** True if the *filename* argument matches the current file name. Normal *Shell* argument syntax may be used if escaped (watch out for '[', '?' and '*').
 - perm onum** True if the file permission flags exactly match the octal number *onum* (see *chmod(I)*). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat(II)*) become significant and the flags will be compared: *(flags&onum)==onum*.
 - type c** True if the type of the file is *c*, where *c* is **b**, **c**, **d** or **f** for block special file, character special file, directory or plain file.
 - links n** True if the file has *n* links.
 - user uname** True if the file belongs to the user *uname*.
 - group gname** As it is for **-user** so shall it be for **-group** (someday).
 - size n** True if the file is *n* blocks long (512 bytes per block).
 - atime n** True if the file has been accessed in *n* days.
 - mtime n** True if the file has been modified in *n* days.
 - exec command** True if the executed command returns exit status zero (most commands do). The end of the command is punctuated by an escaped semicolon. A command argument '{}' is replaced by the current pathname.
 - ok command** Like **-exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds **y**.
 - print** Always true; causes the current pathname to be printed.
- The primaries may be combined with these operators (ordered by precedence):
- !** prefix *not*
 - a** infix *and*, second operand evaluated only if first is true
 - o** infix *or*, second operand evaluated only if first is false
 - (expression)** parentheses for grouping. (Must be escaped.)
- To remove files named 'a.out' and '*.o' not accessed for a week:
- ```
find / "(" -name a.out -o -name "*.o" ")" -a -atime +7 -a -exec rm {} ";"
```

**FILES**

/etc/passwd

**SEE ALSO**

**sh (I)**, **if(I)**, **file system (V)**

**BUGS**

There is no way to check device type.  
Syntax should be reconciled with *if*.

**NAME**

form — form letter generator

**SYNOPSIS**

form proto arg ...

**DESCRIPTION**

*Form* generates a form letter from a prototype letter, an associative memory, arguments and in a special case, the current date.

If *form* is invoked with the *proto* argument *x*, the associative memory is searched for an entry with name *x* and the contents filed under that name are used as the prototype. If the search fails, the message '[*x*]:' is typed on the console and whatever text is typed in from the console, terminated by two new lines, is used as the prototype. If the prototype argument is missing, '{letter}' is assumed.

Basically, *form* is a copy process from the prototype to the output file. If an element of the form [*n*] (where *n* is a digit from 1 to 9) is encountered, the *n*-th argument is inserted in its place, and that argument is then rescanned. If [0] is encountered, the current date is inserted. If the desired argument has not been given, a message of the form '[*n*]:' is typed. The response typed in then is used for that argument.

If an element of the form [*name*] or {*name*} is encountered, the *name* is looked up in the associative memory. If it is found, the contents of the memory under this *name* replaces the original element (again rescanned). If the *name* is not found, a message of the form '[*name*]:' is typed. The response typed in is used for that element. The response is entered in the memory under the name if the name is enclosed in [ ]. The response is not entered in the memory but is remembered for the duration of the letter if the name is enclosed in {}.

In both of the above cases, the response is typed in by entering arbitrary text terminated by two new lines. Only the first of the two new lines is passed with the text.

If one of the special characters [{}]\ is preceded by a \, it loses its special character.

If a file named 'forma' already exists in the user's directory, 'formb' is used as the output file and so forth to 'formz'.

The file 'form.m' is created if none exists. Because form.m is operated on by the disc allocator, it should only be changed by using fed, the form letter editor, or *form*.

**FILES**

form.m associative memory  
form? output file (read only)

**SEE ALSO**

fed (I), roff (I)

**BUGS**

An unbalanced ] or } acts as an end of file but may add a few strange entries to the associative memory.

**NAME**

goto — command transfer

**SYNOPSIS**

goto label

**DESCRIPTION**

*Goto* is only allowed when the Shell is taking commands from a file. The file is searched from the beginning for a line beginning with ‘:’ followed by one or more spaces followed by the *label*. If such a line is found, the *goto* command returns. Since the read pointer in the command file points to the line after the label, the effect is to cause the Shell to transfer to the labelled line.

‘:’ is a do-nothing command that is ignored by the Shell and only serves to place a label.

**SEE ALSO**

sh(I)

**BUGS**

**NAME**

grep — search a file for a pattern

**SYNOPSIS**

grep [ -v ] [ -b ] [ -c ] [ -n ] expression [ file ] ...

**DESCRIPTION**

*Grep* will search the input files (standard input default) for each line containing the regular expression. Normally, each line found is printed on the standard output. If the **-v** flag is used, all lines but those matching are printed. If the **-c** flag is used, each line printed is preceded by its line number. If the **-n** flag is used, each line is preceded by the name of the input file and its relative line number in that file. If the **-b** flag is used, each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context. If interrupt is hit, the file and line number last searched is printed.

For a complete description of the regular expression, see *ed (I)*. Care should be taken when using the characters \$ \* [ ^ | ( ) and \ in the regular expression as they are also meaningful to the Shell. It is generally necessary to enclose the entire *expression* argument in quotes.

**SEE ALSO**

*ed (I)*, *sh (I)*

**BUGS**

Lines are limited to 512 characters; longer lines are truncated.

**NAME**

if — conditional command

**SYNOPSIS**

if *expr* *command* [ *arg* ... ]

**DESCRIPTION**

If evaluates the expression *expr*, and if its value is true, executes the given *command* with the given arguments.

The following primitives are used to construct the *expr*:

**-r** *file* true if the file exists and is readable.

**-w** *file* true if the file exists and is writable.

**s1 = s2** true if the strings *s1* and *s2* are equal.

**s1 != s2** true if the strings *s1* and *s2* are not equal.

{ *command* } The bracketed command is executed to obtain the exit status. Status zero is considered *true*. The command must not be another *if*.

These primaries may be combined with the following operators:

**!** unary negation operator

**-a** binary *and* operator

**-o** binary *or* operator

( *expr* ) parentheses for grouping.

**-a** has higher precedence than **-o**. Notice that all the operators and flags are separate arguments to *if* and hence must be surrounded by spaces. Notice also that parentheses are meaningful to the Shell and must be escaped.

**SEE ALSO**

sh (I), find (I)

**BUGS**

**NAME**

kill — do in an unwanted process

**SYNOPSIS**

kill —[ signo ] processid ...

**DESCRIPTION**

Kills the specified processes. The processid of each asynchronous process started with '&' is reported by the shell. Processid's can also be found by using *ps* (I).

The killed process must have been started from the same typewriter as the current user, unless he is the super-user.

If a signal number preceded by “—” is given as an argument, that signal is sent instead of *kill* (see *signal* (II)).

**SEE ALSO**

*ps* (I), *sh* (I), *signal* (II)

**BUGS**

Clearly people should only be allowed to kill processes owned by them, and having the same typewriter is neither necessary nor sufficient.

**NAME**

*ld* — link editor

**SYNOPSIS**

*ld* [ **-sulxrnd** ] name ...

**DESCRIPTION**

*Ld* combines several object programs into one; resolves external references; and searches libraries. In the simplest case the names of several object programs are given, and *d* combines them, producing an object module which can be either executed or become the input for a further *ld* run. (In the latter case, the **-r** option must be given to preserve the relocation bits.) The output of *ld* is left on **a.out**. This file is executable only if no errors occurred during the load.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries is important.

*Ld* understands several flag arguments which are written preceded by a '**-**'. Except for **-l**, they should appear before the file names.

- s** 'squash' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by *strip*.
- u** take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.
- l** This option is an abbreviation for a library name. **-l** alone stands for '/lib/liba.a', which is the standard system library for assembly language programs. **-lx** stands for '/lib/libx.a' where *x* is any character. A library is searched when its name is encountered, so the placement of a **-l** is significant.
- x** do not preserve local (non-.globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.
- r** generate relocation bits in the output file so that it can be the subject of another *ld* run. This flag also prevents final definitions from being given to common symbols.
- d** force definition of common storage even if the **-r** flag is present (used for reloc (VIII)).
- n** Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data areas up the the first possible 4K word boundary following the end of the text.

**FILES**

/lib/lib?.a libraries  
a.out output file

**SEE ALSO**

as(I), ar(I)

**BUGS**

**NAME**

**ln** — make a link

**SYNOPSIS**

**ln name1 [ name2 ]**

**DESCRIPTION**

A link is a directory entry referring to a file; the same file (together with its size, all its protection information, etc) may have several links to it. There is no way to distinguish a link to a file from its original directory entry; any changes in the file are effective independently of the name by which the file is known.

*Ln* creates a link to an existing file *name1*. If *name2* is given, the link has that name; otherwise it is placed in the current directory and its name is the last component of *name1*.

It is forbidden to link to a directory or to link across file systems.

**SEE ALSO**

**rm(I)**

**BUGS**

There is nothing particularly wrong with *ln*, but *tp* doesn't understand about links and makes one copy for each name by which a file is known; thus if the tape is extracted several copies are restored and the information that links were involved is lost.

**NAME**

login — sign onto UNIX

**SYNOPSIS**

login [ username ]

**DESCRIPTION**

The *login* command is used when a user initially signs onto UNIX, or it may be used at any time to change from one user to another. The latter case is the one summarized above and described here. See 'How to Get Started' for how to dial up initially.

If *login* is invoked without an argument, it will ask for a user name, and, if appropriate, a password. Echoing is turned off (if possible) during the typing of the password, so it will not appear on the written record of the session.

After a successful login, accounting files are updated and the user is informed of the existence of *mailbox* and message-of-the-day files.

Login is recognized by the Shell and executed directly (without forking).

**FILES**

|               |                    |
|---------------|--------------------|
| /tmp/utmp     | accounting         |
| /usr/adm/wtmp | accounting         |
| mailbox       | mail               |
| /etc/motd     | message-of-the-day |
| /etc/passwd   | password file      |

**SEE ALSO**

init (VIII), getty (VIII), mail (I)

**DIAGNOSTICS**

'login incorrect,' if the name or the password is bad. 'No Shell,' 'cannot open password file,' 'no directory': consult a UNIX programming counselor.

**BUGS**

If the first login is unsuccessful, it tends to go into a state where it won't accept a correct login. Hit EOT and try again.

**NAME**

*lpr* — on line print

**SYNOPSIS**

*lpr* [ − ] [ + ] [ +- ]file ...

**DESCRIPTION**

*Lpr* arranges to have the line printer daemon print the file arguments.

Normally, each file is printed in the state it is found when the line printer daemon reads it. If a particular file argument is preceded by +, or a preceding argument of + has been encountered, then *lpr* will make a copy for the daemon to print. If the file argument is preceded by −, or a preceding argument of − has been encountered, then *lpr* will unlink (remove) the file.

If there are no arguments, then the standard input is read and on-line printed. Thus *lpr* may be used as a filter.

**FILES**

|             |                      |
|-------------|----------------------|
| /usr/lpd/*  | spool area           |
| /etc/passwd | personal ident cards |
| /etc/lpd    | daemon               |

**SEE ALSO**

*lpd* (VIII), *passwd* (V)

**BUGS**

**NAME**

*ls* — list contents of directory

**SYNOPSIS**

*ls* [ **-ltasdruf** ] name ...

**DESCRIPTION**

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents. There are several options:

- l** list in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.
- t** sort by time modified (latest first) instead of by name, as is normal
- a** list all entries; usually those beginning with '.' are suppressed
- s** give size in blocks for each entry
- d** if argument is a directory, list only its name, not its contents (mostly used with **-l** to get status on directory)
- r** reverse the order of sort to get reverse alphabetic or oldest first as appropriate
- u** use time of last access instead of last modification for sorting (**-t**) or printing (**-l**)
- i** print i-number in first column of the report for each file listed
- f** force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **-l**, **-t**, **-s**, and **-r**, and turns on **-a**; the order is the order in which entries appear in the directory.

The mode printed under the **-l** option contains 10 characters which are interpreted as follows: the first character is

- d** if the entry is a directory;
- b** if the entry is a block-type special file;
- c** if the entry is a character-type special file;
- if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

- r** if the file is readable
- w** if the file is writable
- x** if the file is executable
- if the indicated permission is not granted

Finally, the group-execute permission character is given as **s** if the file has set-group-ID mode; likewise the user-execute permission character is given as **s** if the file has set-user-ID mode.

**FILES**

/etc/passwd to get user ID's for *ls -l*.

**BUGS**

**NAME**

mail — send mail to another user

**SYNOPSIS**

mail [ -yn ]  
mail letter person ...  
mail person

**DESCRIPTION**

*Mail* without an argument searches for a file called *mailbox*, prints it if present, and asks if it should be saved. If the answer is *y*, the mail is renamed *mbox*, otherwise it is deleted. *Mail* with a *-y* or *-n* argument works the same way, except that the answer to the question is supplied by the argument.

When followed by the names of a letter and one or more people, the letter is prepended to each person's *mailbox*. When a *person* is specified without a *letter*, the letter is taken from the sender's standard input up to an EOT. Each letter is preceded by the sender's name and a postmark.

A *person* is either a user name recognized by *login*, in which case the mail is sent to the default working directory of that user, or the path name of a directory, in which case *mailbox* in that directory is used.

When a user logs in he is informed of the presence of mail.

**FILES**

|             |                                       |
|-------------|---------------------------------------|
| /etc/passwd | to identify sender and locate persons |
| mailbox     | input mail                            |
| mbox        | saved mail                            |
| /tmp/mtm?   | temp file                             |

**SEE ALSO**

*login* (I)

**BUGS**

**NAME**

**man** — run off section of UNIX manual

**SYNOPSIS**

**man [ section ] [ title ... ]**

**DESCRIPTION**

*Man* is a shell command file that will locate and run off one or more sections of this manual. *Section* is the section number of the manual, as an Arabic not Roman numeral, and is optional. *Title* is one or more section names; these names bear a generally simple relation to the page captions in the manual. If the *section* is missing, 1 is assumed. For example,

**man man**

would reproduce this page.

**FILES**

**/usr/man/man?/\***

**BUGS**

The manual is supposed to be reproducible either on the phototypesetter or on a typewriter. However, on a typewriter some information is necessarily lost.

**NAME**

*mesg* — permit or deny messages

**SYNOPSIS**

*mesg* [ *n* ] [ *y* ]

**DESCRIPTION**

*Mesg* with argument *n* forbids messages via *write* by revoking non-user write permission on the user's typewriter. *Mesg* with argument *y* reinstates permission. All by itself, *mesg* reverses the current permission. In all cases the previous state is reported.

**FILES**

/dev/tty?

**SEE ALSO**

*write* (I)

**DIAGNOSTICS**

'?' if the standard input file is not a typewriter

**BUGS**

**NAME**

**mkdir** — make a directory

**SYNOPSIS**

**mkdir dirname ...**

**DESCRIPTION**

*Mkdir* creates specified directories in mode 777. The standard entries '.' and '..' are made automatically.

**SEE ALSO**

**rmdir(l)**

**BUGS**

**NAME**

**mv** — move or rename a file

**SYNOPSIS**

**mv name1 name2**

**DESCRIPTION**

*Mv* changes the name of *name1* to *name2*. If *name2* is a directory, *name1* is moved to that directory with its original file-name. Directories may only be moved within the same parent directory (just renamed).

If *name2* already exists, it is removed before *name1* is renamed. If *name2* has a mode which forbids writing, *mv* prints the mode and reads the standard input to obtain a line; if the line begins with **y**, the move takes place; if not, *mv* exits.

If *name2* would lie on a different file system, so that a simple rename is impossible, *mv* copies the file and deletes the original.

**BUGS**

It should take a **-f** flag, like *rm*, to suppress the question if the target exists and is not writable.

**NAME**

*neqn* — typeset mathematics on terminal

**SYNOPSIS**

*neqn* [ file ] ...

**DESCRIPTION**

*Neqn* is an *nroff* (I) preprocessor. The input language is the same as that of *eqn* (I). Normal usage is almost always

*neqn* file ... | *nroff*

Output is meant for terminals with forward and reverse capabilities, such as the Model 37 teletype or GSI terminal.

If no arguments are specified, *neqn* reads the standard input, so it may be used as a filter.

**SEE ALSO**

*eqn* (I), *gsi* (VI)

**BUGS**

Because of some interactions with *nroff* there may not always be enough space left before and after lines containing equations.

**NAME**

*nice* — run a command at low priority

**SYNOPSIS**

*nice command [ arguments ]*

**DESCRIPTION**

*Nice* executes *command* at low priority.

**SEE ALSO**

*nohup(I), nice(II)*

**BUGS**

**NAME**

**nm** — print name list

**SYNOPSIS**

**nm [ -cnru ] [ name ]**

**DESCRIPTION**

*Nm* prints the symbol table from the output file of an assembler or loader run. Each symbol name is preceded by its value (blanks if undefined) and one of the letters U (undefined) A (absolute) T (text segment symbol), D (data segment symbol), B (bss segment symbol), or C (common symbol). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

If no file is given, the symbols in **a.out** are listed.

Options are:

- c** list only C-style external symbols, that is those beginning with underscore '\_.'
- n** sort by value instead of by name
- r** sort in reverse order
- u** print only undefined symbols.

**FILES**

**a.out**

**BUGS**

**NAME**

**nohup** — run a command immune to hangups

**SYNOPSIS**

**nohup** *command* [ *arguments* ]

**DESCRIPTION**

*Nohup* executes *command* with hangups, quits and interrupts all ignored.

**SEE ALSO**

**nice(I)**, **signal(II)**

**BUGS**

**NAME**

nroff — format text

**SYNOPSIS**

**nroff [ +n ] [ -n ] [ -nn ] [ -mx ] [ -s ] [ -h ] [ -q ] [ -i ] files**

**DESCRIPTION**

*Nroff* formats text according to control lines embedded in the text files. *Nroff* will read the standard input if no file arguments are given. The non-file option arguments are interpreted as follows:

- +n** Output will commence at the first page whose page number is *n* or larger.
- n** will cause printing to stop after page *n*.
- nn** First generated (not necessarily printed) page is given number *n*; simulates ".pn *n*".
- mx** Prepends a standard macro file; simulates ".so /usr/lib/tmac.x".
- s** Stop prior to each page to permit paper loading. Printing is restarted by typing a 'newline' character.
- h** Spaces are replaced where possible with tabs to speed up output (or reduce the size of the output file).
- q** Prompt names for insertions are not printed and the bell character is sent instead; the insertion is not echoed.
- i** Causes the standard input to be read after the files.

**FILES**

/usr/lib/suftab suffix hyphenation tables  
/tmp/rtm? temporary  
/usr/lib/tmac.? standard macro files

**SEE ALSO**

NROFF User's Manual (internal memorandum).  
neqn (I)

**BUGS**

## REQUEST REFERENCE AND INDEX

| Request Form                                      | Initial Value | If no Argument | Cause Break | Explanation                  |
|---------------------------------------------------|---------------|----------------|-------------|------------------------------|
| <b>I. Page Control</b>                            |               |                |             |                              |
| .pl +N                                            | N=66          | N=66           | no          | Page length.                 |
| .bp +N                                            | N=1           | -              | yes         | Begin page.                  |
| .pn +N                                            | N=1           | ignored        | no          | Page number.                 |
| .po +N                                            | N=0           | N=prev         | no          | Page offset.                 |
| .ne N                                             | -             | N=1            | no          | Need N lines.                |
| .mk                                               | none          | -              | no          | Mark current line.           |
| .rt                                               | -             | -              | no          | Return to marked line.       |
| <b>II. Text Filling, Adjusting, and Centering</b> |               |                |             |                              |
| .br                                               | -             | -              | yes         | Break.                       |
| .fi                                               | fill          | -              | yes         | Fill output lines.           |
| .nf                                               | fill          | -              | yes         | No filling and adjusting.    |
| .ad c                                             | adj,norm      | adjust         | no          | Adjust mode on.              |
| .na                                               | adjust        | -              | no          | No adjusting.                |
| .ce N                                             | off           | N=1            | yes         | Center N input text lines.   |
| <b>III. Line Spacing and Blank Lines</b>          |               |                |             |                              |
| .ls +N                                            | N=1           | N=prev         | no          | Line spacing.                |
| .sp N                                             | -             | N=1            | yes         | Space N lines                |
| .lv N                                             | -             | N=1            | no          | Save N lines                 |
| .sv N                                             | -             | N=1            | no          | "                            |
| .os                                               | -             | -              | no          | Output saved lines.          |
| .ns                                               | space         | -              | no          | No-space mode on.            |
| .rs                                               | -             | -              | no          | Restore spacing.             |
| .xh                                               | off           | -              | no          | Extra-half-line mode on.     |
| <b>IV. Line Length and Indenting</b>              |               |                |             |                              |
| .ll +N                                            | N=65          | N=prev         | no          | Line length.                 |
| .in +N                                            | N=0           | N=prev         | yes         | Indent.                      |
| .ti +N                                            | -             | N=1            | yes         | Temporary indent.            |
| <b>V. Macros, Diversion, and Line Traps</b>       |               |                |             |                              |
| .de xx                                            | -             | ignored        | no          | Define or redefine a macro.  |
| .am xx                                            | -             | ignored        | no          | Append to a macro.           |
| .ds xx                                            | -             | ignored        | no          | Define or redefine string.   |
| .as xx                                            | -             | ignored        | no          | Append to a string.          |
| .rm xx                                            | -             | -              | no          | Remove macro name.           |
| .di xx                                            | -             | end            | no          | Divert output to macro "xx". |
| .da xx                                            | -             | end            | no          | Divert and append to "xx".   |
| .wh -N xx                                         | -             | -              | no          | When; set a line trap.       |
| .ch -N -M                                         | -             | -              | no          | Change trap line.            |
| .ch xx -M                                         | -             | -              | no          | "                            |
| <b>VI. Number Registers</b>                       |               |                |             |                              |
| .nr ab +N -M                                      | -             | -              | no          | Number register.             |
| .nr a +N -M                                       | -             | -              | no          | "                            |
| .nc c \n                                          | \n            | \n             | no          | Number character.            |
| .ar arabic                                        | arabic        | -              | no          | Arabic numbers.              |
| .ro arabic                                        | arabic        | -              | no          | Lower case roman numbers.    |
| .RO arabic                                        | arabic        | -              | no          | Upper case roman numbers.    |

**VII. Input and Output Conventions and Character Translations**

|              |       |       |    |                               |
|--------------|-------|-------|----|-------------------------------|
| .ta N,M,...  | -     | none  | no | Pseudotabs setting.           |
| .tc c        | space | space | no | Tab replacement character.    |
| .lc c        | .     | .     | no | Leader replacement character. |
| .ul n        | -     | N=1   | no | Underline input text lines.   |
| .cc c        | ;     | ;     | no | Basic control character.      |
| .c2 c        | ;     | ;     | no | Nobreak control character.    |
| .ec c        | -     | \     | no | Escape character.             |
| .li N        | -     | N=1   | no | Accept input lines literally. |
| .tr abcd.... | -     | -     | no | Translate on output.          |

**VIII. Hyphenation.**

|       |      |      |    |                                  |
|-------|------|------|----|----------------------------------|
| .nh   | on   | -    | no | No hyphenation.                  |
| .hy   | on   | -    | no | Hyphenate.                       |
| .hc c | none | none | no | Hyphenation indicator character. |

**IX. Three Part Titles.**

|                         |      |        |        |                  |
|-------------------------|------|--------|--------|------------------|
| .tl 'left'center'right' | -    | no     | Title. |                  |
| .lt N                   | N=65 | N=prev | no     | Length of title. |

**X. Output Line Numbering.**

|              |     |       |                                        |
|--------------|-----|-------|----------------------------------------|
| .nm +N M S I | off | no    | Number mode on or off, set parameters. |
| .np M S I    | -   | reset | Number parameters set or reset.        |

**XI. Conditional Input Line Acceptance**

|                 |   |    |                                    |
|-----------------|---|----|------------------------------------|
| .if !N anything | - | no | If true accept line of "anything". |
| .if c anything  | - | no | "                                  |
| .if lc anything | - | no | "                                  |
| .if N anything  | - | no | "                                  |

**XII. Environment Switching.**

|       |     |        |    |                                     |
|-------|-----|--------|----|-------------------------------------|
| .ev N | N=0 | N=prev | no | Environment switched (pushed down). |
|-------|-----|--------|----|-------------------------------------|

**XIII. Insertions from the Standard Input Stream**

|            |   |      |    |              |
|------------|---|------|----|--------------|
| .rd prompt | - | bell | no | Read insert. |
| .ex        | - | -    | no | Exit.        |

**XIV. Input File Switching**

|               |   |    |                                 |
|---------------|---|----|---------------------------------|
| .so filename- | - | no | Switch source file (push down). |
| .nx filename  | - | no | Next file.                      |

**XV. Miscellaneous**

|          |   |   |     |                      |
|----------|---|---|-----|----------------------|
| .tm mesg | - | - | no  | Typewriter message   |
| .ig      | - | - | no  | Ignore.              |
| .fl      | - | - | yes | Flush output buffer. |
| .ab      | - | - | no  | Abort.               |

**NAME**

od — octal dump

**SYNOPSIS**

od [ -abcdho ] [ file ] [ [+ ] offset[ . ][ b ] ]

**DESCRIPTION**

*Od* dumps *file* in one or more formats as selected by the first argument. If the first argument is missing **-o** is default. The meanings of the format argument characters are:

- a interprets words as PDP-11 instructions and dis-assembles the operation code. Unknown operation codes print as ???.
- b interprets bytes in octal.
- c interprets bytes in ascii. Unknown ascii characters are printed as \?.
- d interprets words in decimal.
- h interprets words in hex.
- o interprets words in octal.

The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used. Thus *od* can be used as a filter.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If '.' is appended, the offset is interpreted in decimal. If 'b' is appended, the offset is interpreted in blocks. (A block is 512 bytes.) If the file argument is omitted, the offset argument must be preceded by '+'.

Dumping continues until end-of-file.

**SEE ALSO**

db (I)

**BUGS**

**NAME**

opr - off line print

**SYNOPSIS**

opr [ ---[ *id* ] ] [ - ] [ + ] [ +- ]file ...

**DESCRIPTION**

*Opr* arranges to have the 201, data phone daemon submit a job to the Honeywell 6070 to print the file arguments. Normally, the output appears at the GCOS central site. If the first argument is ---, followed by an optional two-character remote station ID, the output is remoted to that station. If no station ID is given, R1 is assumed. (Station R1 has an IBM 1403 printer.)

Normally, each file is printed in the state it is found when the data phone daemon reads it. If a particular file argument is preceded by +, or a preceding argument of + has been encountered, then *opr* will make a copy for the daemon to print. If the file argument is preceded by -, or a preceding argument of - has been encountered, then *opr* will unlink (remove) the file.

If there are no arguments except for the optional ---, then the standard input is read and off-line printed. Thus *opr* may be used as a filter.

**FILES**

|             |                      |
|-------------|----------------------|
| /usr/dpd/*  | spool area           |
| /etc/passwd | personal ident cards |
| /etc/dpd    | daemon               |

**SEE ALSO**

dpd (VIII), passwd (V)

**BUGS**

- X X Station X  
- V remove  
- W mail

**NAME**

passwd — set login password

**SYNOPSIS**

passwd name password

**DESCRIPTION**

The *password* is placed on the given login name. This can only be done by the person corresponding to the login name or by the super-user. An explicit null argument ("") for the password argument will remove any password from the login name.

**FILES**

/etc/passwd

**SEE ALSO**

login(I), passwd(V), crypt(III)

**BUGS**

**NAME**

pfe — print floating exception

**SYNOPSIS**

**pfe**

**DESCRIPTION**

*Pfe* examines the floating point exception register and prints a diagnostic for the last floating point exception.

**SEE ALSO**

signal (II)

**BUGS**

Since the system does not save the exception register in a core image file, the message refers to the last error encountered by anyone. Floating exceptions are therefore volatile.

**NAME**

**pr** — print file

**SYNOPSIS**

**pr** [ **-h header** ] [ **-n** ] [ **+n** ] [ **-wn** ] [ **-ln** ] [ **-t** ] [ **name . . .** ]

**DESCRIPTION**

*Pr* produces a printed listing of one or more files. The output is separated into pages headed by a date, the name of the file or a header (if any), and the page number. If there are no file arguments, *pr* prints its standard input, and is thus usable as a filter.

Options apply to all following files but may be reset between files:

- n** produce *n*-column output
- +n** begin printing with page *n*
- h** treat the next argument as a header
- wn** for purposes of multi-column output, take the width of the page to be *n* characters instead of the default 72
- ln** take the length of the page to be *n* lines instead of the default 66
- t** do not print the 5-line header or the 5-line trailer normally supplied for each page

If there is a header in force, it is printed in place of the file name. Interconsole messages via write(I) are forbidden during a *pr*.

**FILES**

/dev/tty? to suspend messages.

**SEE ALSO**

cat(I), cp(I)

**DIAGNOSTICS**

none; files not found are ignored

**BUGS**

**NAME**

prof — display profile data

**SYNOPSIS**

prof [ -v ] [ -a ] [ -l ] [ file ]

**DESCRIPTION**

*Prof* interprets the file *mon.out* produced by the *monitor* subroutine. Under default modes, the symbol table in the named object file (*a.out* default) is read and correlated with the *mon.out* profile file. For each external symbol, the percentage or time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **-a** option is used, all symbols are reported rather than just external symbols. If the **-l** option is used, the output is listed by symbol value rather than decreasing percentage. If the **-v** option is used, all printing is suppressed and a profile plot is produced on the 611 display.

In order for the number of calls to a routine to be tallied, the **-p** option of *cc* must have been given when the file containing the routine was compiled.

**FILES**

|          |              |
|----------|--------------|
| mon.out  | for profile  |
| a.out    | for namelist |
| /dev/vt0 | for plotting |

**SEE ALSO**

monitor(III), profil(II), cc(I)

**BUGS**

**NAME**

ps — process status

**SYNOPSIS**

ps [ **a****k****l****x** ]

**DESCRIPTION**

*Ps* prints certain indicia about active processes. The **a** flag asks for information about all processes with teletypes (ordinarily only one's own processes are displayed); **x** asks even about processes with no typewriter; **l** asks for a long listing. Ordinarily only the typewriter number (if not one's own), the process number, and an approximation to the command line are given. If the **k** flag is specified, the special file */dev/rk0* is used in place of */dev/mem*. This is used for post-mortem system debugging.

The long listing is columnar and contains

A number encoding the state (last digit) and flags (first 1 or 2 digits) of the process.

The priority of the process; high numbers mean low priority.

A number related in some unknown way to the scheduling heuristic.

The last character of the control typewriter of the process.

The process unique number (as in certain cults it is possible to kill a process if you know its true name).

The size in blocks of the core image of the process.

The last column if non-blank tells the core address in the system of the event which the process is waiting for; if blank, the process is running.

*Ps* makes an educated guess as to the file name and arguments given when the process was created by examining core memory or the swap area. The method is inherently somewhat unreliable and in any event a process is entitled to destroy this information, so the names cannot be counted on too much.

**FILES**

|          |                   |
|----------|-------------------|
| /unix    | system namelist   |
| /dev/mem | core memory       |
| /dev/rf0 | swap device       |
| /dev/rk0 | optional mem file |

**SEE ALSO**

kill (I)

**BUGS**

The command has assumptions built into it about the number of typewriters that exist and what hardware is used to interface them. It also has built into it the name of the device used for swapping.

**NAME**

**pwd** — print working directory pathname

**SYNOPSIS**

**pwd**

**DESCRIPTION**

*Pwd* prints the pathname of the working (current) directory.

**SEE ALSO**

**chdir (I)**

**BUGS**

The algorithm sometimes fails on crossing mounted file systems.

**NAME**

**rew** — rewind tape

**SYNOPSIS**

**rew [ [ m ]digit ]**

**DESCRIPTION**

*Rew* rewinds DECtape or magtape drives. The digit is the logical tape number, and should range from 0 to 7. If the digit is preceded by *m*, *rew* applies to magtape rather than DECtape. A missing digit indicates drive 0.

**FILES**

/dev/tap?  
/dev/mt?

**BUGS**

**NAME**

**rm** — remove (unlink) files

**SYNOPSIS**

**rm** [ **-f** ] [ **-r** ] name ...

**DESCRIPTION**

*Rm* removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If there is no write permission to a file designated to be removed, *rm* will print the file name, its mode and then read a line from the standard input. If the line begins with *y*, the file is removed, otherwise it is not. The optional argument **-f** prevents this interaction.

If a designated file is a directory, an error comment is printed unless the optional argument **-r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory. To remove directories *per se* see *rmdir(1)*.

**FILES**

/etc/glob to implement the **-r** flag

**SEE ALSO**

*rmdir(1)*

**BUGS**

When *rm* removes the contents of a directory under the **-r** flag, full pathnames are not printed in diagnostics.

**NAME**

**rmdir** — remove directory

**SYNOPSIS**

**rmdir** dir ...

**DESCRIPTION**

*Rmdir* removes (deletes) directories. The directory must be empty (except for the standard entries '.' and '..', which *rmdir* itself removes). Write permission is required in the directory in which the directory appears.

**BUGS**

Needs a **-r** flag. Actually, write permission in the directory's parent is *not* required.

**NAME**

**roff** — format text

**SYNOPSIS**

**roff [ +n ] [ -n ] [ -s ] [ -h ] file ...**

**DESCRIPTION**

*Roff* formats text according to control lines embedded in the text in the given files. Encountering a nonexistent file terminates printing. Incoming interconsole messages are turned off during printing. The optional flag arguments mean:

- +n** Start printing at the first page with number *n*.
- n** Stop printing at the first page numbered higher than *n*.
- s** Stop before each page (including the first) to allow paper manipulation; resume on receipt of an interrupt signal.
- h** Insert tabs in the output stream to replace spaces whenever appropriate.

A Request Summary is attached.

**FILES**

/usr/lib/suftab suffix hyphenation tables  
/tmp/rtm? temporary

**SEE ALSO**

**nroff (I), troff (I)**

**BUGS**

*Roff* is the simplest of the runoff programs, but is virtually undocumented.

## REQUEST SUMMARY

| <i>Request</i> | <i>Break</i> | <i>Initial</i> | <i>Meaning</i>                                                      |
|----------------|--------------|----------------|---------------------------------------------------------------------|
| .ad            | yes          | yes            | Begin adjusting right margins.                                      |
| .ar            | no           | arabic         | Arabic page numbers.                                                |
| .br            | yes          | -              | Causes a line break — the filling of the current line is stopped.   |
| .bl n          | yes          | -              | Insert of n blank lines, on new page if necessary.                  |
| .bp +n         | yes          | n=1            | Begin new page and number it n; no n means '+1'.                    |
| .cc c          | no           | c=             | Control character becomes 'c'.                                      |
| .ce n          | yes          | -              | Center the next n input lines, without filling.                     |
| .de xx         | no           | -              | Define macro named 'xx' (definition ends on line beginning '..').   |
| .ds            | yes          | no             | Double space; same as '.ls 2'.                                      |
| .ef t          | no           | t=""           | Even foot title becomes t.                                          |
| .eh t          | no           | t=""           | Even head title becomes t.                                          |
| .fi            | yes          | yes            | Begin filling output lines.                                         |
| .fo            | no           | t=""           | All foot titles are t.                                              |
| .hc c          | no           | none           | Hyphenation character set to 'c'.                                   |
| .he t          | no           | t=""           | All head titles are t.                                              |
| .hx            | no           | -              | Title lines are suppressed.                                         |
| .hy n          | no           | n=1            | Hyphenation is done, if n=1; and is not done, if n=0.               |
| .ig            | no           | -              | Ignore input lines through a line beginning with '..'.              |
| .in +n         | yes          | -              | Indent n spaces from left margin.                                   |
| .ix +n         | no           | -              | Same as '.in' but without break.                                    |
| .li n          | no           | -              | Literal, treat next n lines as text.                                |
| .ll +n         | no           | n=65           | Line length including indent is n characters.                       |
| .ls +n         | yes          | n=1            | Line spacing set to n lines per output line.                        |
| .m1 n          | no           | n=2            | Put n blank lines between the top of page and head title.           |
| .m2 n          | no           | n=2            | n blank lines put between head title and beginning of text on page. |
| .m3 n          | no           | n=1            | n blank lines put between end of text and foot title.               |
| .m4 n          | no           | n=3            | n blank lines put between the foot title and the bottom of page.    |
| .na            | yes          | no             | Stop adjusting the right margin.                                    |
| .ne n          | no           | -              | Begin new page, if n output lines cannot fit on present page.       |
| .nn +n         | no           | -              | The next n output lines are not numbered.                           |
| .n1            | no           | no             | Number output lines; start with 1 each page.                        |
| .n2 n          | no           | no             | Number output lines; stop numbering if n=0.                         |
| .ni +n         | no           | n=0            | Line numbers are indented n.                                        |
| .nf            | yes          | no             | Stop filling output lines.                                          |
| .nx filename   | -            | -              | Change to input file 'filename'.                                    |
| .of t          | no           | t=""           | Odd foot title becomes t.                                           |
| .oh t          | no           | t=""           | Odd head title becomes t.                                           |
| .pa +n         | yes          | n=1            | Same as '.bp'.                                                      |
| .pl +n         | no           | n=66           | Total paper length taken to be n lines.                             |
| .po +n         | no           | n=0            | Page offset. All lines are preceded by N spaces.                    |
| .ro            | no           | arabic         | Roman page numbers.                                                 |
| .sk n          | no           | -              | Produce n blank pages starting next page.                           |
| .sp n          | yes          | -              | Insert block of n blank lines.                                      |
| .ss            | yes          | yes            | Single space output lines, equivalent to '.ls 1'.                   |
| .ta N M ...    | -            | -              | Pseudotab settings. Initial tab settings are columns 9,17,25,...    |
| .tc c          | no           | c=''           | Tab replacement character becomes 'c'.                              |
| .ti +n         | yes          | -              | Temporarily indent next output line n space.                        |
| .tr abcd..     | no           | -              | Translate a into b, c into d, etc.                                  |
| .ul n          | no           | -              | Underline the letters and numbers in the next n input lines.        |

**NAME**

sh — shell (command interpreter)

**SYNOPSIS**

sh [ name [ arg1 ... [ arg9 ] ] ]

**DESCRIPTION**

*Sh* is the standard command interpreter. It is the program which reads and arranges the execution of the command lines typed by most users. It may itself be called as a command to interpret files of commands. Before discussing the arguments to the Shell used as a command, the structure of command lines themselves will be given.

**Commands.** Each command is a sequence of non-blank command arguments separated by blanks. The first argument specifies the name of a command to be executed. Except for certain types of special arguments discussed below, the arguments other than the command name are passed without interpretation to the invoked command.

If the first argument is the name of an executable file, it is invoked; otherwise the string '/bin/' is prepended to the argument. (In this way most standard commands, which reside in '/bin', are found.) If no such command is found, the string '/usr' is further prepended (to give '/usr/bin/command') and another attempt is made to execute the resulting file. (Certain lesser-used commands live in '/usr/bin'.) If the '/usr/bin' file exists, but is not executable, it is used by the Shell as a command file. That is to say it is executed as though it were typed from the console. If all attempts fail, a diagnostic is printed.

**Command lines.** One or more commands separated by '!' or '^' constitute a chain of *filters*. The standard output of each command but the last is taken as the standard input of the next command. Each command is run as a separate process, connected by pipes (see pipe(II)) to its neighbors. A command line contained in parentheses '(' ) may appear in place of a simple command as a filter.

A *command line* consists of one or more pipelines separated, and perhaps terminated by ';' or '&'. The semicolon designates sequential execution. The ampersand causes the preceding pipeline to be executed without waiting for it to finish. The process id of such a pipeline is reported, so that it may be used if necessary for a subsequent *wait* or *kill*.

**Termination Reporting.** If a command (not followed by '&') terminates abnormally, a message is printed. (All terminations other than exit and interrupt are considered abnormal.) Termination reports for commands followed by '&' are given upon receipt of the first command subsequent to the termination of the command, or when a *wait* is executed. The following is a list of the abnormal termination messages:

- Bus error
- Trace/BPT trap
- Illegal instruction
- IOT trap
- EMT trap
- Bad system call
- Quit
- Floating exception
- Memory violation
- Killed

If a core image is produced, '— Core dumped' is appended to the appropriate message.

**Redirection of I/O.** There are three character sequences that cause the immediately following string to be interpreted as a special argument to the Shell itself. Such an argument may appear anywhere among the arguments of a simple command, or before or after a parenthesized command list, and is associated with that command or command list.

An argument of the form '<arg' causes the file 'arg' to be used as the standard input (file descriptor 0) of the associated command.

An argument of the form '> arg' causes file 'arg' to be used as the standard output (file descriptor 1) for the associated command. 'Arg' is created if it did not exist, and in any case is truncated at the outset.

An argument of the form '>>arg' causes file 'arg' to be used as the standard output for the associated command. If 'arg' did not exist, it is created; if it did exist, the command output is appended to the file.

For example, either of the command lines

```
ls >junk; cat tail >>junk
(ls; cat tail) >junk
```

creates, on file 'junk', a listing of the working directory, followed immediately by the contents of file 'tail'.

Either of the constructs '>arg' or '>>arg' associated with any but the last command of a pipeline is ineffectual, as is '<arg' in any but the first.

In commands called by the Shell, file descriptor 2 refers to the standard output of the Shell before any redirection. Thus filters may write diagnostics to a location where they have a chance to be seen.

**Generation of argument lists.** If any argument contains any of the characters '?', '\*' or '[', it is treated specially as follows. The current directory is searched for files which *match* the given argument.

The character '\*' in an argument matches any string of characters in a file name (including the null string).

The character '?' matches any single character in a file name.

Square brackets '[...]' specify a class of characters which matches any single file-name character in the class. Within the brackets, each ordinary character is taken to be a member of the class. A pair of characters separated by '-' places in the class each character lexically greater than or equal to the first and less than or equal to the second member of the pair.

Other characters match only the same character in the file name.

For example, '\*' matches all file names; '?' matches all one-character file names; '[ab]\*.s' matches all file names beginning with 'a' or 'b' and ending with '.s'; '?[zi-m]' matches all two-character file names ending with 'z' or the letters 'i' through 'm'.

If the argument with '\*' or '?' also contains a '/', a slightly different procedure is used: instead of the current directory, the directory used is the one obtained by taking the argument up to the last '/' before a '\*' or '?'. The matching process matches the remainder of the argument after this '/' against the files in the derived directory. For example: '/usr/dmr/a\*.s' matches all files in directory '/usr/dmr' which begin with 'a' and end with '.s'.

In any event, a list of names is obtained which match the argument. This list is sorted into alphabetical order, and the resulting sequence of arguments replaces the single argument containing the '\*', '[', or '?'. The same process is carried out for each argument (the resulting lists are *not* merged) and finally the command is called with the resulting list of arguments.

For example: directory /usr/dmr contains the files a1.s, a2.s, ..., a9.s. From any directory, the command

```
as /usr/dmr/a?.s
```

calls *as* with arguments /usr/dmr/a1.s, /usr/dmr/a2.s, ... /usr/dmr/a9.s in that order.

**Quoting.** The character '\' causes the immediately following character to lose any special meaning it may have to the Shell; in this way '<', '>', and other characters meaningful to the Shell may be passed as part of arguments. A special case of this feature allows the continuation of commands onto more than one line: a new-line preceded by '\' is translated into a blank.

Sequences of characters enclosed in double ("") or single ('') quotes are also taken literally. For example:

```
ls pr -h "My directory"
```

causes a directory listing to be produced by *ls*, and passed on to *pr* to be printed with the heading 'My directory'. Quotes permit the inclusion of blanks in the heading, which is a single argument to *pr*.

**Argument passing.** When the Shell is invoked as a command, it has additional string processing capabilities. Recall that the form in which the Shell is invoked is

```
sh [name [arg1 ... [arg9]]]
```

The *name* is the name of a file which will be read and interpreted. If not given, this subinstance of the Shell will continue to read the standard input file.

In command lines in the file (not in command input), character sequences of the form '\$*n*', where *n* is a digit, are replaced by the *n*th argument to the invocation of the Shell (arg*n*). '\$0' is replaced by *name*.

**End of file.** An end-of-file in the Shell's input causes it to exit. A side effect of this fact means that the way to log out from UNIX is to type an EOT.

**Special commands.** The following commands are treated specially by the Shell.

*chdir* is done without spawning a new process by executing *sys chdir* (II).

*login* is done by executing /bin/login without creating a new process.

*wait* is done without spawning a new process by executing *sys wait* (II).

*shift* is done by manipulating the arguments to the Shell.

':' is simply ignored.

**Command file errors; interrupts.** Any Shell-detected error, or an interrupt signal, during the execution of a command file causes the Shell to cease execution of that file.

Process that are created with a '&' ignore interrupts. Also if such a process has not redirected its input with a '<', its input is automatically redirected to the zero length file /dev/null.

#### FILES

/etc/glob, which interprets '\*', '?', and '['.  
/dev/null as a source of end-of-file.

#### SEE ALSO

'The UNIX Time-sharing System', which gives the theory of operation of the Shell.  
*chdir* (I), *login* (I), *wait* (I), *shift* (I)

#### BUGS

There is no way to redirect the diagnostic output.

**NAME**

shift — adjust Shell arguments

**SYNOPSIS**

**shift**

**DESCRIPTION**

*Shift* is used in Shell command files to shift the argument list left by 1, so that old \$2 can now be referred to by \$1 and so forth. *Shift* is useful to iterate over several arguments to a command file. For example, the command file

```
: loop
if $1x = x exit
pr -3 $1
shift
goto loop
```

prints each of its arguments in 3-column format.

*Shift* is executed within the Shell.

**SEE ALSO**

sh (I)

**BUGS**

**NAME**

size — size of an object file

**SYNOPSIS**

size [ object ... ]

**DESCRIPTION**

*Size* prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in octal and decimal, of each object-file argument. If no file is specified, **a.out** is used.

**BUGS**

**NAME**

**sleep** — suspend execution for an interval

**SYNOPSIS**

**sleep** *time*

**DESCRIPTION**

*Sleep* will suspend execution for *time* seconds. It is used to execute a command in a certain amount of time as in:

(**sleep** 105; **command**)&

Or to execute a command every so often as in this shell command file:

```
: loop
 command
 sleep 37
 goto loop
```

**SEE ALSO**

**sleep(II)**

**BUGS**

*Time* must be less than 65536 seconds.

**NAME**

sort — sort or merge files

**SYNOPSIS**

sort [ -abdnrtx ] [ +pos [ -pos ] ]... [ -mo ] [ name ]...

**DESCRIPTION**

*Sort* sorts all the named files together and writes the result on the standard output. The name ‘-’ means the standard input. The standard input is also used if no input file names are given. Thus *sort* may be used as a filter.

The default sort key is an entire line. Default ordering is lexicographic in ASCII collating sequence, except that lower-case letters are considered the same as the corresponding upper-case letters. Non-ASCII bytes are ignored. The ordering is affected by the flags **-abdnrt**, one or more of which may appear:

- a Do not map lower case letters.
- b Leading blanks (spaces and tabs) are not included in fields.
- d ‘Dictionary’ order: only letters, digits and blanks are significant in ASCII comparisons.
- n An initial numeric string, consisting of optional minus sign, digits and optionally included decimal point, is sorted by arithmetic value.
- r Reverse the sense of comparisons.
- tx Tab character between fields is x.

Selected parts of the line, specified by **+pos** and **-pos**, may be used as sort keys. **Pos** has the form *m.n*, where *m* specifies a number of fields to skip, and *n* a number of characters to skip further into the next field. A missing *n* is taken to be 0. **+pos** denotes the beginning of the key; **-pos** denotes the first position after the key (end of line by default). The ordering rule may be overridden for a particular key by appending one or more of the flags **abdnrt** to **+pos**.

When no tab character has been specified, a field consists of nonblanks and any preceding blanks. Under the **-b** flag, leading blanks are excluded from a field. When a tab character has been specified, a field is a string ending with a tab character.

When keys are specified, later keys are compared only when all earlier ones compare equal. Lines that compare equal are ordered with all bytes significant.

These flag arguments are also understood:

- m Merge only, the input files are already sorted.
- o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs, except under the merge flag **-m**.

**FILES**

/usr/tmp/stm???

**NAME**

spell — find spelling errors

**SYNOPSIS**

spell file ...

**DESCRIPTION**

*Spell* attacks the same problem as *typo (I)*, but from the opposite direction. It extracts words from the input files and looks them up in *Webster's Seventh Collegiate Dictionary*; any words which appear neither in the dictionary nor in a list of about 2000 words frequently occurring in Bell Laboratories documents are listed on the output file *sp.out*. Words which are reasonable transformations of dictionary entries (e.g. a dictionary entry plus *s*) are so marked; words which could not be found even when transformed are marked with asterisks.

The process takes on the order of 5 to 10 minutes. There is a limit of nine input files.

**FILES**

/usr/lib/w2006, /usr/dict/words, sp.out; spjnkq[123] are temporaries.

**SEE ALSO**

*typo (I)*

**BUGS**

There should be no limit on the number of input files.

More suffixes, and perhaps some prefixes, should be added.

It should be usable as a filter.

**NAME**

split — split a file into pieces

**SYNOPSIS**

split *-n* [ file [ name ] ]

**DESCRIPTION**

*Split* reads file and writes it in *n*-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically. If no output name is given, **x** is default

If no input file is given, or if **-** is given in its stead, then the standard input file is used.

**BUGS**

**NAME**

**strip** — remove symbols and relocation bits

**SYNOPSIS**

**strip name ...**

**DESCRIPTION**

*Strip* removes the symbol table and relocation bits ordinarily attached to the output of the assembler and loader. This is useful to save space after a program has been debugged.

The effect of *strip* is the same as use of the **-s** option of *ld*.

**FILES**

/tmp/stm? temporary file

**SEE ALSO**

**ld(I), as(I)**

**BUGS**

**NAME**

**stty** — set typewriter options

**SYNOPSIS**

**stty** option ...

**DESCRIPTION**

*Stty* will set certain I/O options on the current output typewriter. The option strings are selected from the following set:

|                |                                                                                      |
|----------------|--------------------------------------------------------------------------------------|
| <b>even</b>    | allow even parity                                                                    |
| <b>-even</b>   | disallow even parity                                                                 |
| <b>odd</b>     | allow odd parity                                                                     |
| <b>-odd</b>    | disallow odd parity                                                                  |
| <b>raw</b>     | raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)        |
| <b>-raw</b>    | negate raw mode                                                                      |
| <b>-nl</b>     | allow carriage return for new-line, and output CR-LF for carriage return or new-line |
| <b>nl</b>      | accept only new-line to end lines                                                    |
| <b>echo</b>    | echo back every character typed                                                      |
| <b>-echo</b>   | do not echo characters                                                               |
| <b>lcase</b>   | map upper case to lower case                                                         |
| <b>-lcase</b>  | do not map case                                                                      |
| <b>-tabs</b>   | replace tabs by spaces in output                                                     |
| <b>tabs</b>    | preserve tabs                                                                        |
| <b>delay</b>   | calculate cr, tab, and form-feed delays                                              |
| <b>-delay</b>  | no cr/tab/ff delays                                                                  |
| <b>tdelay</b>  | calculate tab delays                                                                 |
| <b>-tdelay</b> | no tab delays                                                                        |

**SEE ALSO**

**stty (II)**

**BUGS**

There should be 'package' options such as **execuport**, **33**, or **terminet**.

**NAME**

**sum** — sum file

**SYNOPSIS**

**sum name ...**

**DESCRIPTION**

*Sum* sums the contents of the bytes (mod  $2^{16}$ ) of one or more files and prints the answer in octal. A separate sum is printed for each file specified, along with the number of whole or partial 512-byte blocks read.

In practice, *sum* is often used to verify that all of a special file can be read without error.

**BUGS**

**NAME**

tee — pipe fitting

**SYNOPSIS**

tee [ name ... ]

**DESCRIPTION**

*Tee* transcribes the standard input to the standard output and makes copies in the named files.

**BUGS**

**NAME**

time — time a command

**SYNOPSIS**

time command

**DESCRIPTION**

The given command is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command.

The execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on the diagnostic output stream.

**BUGS**

Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

**NAME**

**tp** — manipulate DECtape and magtape

**SYNOPSIS**

**tp [ key ] [ name ... ]**

**DESCRIPTION**

*Tp* saves and restores selected portions of the file system hierarchy on DECtape or mag tape. Its actions are controlled by the *key* argument. The key is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped, restored, or listed.

The function portion of the key is specified by one of the following letters:

- r** The indicated files and directories, together with all subdirectories, are dumped onto the tape. If files with the same names already exist, they are replaced. 'Same' is determined by string comparison, so './abc' can never be the same as '/usr/dmr/abc' even if '/usr/dmr' is the current directory. If no file argument is given, '.' is the default.
- u** updates the tape. **u** is the same as **r**, but a file is replaced only if its modification date is later than the date stored on the tape; that is to say, if it has changed since it was dumped. **u** is the default command if none is given.
- d** deletes the named files and directories from the tape. At least one file argument must be given. This function is not permitted on magtapes.
- x** extracts the named files from the tape to the file system. The owner, mode, and date-modified are restored to what they were when the file was dumped. If no file argument is given, the entire contents of the tape are extracted.
- t** lists the names of all files stored on the tape which are the same as or are hierarchically below the file arguments. If no file argument is given, the entire contents of the tape is listed.

The following characters may be used in addition to the letter which selects the function desired.

- m** Specifies magtape as opposed to DECtape.
- 0,...,7** This modifier selects the drive on which the tape is mounted. For DECtape, 'x' is default; for magtape '0' is the default.
- v** Normally *tp* does its work silently. The **v** (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.
- c** means a fresh dump is being created; the tape directory will be zeroed before beginning. Usable only with **r** and **u**. This option is assumed with magtape since it is impossible to selectively overwrite magtape.
- f** causes new entries on tape to be 'fake' in that no data is present for these entries. Such fake entries cannot be extracted. Usable only with **r** and **u**.
- i** Errors reading and writing the tape are noted, but no action is taken. Normally, errors cause a return to the command level.
- w** causes *tp* to pause before treating each file, type the indicative letter and the file name (as with **v**) and await the user's response. Response **y** means 'yes', so the file is treated. Null response means 'no', and the file does not take part in whatever is being done. Response **x** means 'exit'; the *tp* command terminates immediately. In the **x** function, files previously asked about have been extracted already. With **r**, **u**, and **d** no change has been made to the tape.

**FILES**

/dev/tap?  
/dev/mt?

**DIAGNOSTICS**

Several; the non-obvious one is 'Phase error', which means the file changed after it was selected for dumping but before it was dumped.

**BUGS**

**NAME**

tr — transliterate

**SYNOPSIS**

tr [ -cds ] [ string1 [ string2 ] ]

**DESCRIPTION**

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. If *string2* is short, it is padded with corresponding characters from *string1*. Any combination of the options **-cds** may be used. **-c** complements the set of characters in *string1* with respect to the universe of characters whose ascii codes are 001 through 377 octal. **-d** deletes all input characters in *string1*. **-s** squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

**[a—b]** stands for the string of characters whose ascii codes run from character *a* to character *b*.

**[a\*n]**, where *n* is an integer or empty, stands for *n*-fold repetition of character *a*. *n* is taken to be octal or decimal according as its first digit is or is not zero. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character '\' may be used as in *sh* to remove special meaning from any character in a string. In addition, '\' followed by 1, 2 or 3 octal digits stands for the character whose ascii code is given by those digits.

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabets. The strings are quoted to protect the special characters from interpretation by the Shell; 012 is the ascii code for newline.

```
tr -cs "[A-Z][a-z]" "[\012*]" <file1 >file2
```

**SEE ALSO**

*sh* (I), *ed* (I), *ascii* (VII)

**BUGS**

Won't handle ascii NUL in *string1* or *string2*; always deletes NUL from input.

**NAME**

**troff** — format text

**SYNOPSIS**

**troff [ +n ] [ -n ] [ -nn ] [ -mx ] [ -t ] [ -f ] [ -w ] [ -i ] [ -a ] [ -pn ] files**

**DESCRIPTION**

*Troff* formats text for a Graphic Systems phototypesetter according to control lines embedded in the text files. It reads the standard input if no file arguments are given. The non-file option arguments are interpreted as follows:

- +n** Commence typesetting at the first page numbered *n* or larger.
- n** Stop after page *n*.
- nn** First generated (not necessarily printed) page is given the number *n*; simulates ".pn *n*".
- mx** Prepends a standard macro file; simulates ".so /usr/lib/tmac.x".
- t** Place output on standard output instead of the phototypesetter.
- f** Refrain from feeding out paper and stopping the phototypesetter at the end.
- w** Wait until phototypesetter is available, if currently busy.
- i** Read from standard input after the files have been exhausted.
- a** Send a printable approximation of the results to the standard output.
- pn** Print all characters with point-size *n* while retaining all prescribed spacings and motions.

**FILES**

/usr/lib/suftab suffix hyphenation tables  
/tmp/rtm? temporary  
/usr/lib/tmac.x standard macro files

**SEE ALSO**

TROFF User's Manual (internal memorandum).  
TROFF Made Trivial (internal memorandum).  
nroff (I), eqn (I) catsim (VI)

**BUGS**

**NAME**

tss — interface to MH-TSS

**SYNOPSIS**

tss

**DESCRIPTION**

Tss will call the Honeywell 6070 on the 201 data phone. It will then go into direct access with MH-TSS. Output generated by MH-TSS is typed on the standard output and input requested by MH-TSS is read from the standard input with UNIX typing conventions.

An interrupt signal is transmitted as a 'break' to MH-TSS.

Input lines beginning with '!' are interpreted as UNIX commands. Input lines beginning with '^' are interpreted as commands to the interface routine.

- ~<file insert input from named UNIX file
- ~>file deliver tss output to named UNIX file
- ~p pop the output file
- ~q disconnect from tss (quit)
- ~r file receive from HIS routine csr/daccopy
- ~s file send file to HIS routine csr/daccopy

Ascii files may be most efficiently transmitted using the HIS routine csr/daccopy in this fashion. Bold face text comes from MH-TSS. *Afilename* is the 6070 file to be dealt with; *file* is the UNIX file.

**SYSTEM? csr/daccopy (s) *afilename***  
**Send Encoded File ~s *file***

**SYSTEM? csr/daccopy (r) *afilename***  
**Receive Encoded File ~r *file***

**FILES**

/dev/dn0, /dev/dp0, /etc/msh

**DIAGNOSTICS**

Most often, 'Transmission error on last message.'

**BUGS**

When problems occur, and they often do, *tss* exits rather abruptly.

Stand at UNIX

s = send to 6070

r = receive from 6070

**NAME**

**tty** — get typewriter name

**SYNOPSIS**

**tty**

**DESCRIPTION**

*Tty* gives the name of the user's typewriter in the form '*ttn*' for *n* a digit or letter. The actual path name is then '/dev/ttn'.

**DIAGNOSTICS**

'not a tty' if the standard input file is not a typewriter.

**BUGS**

**NAME**

typo — find possible typos

**SYNOPSIS**

typo [ -1 ] [ -n ] file ...

**DESCRIPTION**

*Typo* hunts through a document for unusual words, typographic errors, and *hapax legomena* and prints them on the standard output.

The words used in the document are printed out in decreasing order of peculiarity along with an index of peculiarity. An index of 10 or more is considered peculiar. Printing of certain very common English words is suppressed.

The statistics for judging words are taken from the document itself, with some help from known statistics of English. The **-n** option suppresses the help from English and should be used if the document is written in, for example, Urdu.

The **-1** option causes the final output to appear in a single column instead of three columns. The normal header and pagination is also suppressed.

Roff (I) and nroff (I) control lines are ignored. Upper case is mapped into lower case. Quote marks, vertical bars, hyphens, and ampersands are stripped from within words. Words hyphenated across lines are put back together.

**FILES**

/tmp/ttmp??  
/usr/lib/salt  
/usr/lib/w2006

**BUGS**

Because of the mapping into lower case and the stripping of special characters, words may be hard to locate in the original text.

The escape sequences of troff (I) are not correctly recognized.

**NAME**

**uniq** — report repeated lines in a file

**SYNOPSIS**

**uniq [ -ude [ +n ] [ -n ] ][ input [ output ] ]**

**DESCRIPTION**

*Uniq* reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. Note that repeated lines must be adjacent in order to be found; see *sort(I)*. If the **-u** flag is used, just the lines that are not repeated in the original file are output. The **-d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **-u** and **-d** mode outputs.

The **-c** option supersedes **-u** and **-d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

**-n** The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

**+n** The first *n* characters are ignored. Fields are skipped before characters.

**SEE ALSO**

*sort (I)*, *comm (I)*

**BUGS**

**NAME**

wait — await completion of process

**SYNOPSIS**

**wait**

**DESCRIPTION**

Wait until all processes started with & have completed, and report on abnormal terminations.

Because sys *wait* must be executed in the parent process, the Shell itself executes *wait*, without creating a new process.

**SEE ALSO**

sh (I)

**BUGS**

After executing *wait* you are committed to waiting until termination, because interrupts and quits are ignored by all processes concerned. The only out, if the process does not terminate, is to *kill* it from another terminal or to hang up.

**NAME**

wc — word count

**SYNOPSIS**

**wc [ -rlwpc ] [ name ... ]**

**DESCRIPTION**

*Wc* counts lines and words in the named files, or in the standard input if no name appears. A word is a maximal string of ascii graphics delimited by spaces, tabs or newlines. Other characters are always ignored.

Any of the following options may appear in any order. When any option other than **-r** appears, only the specified information is reported in the order in which the options occur. Otherwise, printing is as with option **-lw**.

- r** Ignore all *roff*, *nroff* and *troff* control lines, i.e. lines beginning with ‘.’ or ‘’.
- l** Print count of lines.
- w** Print count of words.
- a** Print count of alphanumeric strings, with underscore taken as alphanumeric.
- p** Print count of punctuation strings, i.e. all strings of printing characters other than alphanumerics.
- c** Print count of *roff* control lines, regardless of **-r**.

**DIAGNOSTICS**

‘Cannot open’ for unopenable file.

**BUGS**

**NAME**

**who** — who is on the system

**SYNOPSIS**

**who** [ **who-file** ]

**DESCRIPTION**

*Who*, without an argument, lists the name, typewriter channel, and login time for each current UNIX user.

Without an argument, *who* examines the /tmp/utmp file to obtain its information. If a file is given, that file is examined. Typically the given file will be /tmp/wtmp, which contains a record of all the logins since it was created. Then *who* will list logins, logouts, and crashes since the creation of the wtmp file.

Each login is listed with user name, typewriter name (with '/dev/' suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with 'x' in the place of the device name, and a fossil time indicative of when the system went down.

**FILES**

/tmp/utmp

**SEE ALSO**

login (I), init (VIII)

**BUGS**

**NAME**

**write** — write to another user

**SYNOPSIS**

**write user [ ttyno ]**

**DESCRIPTION**

*Write* copies lines from your typewriter to that of another user. When first called, it sends the message

message from yourname...

The recipient of the message should write back at this point. Communication continues until an end of file is read from the typewriter or an interrupt is sent. At that point *write* writes 'EOT' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *ttyno* argument may be used to indicate the last character of the appropriate typewriter name.

Permission to write may be denied or granted by use of the *mesg* command. At the outset writing is allowed. Certain commands, in particular *r off* and *pr*, disallow messages in order to prevent messy output.

If the character '!' is found at the beginning of a line, *write* calls the mini-shell *msh* to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first write to another user, wait for him to write back before starting to send. Each party should end each message with a distinctive signal ((o) for 'over' is conventional) that the other may reply. (oo) (for 'over and out') is suggested when conversation is about to be terminated.

**FILES**

/tmp/utmp      to find user  
/etc/msh      to execute '!'

**SEE ALSO**

*mesg*(I), *who*(I)

**BUGS**

## INTRODUCTION TO SYSTEM CALLS

Section II of this manual lists all the entries into the system. In most cases two calling sequences are specified, one of which is usable from assembly language, and the other from C. Most of these calls have an error return. From assembly language an erroneous call is always indicated by turning on the c-bit of the condition codes. The presence of an error is most easily tested by the instructions *bcs* and *bcc* ("branch on error set (or clear)"). These are synonyms for the *bcs* and *bcc* instructions.

From C, an error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections specify the details.

In both cases an error number is also available. In assembly language, this number is returned in r0 on erroneous calls. From C, the external variable *errno* is set to the error number. *errno* is not cleared on successful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See *perror (III)*.

The possible error numbers are not recited with each writeup in section II, since many errors are possible for most of the calls. Here is a list of the error numbers, their names inside the system (for the benefit of system-readers), and the messages available using *perror*. A short explanation is also provided.

- |   |                |                                                                                                                                                                                                                                              |
|---|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0 | —              | (unused)                                                                                                                                                                                                                                     |
| 1 | <b>EPERM</b>   | Not owner and not super-user<br>Typically this error indicates an attempt to modify a file in some way forbidden except to its owner. It is also returned for attempts by ordinary users to do things allowed only to the super-user.        |
| 2 | <b>ENOENT</b>  | No such file or directory<br>This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.                                                            |
| 3 | <b>ESRCH</b>   | No such process<br>The process whose number was given to <i>signal</i> does not exist, or is already dead.                                                                                                                                   |
| 4 | —              | (unused)                                                                                                                                                                                                                                     |
| 5 | <b>EIO</b>     | I/O error<br>Some physical I/O error occurred during a <i>read</i> or <i>write</i> . This error may in some cases occur on a call following the one to which it actually applies.                                                            |
| 6 | <b>ENXIO</b>   | No such device or address<br>I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialled in or no disk pack is loaded on a drive. |
| 7 | <b>E2BIG</b>   | Arg list too long<br>An argument list longer than 512 bytes (counting the null at the end of each argument) is presented to <i>exec</i> .                                                                                                    |
| 8 | <b>ENOEXEC</b> | Exec format error<br>A request is made to execute a file which, although it has the appropriate permissions, does not start with one of the magic numbers 407 or 410.                                                                        |
| 9 | <b>EBADF</b>   | Bad file number<br>Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which is open only for writing (resp. reading).                                                                        |

- 10 ECHILD No children  
*Wait* and the process has no living or unwaited-for children.
- 11 EAGAIN No more processes  
In a *fork*, the system's process table is full and no more processes can for the moment be created.
- 12 ENOMEM Not enough core  
During an *exec* or *break*, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments is such as to require more than the existing 8 segmentation registers.
- 13 EACCES Permission denied  
An attempt was made to access a file in a way forbidden by the protection system.
- 14 — (unused)
- 15 ENOTBLK Block device required  
A plain file was mentioned where a block device was required, e.g. in *mount*.
- 16 EBUSY Mount device busy  
An attempt was made to dismount a device on which there is an open file or some process's current directory.
- 17 EEXIST File exists  
An existing file was mentioned in a context in which it should not have, e.g. *link*.
- 18 EXDEV Cross-device link  
A link to a file on another device was attempted.
- 19 ENODEV No such device  
An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only device.
- 20 ENOTDIR Not a directory  
A non-directory was specified where a directory is required, for example in a path name or as an argument to *chdir*.
- 21 EISDIR Is a directory  
An attempt to write on a directory.
- 22 EINVAL Invalid argument  
Some invalid argument: currently, dismounting a non-mounted device, mentioning an unknown signal in *signal*, and giving an unknown request in *stty* to the TIU special file.
- 23 ENFILE File table overflow  
The system's table of open files is full, and temporarily no more *opens* can be accepted.
- 24 EMFILE Too many open files  
Only 10 files can be open per process; this error occurs when the eleventh is opened.
- 25 ENOTTY Not a typewriter  
The file mentioned in *stty* or *gtty* is not a typewriter or one of the other devices to which these calls apply.
- 26 ETXTBSY Text file busy  
An attempt to execute a pure-procedure program which is currently open for writing (or reading!).

- 27 EFBIG File too large  
An attempt to make a file larger than the maximum of 2048 blocks.
- 28 ENOSPC No space left on device  
During a *write* to an ordinary file, there is no free space left on the device.
- 29 ESPIPE Seek on pipe  
A *seek* was issued to a pipe.. This error should also be issued for other non-seekable devices.
- 30 EROFS Read-only file system  
An attempt to modify a file or directory was made on a device mounted read-only.

**NAME**

break — set program break

**SYNOPSIS**

(break = 17.)  
sys break; addr  
char \*brk(addr)  
char \*sbrk(incr)

**DESCRIPTION**

*Break* sets the system's idea of the lowest location not used by the program (called the break) to *addr* (rounded up to the next multiple of 64 bytes). Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

From C, *brk* will set the break to *addr*. The old break is returned.

In the alternate entry *sbrk*, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via *exec* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *break*.

**SEE ALSO**

exec (II), alloc (III)

**DIAGNOSTICS**

The c-bit is set if the program requests more memory than the system limit or if more than 8 segmentation registers would be required to implement the break. From C, -1 is returned for these errors.

**NAME**

chdir — change working directory

**SYNOPSIS**

(chdir = 12.)  
sys chdir; dirname  
chdir(dirname)  
char \*dirname;

**DESCRIPTION**

*Dirname* is the address of the pathname of a directory, terminated by a null byte. *Chdir* causes this directory to become the current working directory.

**SEE ALSO**

chdir(I)

**DIAGNOSTICS**

The error bit (c-bit) is set if the given name is not that of a directory or is not readable. From C, a -1 returned value indicates an error, 0 indicates success.

**NAME**

chmod — change mode of file

**SYNOPSIS**

(chmod = 15.)  
sys chmod; name; mode  
chmod(name, mode)  
char \*name;

**DESCRIPTION**

The file whose name is given as the null-terminated string pointed to by *name* has its mode changed to *mode*. Modes are constructed by ORing together some combination of the following:

4000 set user ID on execution  
2000 set group ID on execution  
0400 read by owner  
0200 write by owner  
0100 execute (search on directory) by owner  
0070 read, write, execute (search) by group  
0007 read, write, execute (search) by others

Only the owner of a file (or the super-user) may change the mode.

**SEE ALSO**

chmod (I)

**DIAGNOSTIC**

Error bit (c-bit) set if *name* cannot be found or if current user is neither the owner of the file nor the super-user. From C, a -1 returned value indicates an error, 0 indicates success.

**NAME**

chown - change owner

**SYNOPSIS**

(chmod = 16.)  
sys chown; name; owner  
chown(name, owner)  
char \*name;

**DESCRIPTION**

The file whose name is given by the null-terminated string pointed to by *name* has its owner changed to *owner* (a numerical user ID). Only the present owner of a file (or the super-user) may donate the file to another user. Changing the owner of a file removes the set-user-ID protection bit unless it is done by the super user.

**SEE ALSO**

chown (I), passwd (V)

**DIAGNOSTICS**

The error bit (c-bit) is set on illegal owner changes. From C a -1 returned value indicates error, 0 indicates success.

**NAME**

close — close a file

**SYNOPSIS**

(close = 6.)  
(file descriptor in r0)  
sys close  
close(fildes)

**DESCRIPTION**

Given a file descriptor such as returned from an *open*, *creat*, or *pipe* call, *close* closes the associated file. A close of all files is automatic on *exit*, but since processes are limited to 15 simultaneously open files, *close* is necessary for programs which deal with many files.

**SEE ALSO**

*creat* (II), *open* (II), *pipe* (II)

**DIAGNOSTICS**

The error bit (c-bit) is set for an unknown file descriptor. From C a -1 indicates an error, 0 indicates success.

**NAME**

`creat` — create a new file

**SYNOPSIS**

```
(creat = 8.)
sys creat; name; mode
(file descriptor in r0)
creat(name, mode)
char *name;
```

**DESCRIPTION**

*Creat* creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*. See *chmod(II)* for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned (in *r0*).

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a *creat*, an error is returned and the program knows that the name is unusable for the moment.

**SEE ALSO**

*write (II)*, *close (II)*, *stat (II)*

**DIAGNOSTICS**

The error bit (c-bit) may be set if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

From C, a -1 return indicates an error.

**NAME**

csw — read console switches

**SYNOPSIS**

(csw = 38.; not in assembler)

sys csw

getcsw( )

**DESCRIPTION**

The setting of the console switches is returned (in r0).

**NAME**

dup — duplicate an open file descriptor

**SYNOPSIS**

(dup = 41.; not in assembler)

(file descriptor in r0)

sys dup

dup(fildes)

int fildes;

**DESCRIPTION**

Given a file descriptor returned from an *open*, *pipe*, or *creat* call, *dup* will allocate another file descriptor synonymous with the original. The new file descriptor is returned in r0.

*Dup* is used more to reassign the value of file descriptors than to genuinely duplicate a file descriptor. Since the algorithm to allocate file descriptors returns the lowest available value, combinations of *dup* and *close* can be used to manipulate file descriptors in a general way. This is handy for manipulating standard input and/or standard output.

**SEE ALSO**

*creat* (II), *open* (II), *close* (II), *pipe* (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if: the given file descriptor is invalid; there are already too many open files. From C, a -1 returned value indicates an error.

**NAME**

`exec` — execute a file

**SYNOPSIS**

```
(exec = 11.
sys exec; name; args
...
name: <...\\0>
...
args: arg1; arg2; ...; 0
arg1: <...\\0>
arg2: <...\\0>
...
execl(name, arg1, arg2, ..., argn, 0)
char *name, *arg1, *arg2, ..., *argn;
execv(name, argv)
char *name;
char *argv[];
```

**DESCRIPTION**

*Exec* overlays the calling process with the named file, then transfers to the beginning of the core image of the file. There can be no return from the file; the calling core image is lost.

Files remain open across *exec* calls. Ignored signals remain ignored across *exec*, but signals that are caught are reset to their default values.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID (The real ID identifies the person using the system; the effective ID determines his access privileges.) *Exec* changes the effective user and group ID to the owner of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The real user ID is not affected.

The form of this call differs somewhat depending on whether it is called from assembly language or C; see below for the C version.

The first argument to *exec* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

Once the called file starts execution, the arguments are available as follows. The stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings. The arguments are placed as high as possible in core.

```
sp→ nargs
 arg1
 ...
 argn
 -1
 arg1: <arg1\\0>
 ...
 argn: <argn\\0>
```

From C, two interfaces are available. *exec*/ is useful when a known file with known arguments is being called; the arguments to *exec*/ are the character strings constituting the file and the arguments; as in the basic call, the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The *execv* version is useful when the number of arguments is unknown in advance; the arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv)
int argc;
char *argv[];
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

*Argv* is not directly usable in another *execv*, since *argv[argc]* is -1 and not 0.

**SEE ALSO**

fork (II)

**DIAGNOSTICS**

If the file cannot be found, if it is not executable, if it does not have a valid header (407 or 410 octal as first word), if maximum memory is exceeded, or if the arguments require more than 512 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. From C the returned value is -1.

**BUGS**

Only 512 characters of arguments are allowed.

**NAME**

exit — terminate process

**SYNOPSIS**

(exit = 1.)

(status in r0)

sys exit

exit(status)

int status;

**DESCRIPTION**

*Exit* is the normal means of terminating a process. *Exit* closes all the process's files and notifies the parent process if it is executing a *wait*. The low byte of r0 (resp. the argument to *exit*) is available as status to the parent process.

This call can never return.

**SEE ALSO**

wait (II)

**DIAGNOSTICS**

None.

**NAME**

fork — spawn new process

**SYNOPSIS**

(fork = 2.)

sys fork

(new process return)

(old process return)

fork()

**DESCRIPTION**

*Fork* is the only way new processes are created. The new process's core image is a copy of that of the caller of *fork*. The only distinction is the return location and the fact that r0 in the old (parent) process contains the process ID of the new (child) process. This process ID is used by *wait*.

The two returning processes share all open files that existed before the call. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

From C, the returned value is 0 in the child process, non-zero in the parent process; however, a return of -1 indicates inability to create a new process.

**SEE ALSO**

wait (II), exec (II)

**DIAGNOSTICS**

The error bit (c-bit) is set in the old process if a new process could not be created because of lack of process space. From C, a return of -1 (not just negative) indicates an error.

**NAME**

fstat — get status of open file

**SYNOPSIS**

```
(fstat = 28.)
(file descriptor in r0)
sys fstat; buf

fstat(fildes, buf)
struct inode buf;
```

**DESCRIPTION**

This call is identical to *stat*, except that it operates on open files instead of files given by name. It is most often used to get the status of the standard input and output files, whose names are unknown.

**SEE ALSO**

stat(II)

**DIAGNOSTICS**

The error bit (c-bit) is set if the file descriptor is unknown; from C, a -1 return indicates an error, 0 indicates success.

**NAME**

**getgid** — get group identifications

**SYNOPSIS**

(*getgid* = 47; not in assembler)

**sys getgid**

**getgid( )**

**DESCRIPTION**

*Getgid* returns a word, the low byte of which contains the real group ID of the current process. The high byte contains the effective group ID of the current process. The real group ID identifies the group of the person who is logged in, in contradistinction to the effective group ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set group ID" mode, to find out who invoked them.

**SEE ALSO**

**setgid (II)**

**DIAGNOSTICS**

**NAME**

getuid — get user identifications

**SYNOPSIS**

(getuid = 24.)

sys getuid

getuid( )

**DESCRIPTION**

*Getuid* returns a word, the low byte of which contains the real user ID of the current process. The high byte contains the effective user ID of the current process. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set user ID" mode, to find out who invoked them.

**SEE ALSO**

setuid (II)

**DIAGNOSTICS**

—

**NAME**

gtty — get typewriter status

**SYNOPSIS**

```
(gtty = 32.)
(file descriptor in r0)
sys gtty; arg
...
arg: .=.+6
gtty(fildes, arg)
int arg[3];
```

**DESCRIPTION**

*Gtty* stores in the three words addressed by *arg* the status of the typewriter whose file descriptor is given in *r0* (resp. given as the first argument). The format is the same as that passed by *stty*.

**SEE ALSO**

stty (II)

**DIAGNOSTICS**

Error bit (c-bit) is set if the file descriptor does not refer to a typewriter. From C, a -1 value is returned for an error, 0, for a successful call.

**NAME**

indir — indirect system call

**SYNOPSIS**

(indir = 0; not in assembler)  
sys indir; syscall

**DESCRIPTION**

The system call at the location *syscall* is executed. Execution resumes after the *indir* call.

The main purpose of *indir* is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If *indir* is executed indirectly, it is a no-op.

**SEE ALSO**

—

**DIAGNOSTICS**

—

**NAME**

kill - send signal to a process

**SYNOPSIS**

(kill = 37.; not in assembler)

(process number in r0)

sys kill; sig

kill(pid, sig);

**DESCRIPTION**

*Kill* sends the signal *sig* to the process specified by the process number in r0. See signal (II) for a list of signals.

The sending and receiving processes must have the same controlling typewriter, otherwise this call is restricted to the super-user.

**SEE ALSO**

signal (II), kill (I)

**DIAGNOSTICS**

The error bit (c-bit) is set if the process does not have the same controlling typewriter and the user is not super-user, or if the process does not exist.

**BUGS**

Equality between the controlling typewriters of the sending and receiving process is neither a necessary nor sufficient condition for allowing the sending of a signal. The correct condition is equality of user IDs.

**NAME**

link — link to a file

**SYNOPSIS**

```
(link = 9.)
sys link; name1; name2
link(name1, name2)
char *name1, *name2;
```

**DESCRIPTION**

A link to *name1* is created; the link has the name *name2*. Either name may be an arbitrary path name.

**SEE ALSO**

link(I), unlink(II)

**DIAGNOSTICS**

The error bit (c-bit) is set when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system. From C, a -1 return indicates an error, a 0 return indicates success.

**NAME**

mknod — make a directory or a special file

**SYNOPSIS**

(mknod = I4.; not in assembler)  
sys mknod; name; mode; addr

mknod(name, mode, addr)  
char \*name;

**DESCRIPTION**

*Mknod* creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory and special file bits) is initialized from *mode*. The first physical address of the file is initialized from *addr*. Note that in the case of a directory, *addr* should be zero. In the case of a special file, *addr* specifies which special file.

*Mknod* may be invoked only by the super-user.

**SEE ALSO**

mkdir (I), mknod (VIII), fs (V)

**DIAGNOSTICS**

Error bit (c-bit) is set if the file already exists or if the user is not the super-user. From C, a -1 value indicates an error.

**NAME**

mount — mount file system

**SYNOPSIS**

```
(mount = 21.)
sys mount; special; name; rwflag
mount(special, name, rwflag)
char *special, *name;
```

**DESCRIPTION**

*Mount* announces to the system that a removable file system has been mounted on the block-structured special file *special*; from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

*Name* must exist already. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

**SEE ALSO**

mount (VIII), umount (II)

**DIAGNOSTICS**

Error bit (c-bit) set if: *special* is inaccessible or not an appropriate file; *name* does not exist; *special* is already mounted; there are already too many file systems mounted.

**NAME**

    nice — set program priority

**SYNOPSIS**

    (nice = 34.)  
    (priority in r0)  
    sys nice  
    nice(priority)

**DESCRIPTION**

The currently executing process is set into the priority specified by *priority*. If *priority* is positive, the priority of the process is below default; if negative the process must be the super-user and its priority is raised. The valid range of *priority* is between 20 and -220. The value of 16 is recommended to users who wish to execute long-running programs without flak from the administration.

The effect of this call is passed to a child process by the *fork* system call. The effect can be cancelled by another call to *nice* with a *priority* of 0.

**SEE ALSO**

    nice(I)

**DIAGNOSTICS**

The error bit (c-bit) is set if the user requests a *priority* outside the range of 0 to 20 and is not the super-user.

**NAME**

**open** — open for reading or writing

**SYNOPSIS**

```
(open = 5.)
sys open; name; mode
open(name, mode)
char *name;
```

**DESCRIPTION**

*Open* opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The returned file descriptor should be saved for subsequent calls to *read*, *write*, and *close*.

**SEE ALSO**

*creat* (II), *read* (II), *write* (II), *close* (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open. From C, a -1 value is returned on an error.

**NAME**

pipe — create a pipe

**SYNOPSIS**

(pipe = 42.)

sys pipe

(read file descriptor in r0)

(write file descriptor in r1)

pipe(fildes)

int fildes[2];

**DESCRIPTION**

The *pipe* system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor returned in r1 (resp. fildes[1]), up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor returned in r0 (resp. fildes[0]) will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent *fork* calls) will pass data through the pipe with *read* and *write* calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) return an end-of-file. Write calls under similar conditions are ignored.

**SEE ALSO**

sh (I), read (II), write (II), fork (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if too many files are already open. From C, a -1 returned value indicates an error.

**NAME**

profil — execution time profile

**SYNOPSIS**

(profil = 44.; not in assembler)  
sys profil; buff; bufsiz; offset; scale  
profil(buff, bufsiz, offset, scale)  
char buff[ ];  
int bufsiz, offset, scale;

**DESCRIPTION**

*Buff* points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 177777(8) gives a 1-1 mapping of pc's to words in *buff*; 77777(8) maps each pair of instruction words together. 2(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is also turned off when an *exec* is executed but remains on in child and parent both after a *fork*.

**SEE ALSO**

monitor (III), prof (I)

**DIAGNOSTICS**

**NAME**

read — read from file

**SYNOPSIS**

```
(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes
read(fildes, buffer, nbytes)
char *buffer;
```

**DESCRIPTION**

A file descriptor is a word returned from a successful *open*, or *pipe* call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned (in *r0*).

If the returned value is 0, then end-of-file has been reached.

**SEE ALSO**

*open* (II), *pipe* (II)

**DIAGNOSTICS**

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the error bit (c-bit) is set. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes*, file descriptor not that of an input file. From C, a -1 return indicates the error.

**NAME**

seek — move read/write pointer

**SYNOPSIS**

(seek = 19.)  
(file descriptor in r0)  
sys seek; offset; ptrname  
seek(fildes, offset, ptrname)

**DESCRIPTION**

The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

- if *ptrname* is 0, the pointer is set to *offset*.
- if *ptrname* is 1, the pointer is set to its current location plus *offset*.
- if *ptrname* is 2, the pointer is set to the size of the file plus *offset*.
- if *ptrname* is 3, 4 or 5, the meaning is as above for 0, 1 and 2 except that the offset is multiplied by 512.

If *ptrname* is 0 or 3, *offset* is unsigned, otherwise it is signed.

**SEE ALSO**

open(II), creat(II)

**DIAGNOSTICS**

The error bit (c-bit) is set for an undefined file descriptor. From C, a -1 return indicates an error.

**NAME**

setgid — set process's group ID

**SYNOPSIS**

(setgid = 46.)  
(group ID in r0)  
sys setgid  
setgid(gid)

**DESCRIPTION**

The group ID of the current process is set to the argument. Both the effective and the real group ID are set. This call is only permitted to the super-user or if the argument is the real group ID.

**SEE ALSO**

getgid(II)

**DIAGNOSTICS**

Error bit (c-bit) is set as indicated; from C, a -1 value is returned.

**NAME**

**setuid** — set process's user ID

**SYNOPSIS**

(**setuid** = 23.)

(user ID in r0)

**sys setuid**

**setuid(uid)**

**DESCRIPTION**

The user ID of the current process is set to the argument. Both the effective and the real user ID are set. This call is only permitted to the super-user or if the argument is the real user ID.

**SEE ALSO**

**getuid(II)**

**DIAGNOSTICS**

Error bit (c-bit) is set as indicated; from C, a -1 value is returned.

**NAME**

signal — catch or ignore signals

**SYNOPSIS**

```
(signal = 48.)
sys signal; sig; label
(old value in r0)
signal(sig, func)
int (*func)();
```

**DESCRIPTION**

When the signal defined by *sig* is sent to the current process, it is to be treated according to *label* (resp. *func*). The following is the list of signals:

- 1 hangup
- 2 interrupt
- 3\* quit
- 4\* illegal instruction
- 5\* trace trap
- 6\* IOT instruction
- 7\* EMT instruction
- 8\* floating point exception
- 9 kill (cannot be caught or ignored)
- 10\* bus error
- 11\* segmentation violation
- 12\* bad argument to sys call

If *label* is 0, the default system action applies to the signal. This is processes termination with or without a core dump. If *label* is odd, the signal is ignored. Any other even *label* specifies an address in the process where an interrupt is simulated. An RTI instruction will return from the interrupt. As a signal is caught, it is reset to 0. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

In C, if *func* is 0 or 1, the action is as described above. If *func* is even, it is assumed to be the address of a function entry point. When the signal occurs, the function will be called. A return from the function will simulate the RTI.

The starred signals in the list above cause a core image if not caught or ignored.

In assembly language, the old value of the signal is returned in r0. In C, that value is returned.

After a *fork*, the child inherits all signals. The *exec* call resets all caught signals to default action.

**SEE ALSO**

kill (I), kill (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if the given signal is out of range. In C, a -1 indicates an error; 0 indicates success.

**NAME**

sleep — stop execution for interval

**SYNOPSIS**

(sleep = 35.; not in assembler)

(seconds in r0)

sys sleep

sleep(seconds)

**DESCRIPTION**

The current process is suspended from execution for the number of seconds specified by the argument.

**SEE ALSO**

sleep (I)

**DIAGNOSTICS**

**NAME**

stat — get file status

**SYNOPSIS**

```
(stat = 18.)
sys stat; name; buf

stat(name, buf)
char *name;
struct inode *buf;
```

**DESCRIPTION**

*Name* points to a null-terminated string naming a file; *buf* is the address of a 36(10) byte buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be readable. After *stat*, *buf* has the following structure (starting offset given in bytes):

```
struct {
 char minor; /* +0: minor device of i-node */
 char major; /* +1: major device */
 int inumber; /* +2 */
 int flags; /* +4: see below */
 char nlinks; /* +6: number of links to file */
 char uid; /* +7: user ID of owner */
 char gid; /* +8: group ID of owner */
 char size0; /* +9: high byte of 24-bit size */
 int size1; /* +10: low word of 24-bit size */
 int addr[8]; /* +12: block numbers or device number */
 int actime[2]; /* +28: time of last access */
 int modtime[2]; /* +32: time of last modification */
};
```

The flags are as follows:

|        |                                                                                                                                    |
|--------|------------------------------------------------------------------------------------------------------------------------------------|
| 100000 | i-node is allocated                                                                                                                |
| 060000 | 2-bit file type:<br>000000 plain file<br>040000 directory<br>020000 character-type special file<br>060000 block-type special file. |
| 010000 | large file                                                                                                                         |
| 004000 | set user-ID on execution                                                                                                           |
| 002000 | set group-ID on execution                                                                                                          |
| 000400 | read (owner)                                                                                                                       |
| 000200 | write (owner)                                                                                                                      |
| 000100 | execute (owner)                                                                                                                    |
| 000070 | read, write, execute (group)                                                                                                       |
| 000007 | read, write, execute (others)                                                                                                      |

**SEE ALSO**

ls (I), fstat (II), fs (V)

**DIAGNOSTICS**

Error bit (c-bit) is set if the file cannot be found. From C, a -1 return indicates an error.

**NAME**

stime — set time

**SYNOPSIS**

(stime = 25.)

(time in r0-r1)

sys stime

stime(tbuf)

int tbuf[2];

**DESCRIPTION**

*Stime* sets the system's idea of the time and date. Time is measured in seconds from 0000 GMT Jan 1 1970. Only the super-user may use this call.

**SEE ALSO**

date (I), time (II), ctime (III)

**DIAGNOSTICS**

Error bit (c-bit) set if user is not the super-user.

**NAME**

stty — set mode of typewriter

**SYNOPSIS**

```
(stty = 31.)
(file descriptor in r0)
sys stty; arg
...
arg: speed; 0; mode
stty(fildes, arg)
int arg[3];
```

**DESCRIPTION**

*Stty* sets mode bits and character speeds for the typewriter whose file descriptor is passed in r0 (resp. is the first argument to the call). First, the system delays until the typewriter is quiescent. Then the speed and general handling of the input side of the typewriter is set from the low byte of the first word of the *arg*, and the speed of the output side is set from the high byte of the first word of the *arg*. The speeds are selected from the following table, which corresponds to the speeds supported by the DH-11 interface. If DC-11, DL-11 or KL-11 interfaces are used, impossible speed changes are ignored.

|    |                   |
|----|-------------------|
| 0  | (turn off device) |
| 1  | 50 baud           |
| 2  | 75 baud           |
| 3  | 110 baud          |
| 4  | 134.5 baud        |
| 5  | 150 baud          |
| 6  | 200 baud          |
| 7  | 300 baud          |
| 8  | 600 baud          |
| 9  | 1200 baud         |
| 10 | 1800 baud         |
| 11 | 2400 baud         |
| 12 | 4800 baud         |
| 13 | 9600 baud         |
| 14 | External A        |
| 15 | External B        |

In the current configuration, only 150 and 300 baud are really supported, in that the code conversion and line control required for 2741's (134.5 baud) must be implemented by the user's program, and the half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied.

The second word of the *arg* is currently unused and is available for expansion.

The third word of the *arg* sets the *mode*. It contains several bits which determine the system's treatment of the typewriter:

|       |                                                                                 |
|-------|---------------------------------------------------------------------------------|
| 10000 | <del>no delays after tabs (e.g. TN 300)</del>                                   |
| 200   | even parity allowed on input (e. g. for M37s)                                   |
| 100   | odd parity allowed on input                                                     |
| 040   | raw mode: wake up on all characters                                             |
| 020   | map CR into LF; echo LF or CR as CR-LF                                          |
| 010   | echo (full duplex)                                                              |
| 004   | map upper case to lower on input (e. g. M33)                                    |
| 002   | echo and print tabs as spaces                                                   |
| 001   | <del>inhibit all function delays (e. g. CRTs)</del> <i>hangup on first char</i> |

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed back immediately to the program. No erase or kill processing is done; the end-of-file character (EOT), the interrupt character (DELETE) and the quit character (FS) are not treated specially.

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (used for GE TermiNet 300's and other terminals without the newline function).

**SEE ALSO**

stty (I), stty (II)

**DIAGNOSTICS**

The error bit (c-bit) is set if the file descriptor does not refer to a typewriter. From C, a negative value indicates an error.

**NAME**

sync — update super-block

**SYNOPSIS**

(sync = 36.; not in assembler)

sys sync

**DESCRIPTION**

*Sync* causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *check*, *df*, etc. It is mandatory before a boot.

**SEE ALSO**

sync (VIII)

**DIAGNOSTICS**

TIME(II)

3/15/72

TIME(II)

**NAME**

time — get date and time

**SYNOPSIS**

(time = 13.)

sys time

time(tvec)

int tvec[2];

**DESCRIPTION**

*Time* returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. From *as*, the high order word is in the r0 register and the low order is in r1. From C, the user-supplied vector is filled in.

**SEE ALSO**

date(I), stime(II), ctime(III)

**DIAGNOSTICS**

none

**NAME**

times — get process times

**SYNOPSIS**

(times = 43; not in assembler)  
sys times; buffer

times(buffer)  
struct tbuffer \*buffer;

**DESCRIPTION**

*Times* returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/60 seconds.

After the call, the buffer will appear as follows:

```
struct tbuffer {
 int proc_user_time;
 int proc_system_time;
 int child_user_time[2];
 int child_system_time[2];
};
```

The children times are the sum of the children's process times and their children's times.

**SEE ALSO**

time(I)

**DIAGNOSTICS**

—

**BUGS**

The process times should be 32 bits as well.

**NAME**

umount — dismount file system

**SYNOPSIS**

(umount = 22.)

sys umount; special

**DESCRIPTION**

*Umount* announces to the system that special file *special* is no longer to contain a removable file system. The file associated with the special file reverts to its ordinary interpretation (see *mount* ).

**SEE ALSO**

umount (VIII), mount (II)

**DIAGNOSTICS**

Error bit (c-bit) set if no file system was mounted on the special file or if there are still active files on the mounted file system.

**NAME**

**unlink** — remove directory entry

**SYNOPSIS**

```
(unlink = 10.)
sys unlink; name
unlink(name)
char *name;
```

**DESCRIPTION**

*Name* points to a null-terminated string. *Unlink* removes the entry for the file pointed to by *name* from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

**SEE ALSO**

**rm** (I), **rmdir** (I), **link** (II)

**DIAGNOSTICS**

The error bit (c-bit) is set to indicate that the file does not exist or that its directory cannot be written. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user). From C, a -1 return indicates an error.

**NAME**

wait — wait for process to die

**SYNOPSIS**

(wait = 7.)  
sys wait  
wait(status)  
int \*status;

Wait will die when wait(?) returns.  
Wait does wait(?) return?

**DESCRIPTION**

*Wait* causes its caller to delay until one of its child processes terminates. If any child has died since the last *wait*, return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of -1 returned). In the case of several children several *wait* calls are needed to learn of all the deaths.

If no error is indicated on return, the r1 high byte (resp. the high byte stored into *status*) contains the low byte of the child process r0 (resp. the argument of *exit*) when it terminated. The r1 (resp. *status*) low byte contains the termination status of the process. See signal (II) for a list of termination statuses (signals); 0 status indicates normal termination. If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

**SEE ALSO**

exit (II), fork (II), signal (II), kill (?)

**DIAGNOSTICS**

The error bit (c-bit) is set if there are no children not previously waited for. From C, a returned value of -1 indicates an error.

**NAME**

write — write on a file

**SYNOPSIS**

```
(write = 4.)
(file descriptor in r0)
sys write; buffer; nbytes
write(fildes, buffer, nbytes)
char *buffer;
```

**DESCRIPTION**

A file descriptor is a word returned from a successful *open*, *creat* or *pipe* call.

*Buffer* is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned (in *r0*). It should be regarded as an error if this is not the same as requested.

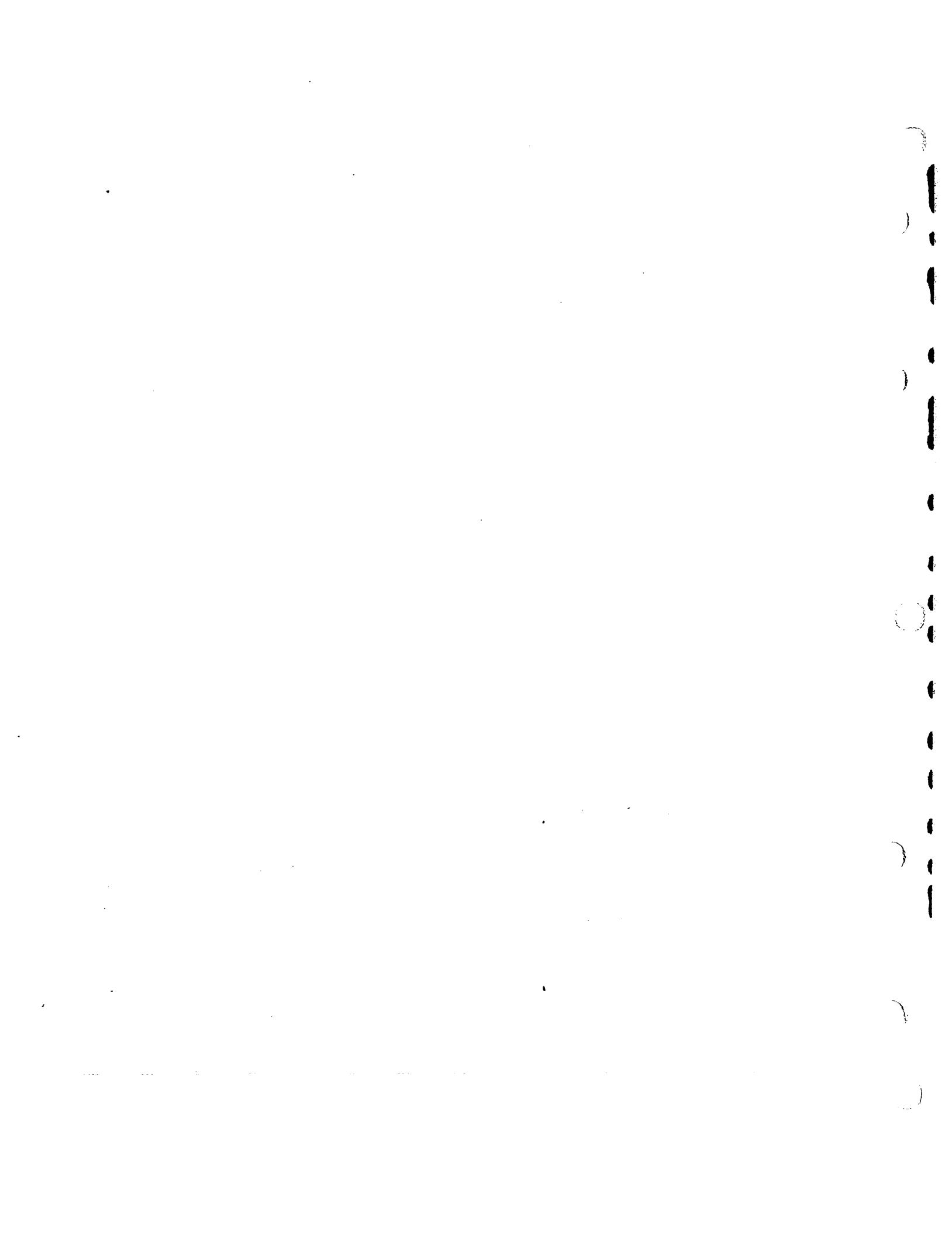
Writes which are multiples of 512 characters long and begin on a 512-byte boundary are more efficient than any others.

**SEE ALSO**

*creat(II)*, *open(II)*, *pipe(II)*

**DIAGNOSTICS**

The error bit (c-bit) is set on an error: bad descriptor, buffer address, or count; physical I/O errors. From C, a returned value of -1 indicates an error.



**NAME**

alloc — core allocator

**SYNOPSIS**

**char \*alloc(size)**

**free(ptr)**

**char \*ptr;**

**DESCRIPTION**

*Alloc* and *free* provide a simple general-purpose core management package. *Alloc* is given a size in bytes; it returns a pointer to an area at least that size which is even and hence can hold an object of any type. The argument to *free* is a pointer to an area previously allocated by *alloc*; this space is made available for further allocation.

Needless to say, grave disorder will result if the space assigned by *alloc* is overrun or if some random number is handed to *free*.

The routine uses a first-fit algorithm which coalesces blocks being freed with other blocks already free. It calls *sbrk* (see *break (II)*) to get more core from the system when there is no suitable space already free, and writes "Out of space" on the standard output, then exits, if that fails.

The external variable *slop* (which is 2 if not set) is a number such that if *n* bytes are requested, and if the first free block of size at least *n* is no larger than *n+slop*, then the whole block will be allocated instead of being split up. Larger values of *slop* tend to reduce fragmentation at the expense of unused space in the allocated blocks.

**DIAGNOSTICS**

"Out of space" if it needs core and can't get it.

**BUGS**

**NAME**

atan — arc tangent function

**SYNOPSIS**

```
jsr r5,atan[2]
double atan(x)
double x;
double atan2(x, y)
double x, y;
```

**DESCRIPTION**

The *atan* entry returns the arc tangent of fr0 in fr0; from C, the arc tangent of *x* is returned. The range is  $-\pi/2$  to  $\pi/2$ . The *atan2* entry returns the arc tangent of fr0/fr1 in fr0; from C, the arc tangent of *x/y* is returned. The range is  $-\pi$  to  $\pi$ .

**DIAGNOSTIC**

There is no error return.

**BUGS**

**NAME**

atof — ascii to floating

**SYNOPSIS**

```
double atof(nptr)
char *nptr;
```

**DESCRIPTION**

*Atof* converts a string to a floating number. *Nptr* should point to a string containing the number; the first unrecognized character ends the number.

The only numbers recognized are: an optional minus sign followed by a string of digits optionally containing one decimal point, then followed optionally by the letter e followed by a signed integer.

**DIAGNOSTICS**

There are none; overflow results in a very large number and garbage characters terminate the scan.

**BUGS**

The routine should accept initial +, initial blanks, and E for e. Overflow should be signalled.

**NAME**

crypt — password encoding

**SYNOPSIS**

```
mov $key,r0
jsr pc,crypt
char *crypt(key)
char *key;
```

**DESCRIPTION**

On entry, r0 should point to a string of characters terminated by an ASCII NULL. The routine performs an operation on the key which is difficult to invert (i.e. encrypts it) and leaves the resulting eight bytes of ASCII alphanumerics in a global cell called "word".

From C, the *key* argument is a string and the value returned is a pointer to the eight-character encrypted password.

Login uses this result as a password.

**SEE ALSO**

passwd(I), passwd(V), login(I)

**NAME**

**ctime** — convert date and time to ASCII

**SYNOPSIS**

```
char *ctime(tvec)
int tvec[2];

[from Fortran]
double precision ctime
... = ctime(dummy)

int *localtime(tvec)
int tvec[2];

int *gmtime(tvec)
int tvec[2];
```

**DESCRIPTION**

*Ctime* converts a time in the vector *tvec* such as returned by *time* (II) into ASCII and returns a pointer to a character string in the form

Sun Sep 16 01:03:52 1973\n\0

All the fields have constant width.

Once the time has been placed into *t* and *t+2*, this routine is callable from assembly language as follows:

```
mov $t,-(sp)
jsr pc,_ctime
tst (sp)+
```

and a pointer to the string is available in *r0*.

The *localtime* and *gmtime* entries return pointers to integer vectors containing the broken-down time. *Localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. The value is a pointer to an array whose components are

|   |                                       |
|---|---------------------------------------|
| 0 | seconds                               |
| 1 | minutes                               |
| 2 | hours                                 |
| 3 | day of the month (1-31)               |
| 4 | month (0-11)                          |
| 5 | year - 1900                           |
| 6 | day of the week (Sunday = 0)          |
| 7 | day of the year (0-365)               |
| 8 | Daylight Saving Time flag if non-zero |

The external variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, is  $5*60*60$ ); the external variable *daylight* is non-zero iff the standard U.S.A. Daylight Saving Time conversion should be applied between the last Sundays in April and October. The external variable *nixonflg* if non-zero supersedes *daylight* and causes daylight time all year round.

A routine named *ctime* is also available from Fortran. Actually it more resembles the *time* (II) system entry in that it returns the number of seconds since the epoch 0000 GMT Jan. 1, 1970 (as a floating-point number).

**SEE ALSO**

*time*(II)

**BUGS**

**NAME**

ecvt — output conversion

**SYNOPSIS**

jsr pc,ecvt

jsr pc,fcvt

char \*ecvt(value, ndigit, decpt, sign)  
double value;

int ndigit, \*decpt, \*sign;

char \*fcvt(value, ndigit, decpt, sign)

...

**DESCRIPTION**

*Ecvt* is called with a floating point number in fr0.

On exit, the number has been converted into a string of ascii digits in a buffer pointed to by r0. The number of digits produced is controlled by a global variable *\_ndigits*.

Moreover, the position of the decimal point is contained in r2: r2=0 means the d.p. is at the left hand end of the string of digits; r2>0 means the d.p. is within or to the right of the string.

The sign of the number is indicated by r1 (0 for +; 1 for -).

The low order digit has suffered decimal rounding (i. e. may have been carried into).

From C, the *value* is converted and a pointer to a null-terminated string of *ndigit* digits is returned. The position of the decimal point is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

*Fcvt* is identical to *ecvt*, except that the correct digit has had decimal rounding for F-style output of the number of digits specified by *ndigits*.

**SEE ALSO**

printf(III)

**BUGS**

**NAME**

exp - exponential function

**SYNOPSIS**

jsr r5,exp

double exp(x)

double x;

**DESCRIPTION**

The exponential of fr0 is returned in fr0. From C, the exponential of x is returned.

**DIAGNOSTICS**

If the result is not representable, the c-bit is set and the largest positive number is returned. From C, no diagnostic is available.

Zero is returned if the result would underflow.

**BUGS**

**NAME**

floor — floor and ceiling functions

**SYNOPSIS**

```
double floor(x)
double x;

double ceil(x)
double x;
```

**DESCRIPTION**

The floor function returns the largest integer (as a double precision number) not greater than x.

The ceil function returns the smallest integer not less than x.

**BUGS**

**NAME**

fptrap — floating point interpreter

**SYNOPSIS**

sys signal; 4; fptrap

**DESCRIPTION**

*Fptrap* is a simulator of the 11/45 FP11-B floating point unit. It works by intercepting illegal instruction faults and examining the offending operation codes for possible floating point.

**FILES**

found in /lib/libu.a; a fake version is in /lib/liba.a

**DIAGNOSTICS**

A break point trap is given when a real illegal instruction trap occurs.

**SEE ALSO**

signal(II)

**BUGS**

Rounding mode is not interpreted. It's slow.

**NAME**

gamma — log gamma function

**SYNOPSIS**

```
jsr r5,gamma
double gamma(x)
double x;
```

**DESCRIPTION**

If  $x$  is passed (in fr0) *gamma* returns  $\ln |\Gamma(x)|$  (in fr0). The sign of  $\Gamma(x)$  is returned in the external integer *signgam*. The following C program might be used to calculate  $\Gamma$ :

```
y = gamma(x);
if (y > 88.)
 error();
y = exp(y);
if(signgam)
 y = -y;
```

**DIAGNOSTICS**

The c-bit is set on negative integral arguments and the maximum value is returned. There is no error return for C programs.

**BUGS**

No error return from C.

**NAME**

getarg — get command arguments from Fortran

**SYNOPSIS**

```
call getarg (i, iarray [, isize])
... = iargc(dummy)
```

**DESCRIPTION**

The *getarg* entry fills in *iarray* (which is considered to be *integer*) with the Hollerith string representing the *i* th argument to the command in which it is called. If no *isize* argument is specified, at least one blank is placed after the argument, and the last word affected is blank padded. The user should make sure that the array is big enough.

If the *isize* argument is given, the argument will be followed by blanks to fill up *isize* words, but even if the argument is long no more than that many words will be filled in.

The blank-padded array is suitable for use as an argument to *setfil* (III).

The *iargc* entry returns the number of arguments to the command, counting the first (file-name) argument.

**SEE ALSO**

*exec* (II), *setfil* (III)

**BUGS**

**NAME**

**getc** — buffered input

**SYNOPSIS**

```
'mov $filename,r0
jsr r5,fopen; iobuf
fopen(filename, iobuf)
char *filename;
struct buf *iobuf;

jsr r5,getc; iobuf
(character in r0)

getc(iobuf)
struct buf *iobuf;

jsr r5,getw; iobuf
(word in r0)

getw(iobuf)
struct buf *iobuf;
```

**DESCRIPTION**

These routines provide a buffered input facility. *Iobuf* is the address of a 518(10) byte buffer area whose contents are maintained by these routines. Its format is:

|               |          |                             |
|---------------|----------|-----------------------------|
| <i>ioptr:</i> | .=.+2    | / file descriptor           |
|               | .=.+2    | / characters left in buffer |
|               | .=.+2    | / ptr to next character     |
|               | .=.+512. | / the buffer                |

Or in C,

```
struct buf {
 int fildes;
 int nleft;
 char *nextp;
 char buffer[512];
};
```

*Fopen* may be called initially to open the file. On return, the error bit (c-bit) is set if the open failed. If *fopen* is never called, *get* will read from the standard input file. From C, the value is negative if the open failed.

*Getc* returns the next byte from the file in r0. The error bit is set on end of file or a read error. From C, the character is returned; it is -1 on end-of-file or error.

*Getw* returns the next word in r0. *Getc* and *getw* may be used alternately; there are no odd/even problems. *Getw* is may be called from C; -1 is returned on end-of-file or error, but of course is also a legitimate value.

*Iobuf* must be provided by the user; it must be on a word boundary.

To reuse the same buffer for another file, it is sufficient to close the original file and call *fopen* again.

**SEE ALSO**

*open(II)*, *read(II)*, *putc(III)*

**DIAGNOSTICS**

c-bit set on EOF or error;  
from C, negative return indicates error or EOF.

**NAME**

`getchar` — read character

**SYNOPSIS**

`getchar()`

**DESCRIPTION**

*Getchar* provides the simplest means of reading characters from the standard input for C programs. It returns successive characters until end-of-file, when it returns "\0".

Associated with this routine is an external variable called *fin*, which is a structure containing a buffer such as described under *getc* (III).

Generally speaking, *getchar* should be used only for the simplest applications; *getc* is better when there are multiple input files.

**SEE ALSO**

*getc* (III)

**DIAGNOSTICS**

Null character returned on EOF or error.

**BUGS**

-1 should be returned on EOF; null is a legitimate character.

**NAME**

getpw — get name from UID

**SYNOPSIS**

```
getpw(uid, buf)
char *buf;
```

**DESCRIPTION**

*Getpw* searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

**FILES**

/etc/passwd

**SEE ALSO**

passwd(V)

**DIAGNOSTICS**

non-zero return on error.

**BUGS**

**NAME**

hmul — high-order product

**SYNOPSIS**

hmul(x, y)

**DESCRIPTION**

*Hmul* returns the high-order 16 bits of the product of x and y. (The binary multiplication operator generates the low-order 16 bits of a product.)

**NAME**

hypot - calculate hypotenuse

**SYNOPSIS**

jsr r5,hypot

**DESCRIPTION**

The square root of  $fr0 \times fr0 + fr1 \times fr1$  is returned in fr0. The calculation is done in such a way that overflow will not occur unless the answer is not representable in floating point.

**DIAGNOSTICS**

The c-bit is set if the result cannot be represented.

**BUGS**

**NAME**

*ierror* — catch Fortran errors

**SYNOPSIS**

```
if (ierror (errno) .ne. 0) goto label
```

**DESCRIPTION**

*ierror* provides a way of detecting errors during the running of a Fortran program. Its argument is a run-time error number such as enumerated in *fc (I)*.

When *ierror* is called, it returns a 0 value; thus the *goto* statement in the synopsis is not executed. However, the routine stores inside itself the call point and invocation level. If and when the indicated error occurs, a **return** is simulated from *ierror* with a non-zero value; thus the *goto* (or other statement) is executed. It is a ghastly error to call *ierror* from a subroutine which has already returned when the error occurs.

This routine is essentially tailored to catching end-of-file situations. Typically it is called just before the start of the loop which reads the input file, and the *goto* jumps to a graceful termination of the program.

There is a limit of 5 on the number of different error numbers which can be caught.

**SEE ALSO**

*fc (I)*

**BUGS**

There is no way to ignore errors.

**NAME**

*ldiv* — long division

**SYNOPSIS**

*ldiv(hividend, ldividend, divisor)*

*lrem(hividend, ldividend, divisor)*

**DESCRIPTION**

The concatenation of the signed 16-bit *hidividend* and the unsigned 16-bit *ldividend* is divided by *divisor*. The 16-bit signed quotient is returned by *ldiv* and the 16-bit signed remainder is returned by *lrem*. Divide check and erroneous results will occur unless the magnitude of the divisor is greater than that of the high-order dividend.

An integer division of an unsigned dividend by a signed divisor may be accomplished by

*quo = ldiv(0, dividend, divisor);*

and similarly for the remainder operation.

Often both the quotient and the remainder are wanted. Therefore *ldiv* leaves a remainder in the external cell *ldivr*.

**BUGS**

No divide check check.

**NAME**

locv — long output conversion

**SYNOPSIS**

```
char *locv(hi, lo)
int hi, lo;
```

**DESCRIPTION**

*Locv* converts a signed double-precision integer, whose parts are passed as arguments, to the equivalent ASCII character string and returns a pointer to that string.

**BUGS**

**NAME**

log — natural logarithm

**SYNOPSIS**

jsr r5,log

double log(x)

double x;

**DESCRIPTION**

The natural logarithm of fr0 is returned in fr0. From C, the natural logarithm of x is returned.

**DIAGNOSTICS**

The error bit (c-bit) is set if the input argument is less than or equal to zero and the result is a negative number very large in magnitude. From C, there is no error indication.

**NAME**

monitor — prepare execution profile

**SYNOPSIS**

```
monitor(lowpc, highpc, buffer, bufsize)
int lowpc(), highpc(), buffer[], bufsize;
```

**DESCRIPTION**

*Monitor* is an interface to the system's profile entry (II). *Lowpc* and *highpc* are the names of two functions; *buffer* is the address of a (user supplied) array of *bufsize* integers. *Monitor* arranges for the system to sample the user's program counter periodically and record the execution histogram in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor(2, &etext, buf, bufsize);
```

*Etext* is a loader-defined symbol which lies just above all the program text.

To stop execution monitoring and write the results on the file *mon.out*, use

```
monitor(0);
```

Then, when the program exits, *prof* (I) can be used to examine the results.

It is seldom necessary to call this routine directly; the *-p* option of *cc* is simpler if one is satisfied with its default profile range and resolution.

**FILES**

*mon.out*

**SEE ALSO**

*prof* (I), *profil* (II), *cc* (I)

**NAME**

nargs — argument count

**SYNOPSIS**

nargs( )

**DESCRIPTION**

*Nargs* returns the number of actual parameters supplied by the caller of the routine which calls *nargs*.

The argument count is accurate only when none of the actual parameters is *float* or *double*. Such parameters count as four arguments instead of one.

**BUGS**

As indicated.

**NAME**

nlist — get entries from name list

**SYNOPSIS**

```
jsrr5,nlist; file; list
...
file: <file name\0>; .even
list: <name1xxx>; type1; value1
 <name2xxx>; type2; value2
...
0
nlist(filename, nl)
char *filename;
struct {
 char name[8];
 int type;
 int value;
} nll[];
```

**DESCRIPTION**

*Nlist* examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of a list of 8-character names (null padded) each followed by two words. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are placed in the two words following the name. If the name is not found, the type entry is set to -1.

This subroutine is useful for examining the system name list kept in the file /unix. In this way programs can obtain system addresses that are up to date.

**SEE ALSO**

a.out (V)

**DIAGNOSTICS**

All type entries are set to -1 if the file cannot be found or if it is not a valid namelist.

**BUGS**

**NAME**

perror — system error messages

**SYNOPSIS**

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];

int errno;
```

**DESCRIPTION**

*Perror* produces a short error message describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys\_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys\_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

**SEE ALSO**

Introduction to System Calls

**BUGS**

**NAME**

**pow** — floating exponentiation

**SYNOPSIS**

```
movf x,fr0
movf y,fr1
jsr pc,pow
double pow(x,y)
double x, y;
```

**DESCRIPTION**

*Pow* returns the value of  $x^y$  (in fr0). *Pow(0, y)* is 0 for any  $y$ . *Pow( $-x$ ,  $y$ )* returns a result only if  $y$  is an integer.

**SEE ALSO**

*exp(III)*, *log(III)*

**DIAGNOSTICS**

The carry bit is set on return in case of overflow, *pow(0, 0)*, or *pow( $-x$ ,  $y$ )* for non-integral  $y$ . From C there is no diagnostic.

**BUGS**

**NAME**

printf — formatted print

**SYNOPSIS**

```
printf(format, arg1, ...);
char *format;
```

**DESCRIPTION**

*Printf* converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to *printf*.

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign “—” which specifies *left adjustment* of the converted argument in the indicated field;
- an optional digit string specifying a *field width*; if the converted argument has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width;
- an optional period “.” which serves to separate the field width from the next digit string;
- an optional digit string (*precision*) which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are

d

o

x The (integral) argument is converted to decimal, octal, or hexadecimal notation respectively.

f The argument is converted to decimal notation in the style “[—]ddd.ddd” where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed. The argument should be *float* or *double*.

e The argument is converted in the style “[—]d.ddde±dd” where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. The argument should be a *float* or *double* quantity.

c The argument character or character-pair is printed if non-null.

s The argument is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

l The argument is taken to be an unsigned integer which is converted to decimal and printed (the result will be in the range 0 to 65535).

If no recognizable character appears after the %, that character is printed; thus % may be printed by use of the string %. In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by calling *putchar*.

**SEE ALSO**

*putchar* (III)

**NAME**

**putc** — buffered output

**SYNOPSIS**

```

mov $filename,r0
jsr r5,fcreat; iobuf
fcreat(file, iobuf)
char *file;
struct buf *iobuf;
(get byte in r0)
jsr r5,putc; iobuf
putc(c, iobuf)
int c;
struct buf *iobuf;
(get word in r0)
jsr r5,putw; iobuf
putw(w, iobuf);
int w;
struct buf *iobuf;
jsr r5,flush; iobuf
fflush(iobuf)
struct buf *iobuf;

```

**DESCRIPTION**

*Fcreat* creates the given file (mode 666) and sets up the buffer *iobuf* (size 518 bytes); *putc* and *putw* write a byte or word respectively onto the file; *flush* forces the contents of the buffer to be written, but does not close the file. The format of the buffer is:

|               |          |                               |
|---------------|----------|-------------------------------|
| <i>iobuf:</i> | .=.+2    | / file descriptor             |
|               | .=.+2    | / characters unused in buffer |
|               | .=.+2    | / ptr to next free character  |
|               | .=.+512. | / buffer                      |

Or in C,

```

struct buf {
 int fildes;
 int nunused;
 char *nxtnfree;
 char buff[512];
};

```

*Fcreat* sets the error bit (c-bit) if the file creation failed (from C, returns -1); none of the other routines returns error information.

Before terminating, a program should call *flush* to force out the last of the output (*fflush* from C).

The user must supply *iobuf*, which should begin on a word boundary.

To write a new file using the same buffer, it suffices to call [*f*]*flush*, close the file, and call *fcreat* again.

**SEE ALSO**

*creat(II)*, *write(II)*, *getc(III)*

**DIAGNOSTICS**

error bit possible on *fcreat* call.

**NAME**

**putchar** — write character

**SYNOPSIS**

**putchar(ch)**

**flush( )**

**DESCRIPTION**

*Putchar* writes out its argument and returns it unchanged. Only the low-order byte is written, and only if it is non-null. Unless other arrangements have been made, *putchar* writes in unbuffered fashion on the standard output file.

Associated with this routine is an external variable *fout* which has the structure of a buffer discussed under *putc* (III). If the file descriptor part of this structure (first word) is greater than 2, output via *putchar* is buffered. To achieve buffered output one may say, for example,

```
fout = dup(1); or
fout = creat(...);
```

In such a case *flush* must be called before the program terminates in order to flush out the buffered output. *Flush* may be called at any time.

**SEE ALSO**

*putc*(III)

**BUGS**

The *fout* notion is kludgy.

**NAME**

qsort — quicker sort

**SYNOPSIS**

```
(base of data in r1)
(end+1 of data in r2)
(element width in r3)
jsr pc,qsort
qsort(base, nel, width, compar)
char *base;
int (*compar);
```

**DESCRIPTION**

*Qsort* is an implementation of the quicker-sort algorithm. The assembly-language version is designed to sort equal length elements. Registers r1 and r2 delimit the region of core containing the array of byte strings to be sorted: r1 points to the start of the first string, r2 to the first location above the last string. Register r3 contains the length of each string. r2-r1 should be a multiple of r3. On return, r0, r1, r2, r3 are destroyed.

The C version has somewhat different arguments and the user must supply a comparison routine. The first argument is to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return a negative integer if the first element is to be considered less than the second, a positive integer if the second element is smaller than the first, and 0 if the elements are equal.

**SEE ALSO**

sort (I)

**BUGS**

**NAME**

rand — random number generator

**SYNOPSIS**

```
(seed in r0)
jsr pc,strand /to initialize
jsr pc,rand /to get a random number
strand(seed)
int seed;
rand()
```

**DESCRIPTION**

*Rand* uses a multiplicative congruential random number generator to return successive pseudo-random numbers (in r0) in the range from 1 to  $2^{15}-1$ .

The generator is reinitialized by calling *strand* with 1 as argument (in r0). It can be set to a random starting point by calling *strand* with whatever you like as argument, for example the low-order word of the time.

**BUGS**

The low-order bits are not very random.

**NAME**

**reset** — execute non-local goto

**SYNOPSIS**

```
setexit()
reset()
```

**DESCRIPTION**

These routines are useful for dealing with errors discovered in a low-level subroutine of a program.

*Setexit* is typically called just at the start of the main loop of a processing program. It stores certain parameters such as the call point and the stack level.

*Reset* is typically called after diagnosing an error in some subprocedure called from the main loop. When *reset* is called, it pops the stack appropriately and generates a non-local return from the last call to *setexit*.

It is erroneous, and generally disastrous, to call *reset* unless *setexit* has been called in a routine which is an ancestor of *reset*.

**BUGS**

**NAME**

**setfil** — specify Fortran file name

**SYNOPSIS**

**call setfil ( unit, hollerith-string )**

**DESCRIPTION**

*Setfil* provides a primitive way to associate an integer I/O *unit* number with a file named by the *hollerith-string*. The end of the file name is indicated by a blank. Subsequent I/O on this unit number will refer to the file whose name is specified by the string.

*Setfil* should be called only before any I/O has been done on the *unit*, or just after doing a **rewind** or **endfile**. It is ineffective for unit numbers 5 and 6.

**SEE ALSO**

**fc (I)**

**BUGS**

The exclusion of units 5 and 6 is unwarranted.

**NAME**

sin — sine, cosine

**SYNOPSIS**

**jsr r5,sin (cos)**

**double sin(x)**

**double x;**

**double cos(x)**

**double x;**

**DESCRIPTION**

The sine (cosine) of fr0 (resp. x), measured in radians, is returned (in fr0).

The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

**BUGS**

**NAME**

sqrt — square root function

**SYNOPSIS**

jsr r5,sqrt

double sqrt(x)

double x;

**DESCRIPTION**

The square root of fr0 (resp. x) is returned (in fr0).

**DIAGNOSTICS**

The c-bit is set on negative arguments and 0 is returned. There is no error return for C programs.

**BUGS**

No error return from C.

**NAME**

ttyn — return name of current typewriter

**SYNOPSIS**

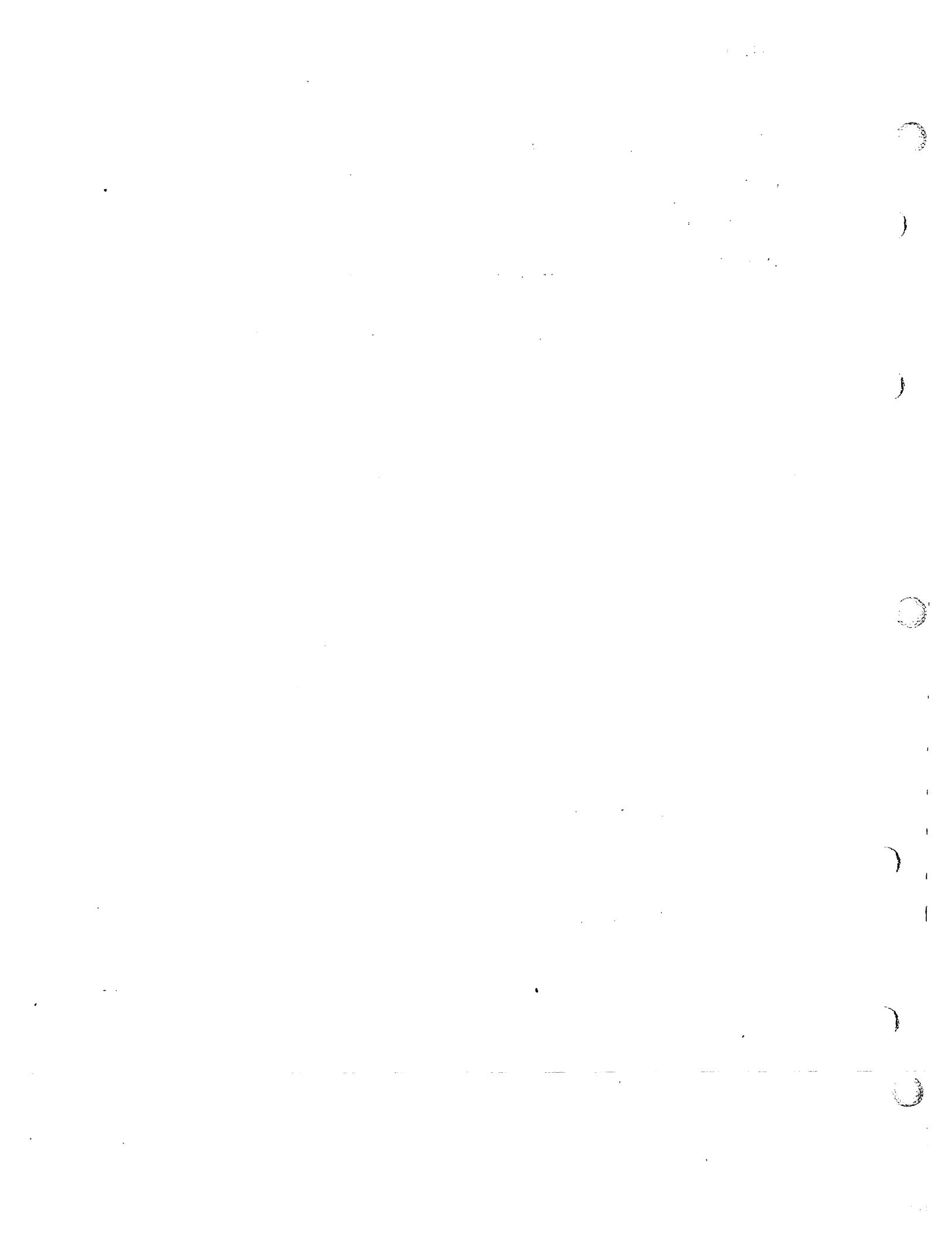
jsr pc,ttyn

ttyn(file)

**DESCRIPTION**

*Ttyn* hunts up the last character of the name of the typewriter which is the standard input (from *as*) or is specified by the argument *file* descriptor (from C). If *n* is returned, the typewriter name is then "/dev/ttyn".

x is returned if the indicated file does not correspond to a typewriter.



**NAME**

vt — display (vt01) interface

**SYNOPSIS**

```
openvt()
erase()
label(s)
char sl [];
line(x,y)
circle(x,y,r)
arc(x,y,x0,y0,x1,y1)
dot(x,y,dx,n,pattern)
int pattern[];
move(x,y)
```

**DESCRIPTION**

C interface routines to perform similarly named functions described in vt(IV). *Openvt* must be used before any of the others to open the storage scope for writing.

**FILES**

/dev/vt0, found in /lib/libp.a

**SEE ALSO**

vt (IV)

**BUGS**

?

)

)

○

)

)

○

**NAME**

cat - phototypesetter interface

**DESCRIPTION**

*Cat* provides the interface to a Graphic Systems C/A/T phototypesetter. Bytes written on the file specify font, size, and other control information as well as the characters to be flashed. The coding will not be described here.

Only one process may have this file open at a time. It is write-only.

**FILES**

/dev/cat

**SEE ALSO**

troff (I), Graphic Systems specification (available on request)

**BUGS**

**NAME**

dc - DC-11 communications interface

**DESCRIPTION**

The discussion of typewriter I/O given in tty (IV) applies to these devices.

The DC-11 typewriter interface operates at any of four speeds, independently settable for input and output. The speed is selected by the same encoding used by the DH (IV) device (enumerated in stty (II)); impossible speed changes are ignored.

**FILES**

/dev/tty[01234567abcd] 113B Dataphones (not currently connected— see dh (IV))

**SEE ALSO**

tty (IV), stty (II), dh (IV)

**BUGS**

**NAME**

dh - DH-11 communications multiplexer

**DESCRIPTION**

Each line attached to the DH-11 communications multiplexer behaves as described in tty (IV). Input and output for each line may independently be set to run at any of 16 speeds; see stty (II) for the encoding.

**FILES**

/dev/tty[f-u]

**SEE ALSO**

tty (IV), stty (II)

**BUGS**

**NAME**

dn - dn11 ACU interface

**DESCRIPTION**

The *dn?* files are write-only. The permissible codes are:

0-9 dial 0-9  
: dial \*  
; dial #  
— 4 second delay for second dial tone  
= end-of-number

The entire telephone number must be presented in a single *write* system call.

It is recommended that an end-of-number code be given even though not all ACU's actually require it.

**FILES**

|          |                           |
|----------|---------------------------|
| /dev/dn0 | connected to 801 with dp0 |
| /dev/dn1 | not currently connected   |
| /dev/dn2 | not currently connected   |

**SEE ALSO**

dp (IV)

**BUGS**

**NAME**

dp - dp11 201 data-phone interface

**DESCRIPTION**

The *dp0* file is a 201 data-phone interface. *Read* and *write* calls to *dp0* are limited to a maximum of 512 bytes. Each write call is sent as a single record. Seven bits from each byte are written along with an eighth odd parity bit. The sync must be user supplied. Each read call returns characters received from a single record. Seven bits are returned unaltered; the eighth bit is set if the byte was not received in odd parity. A 10 second time out is set and a zero-byte record is returned if nothing is received in that time.

**FILES**

/dev/dp0

**SEE ALSO**

dn (IV), gerts (III)

**BUGS**

**NAME**

kl - KL-11 or DL-11 asynchronous interface

**DESCRIPTION**

The discussion of typewriter I/O given in tty (IV) applies to these devices.

Since they run at a constant speed, attempts to change the speed via stty (II) are ignored.

The on-line console typewriter is interfaced using a KL-11 or DL-11. By appropriate switch settings during a reboot, UNIX will come up as a single-user system with I/O on the console typewriter.

**FILES**

|           |                                          |
|-----------|------------------------------------------|
| /dev/tty  |                                          |
| /dev/tty8 | synonym for /dev/tty                     |
| /dev/tty9 | second console (not currently connected) |

**SEE ALSO**

tty (IV), init (VIII)

**BUGS**

Modem control for the DL-11E is not implemented.

**NAME**

lp — line printer

**DESCRIPTION**

*Lp* provides the interface to any of the standard DEC line printers. When it is opened or closed, a suitable number of page ejects are generated. Bytes written are printed.

An internal parameter within the driver determines whether or not the device is treated as having a 96- or 64-character set. In half-ASCII mode, lower case letters are turned into upper case and certain characters are escaped according to the following table:

|   |   |
|---|---|
| { | ← |
| } | → |
| _ | — |
|   | + |
| — | — |

The driver correctly interprets carriage returns, backspaces, tabs, and form feeds. A sequence of newlines which extends over the end of a page is turned into a form feed. Lines longer than 80 characters are truncated. (This number is a parameter in the driver.)

**FILES**

/dev/lp

**SEE ALSO**

lpr (I)

**BUGS**

Half-ASCII mode and the maximum line length should be settable by a call analogous to scty (II).

**NAME**

mem — core memory

**DESCRIPTION**

*Mem* is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system using the debugger.

A memory address is an 18-bit quantity which is used directly as a UNIBUS address. References to non-existent locations cause errors to be returned.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed. In particular, the I/O area of *kmem* is located beginning at 160000 (octal) rather than at 760000. The 1K region beginning at 140000 (octal) is the system's data for the current process.

The file *null* returns end-of-file on *read* and ignores *write*.

**FILES**

/dev/mem, /dev/kmem, /dev/null

**NAME**

pc — PC-11 paper tape reader/punch

**DESCRIPTION**

*Ppt* refers to the PC-11 paper tape reader or punch, depending on whether it is read or written.

When *ppt* is opened for writing, a 100-character leader is punched. Thereafter each byte written is punched on the tape. No editing of the characters is performed. When the file is closed, a 100-character trailer is punched.

When *ppt* is opened for reading, the process waits until tape is placed in the reader and the reader is on-line. Then requests to read cause the characters read to be passed back to the program, again without any editing. This means that several null leader characters will usually appear at the beginning of the file. Likewise several nulls are likely to appear at the end. End-of-file is generated when the tape runs out.

Seek calls for this file are meaningless.

**FILES**

/dev/ppt

**BUGS**

If both the reader and the punch are open simultaneously, the trailer is sometimes not punched. Sometimes the reader goes into a dead state in which it cannot be opened.

**NAME**

rf - RF11/RS11 fixed-head disk file

**DESCRIPTION**

This file refers to the concatenation of all RS-11 disks.

Each disk contains 1024 256-word blocks. The length of the combined RF file is  $1024 \times (\text{minor} + 1)$  blocks. That is minor device zero is 1024 blocks long; minor device one is 2048, etc.

The *rf0* file accesses the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The name of the raw RF file is *rrf0*. The same minor device considerations hold for the raw interface as for the normal interface.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

**FILES**

/dev/rf0, /dev/rrf0

**BUGS**

The 512-byte restrictions are not physically necessary, but are still imposed.

**NAME**

**rk** — RK-11/RK03 (or RK05) disk

**DESCRIPTION**

*Rk?* refers to an entire RK03 disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to 4871.

Drive numbers (minor devices) of eight and greater are treated specially. Drive 8+x is the  $x+1$  way interleaving of devices *rk0* to *rкx*. Thus blocks on *rk10* are distributed alternately among *rk0*, *rk1*, and *rk2*.

The *rk* files discussed above access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RK files begin with *rrk* and end with a number which selects the same disk as the corresponding *rk* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

**FILES**

/dev/rk?, /dev/rrk?

**BUGS**

Care should be taken in using the interleaved files. First, the same drive should not be accessed simultaneously using the ordinary name and as part of an interleaved file, because the same physical blocks have in effect two different names; this fools the system's buffering strategy. Second, the combined files cannot be used for swapping or raw I/O.

**NAME**

*rp* — RP-11/RP03 moving-head disk

**DESCRIPTION**

The files *rp0* ... *rp7* refer to sections of RP disk drive 0. The files *rp8* ... *rp15* refer to drive 1 etc. This is done since the size of a full RP drive is 81200 blocks and internally the system is only capable of addressing 65536 blocks. Also since the disk is so large, this allows it to be broken up into more manageable pieces.

The origin and size of the pseudo-disks on each drive are as follows:

| disk | start      | length |
|------|------------|--------|
| 0    | 0          | 40600  |
| 1    | 40600      | 40600  |
| 2    | 0          | 9200   |
| 3    | 72000      | 9200   |
| 4    | 0          | 65535  |
| 5    | 15600      | 65535  |
| 6-7  | unassigned |        |

It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter. Here is a suggestion for two useful configurations: If the root of the file system is on some other device and the RP used as a mounted device, then *rp0* and *rp1*, which divide the disk into two equal size portions, is a good idea. Other things being equal, it is advantageous to have two equal-sized portions since one can always be copied onto the other, which is occasionally useful.

If the RP is the only disk and has to contain the root and the swap area, the root can be put on *rp2* and a mountable file system on *rp5*. Then the swap space can be put in the unused blocks 9200 to 15600 of *rp0* (or, equivalently, *rp4*). This arrangement puts the root file system, the swap area, and the i-list of the mounted file system relatively near each other and thus tends to minimize head movement.

The *rp* access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with *rrp* and end with a number which selects the same disk section as the corresponding *rp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise seek calls should specify a multiple of 512 bytes.

**FILES**

/dev/rp?, /dev/rrp?

**BUGS**

**NAME**

tc — TC-11/TU56 DECtape

**DESCRIPTION**

The files *tap0* ... *tap7* refer to the TC-11/TU56 DECtape drives 0 to 7.

The 256-word blocks on a standard DECtape are numbered 0 to 577.

**FILES**

/dev/tap?

**SEE ALSO**

tp (I)

**BUGS**

Since reading is synchronous, only one block is picked up per tape reverse.

**NAME**

tiu - Spider interface

**DESCRIPTION**

Spider is a fast digital switching network. *Tiu* is a directory which contains files each referring to a Spider control or data channel. The file */dev/tiu/dn* refers to data channel *n*, likewise */dev/tiu/cn* refers to control channel *n*.

The precise nature of the UNIX interface has not been defined yet.

**FILES**

*/dev/tiu/d?*, */dev/tiu/c?*

**BUGS**

**NAME**

*tm* — TM-11/TU-10 magtape interface

**DESCRIPTION**

The files *mt0*, ..., *mt7* refer to the DEC TU10/TM11 magtape. When opened for reading or writing, the tape is rewound. When closed, it is rewound; if it was open for writing, an end-of-file is written first.

A standard tape consists of a series of 512 byte records terminated by an end-of-file. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seek have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the "raw" interface is appropriate. The associated files are named *rmt0*, ..., *rmt7*. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seek are ignored. An error is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

**FILES**

*/dev/mt?*, */dev/rmt?*

**SEE ALSO**

*tp* (I)

**BUGS**

If any non-data error is encountered, it refuses to do anything more until closed. In raw I/O, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

**NAME**

tty — general typewriter interface

**DESCRIPTION**

All of the low-speed asynchronous communications ports use the same general interface, no matter what hardware is involved. This section discusses the common features of the interface; the KL, DC, and DH writeups (IV) describe peculiarities of the individual devices.

When a typewriter file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by *init* and become a user's input and output file. The very first typewriter file open in a process becomes the *control typewriter* for that process. The control typewriter plays a special role in handling quit or interrupt signals, as discussed below. The control typewriter is inherited by a child process during a *fork*.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters which have not yet been read by some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

When first opened, the interface mode is 300 baud; either parity accepted; 10 bits/character (one stop bit); and newline action character. The system delays transmission after sending certain function characters. Delays for horizontal tab, newline, and form feed are calculated for the Teletype Model 37; the delay for carriage return is calculated for the GE TermiNet 300. Most of these operating states can be changed by using the system call *stty(II)*. In particular, provided the hardware permits, the speed of received and transmitted characters can be changed. In addition, the following software modes can be invoked: acceptance of even parity, odd parity, or both; a raw mode in which all characters may be read one at a time; a carriage return (CR) mode in which CR is mapped into newline on input and either CR or line feed (LF) cause echoing of the sequence LF-CR; mapping of upper case letters into lower case; suppression of echoing; suppression of delays after function characters; and the printing of tabs as spaces. See *getty(VIII)* for the way that terminal speed and type are detected.

Normally, typewriter input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. The character '#' erases the last character typed, except that it will not erase beyond the beginning of a line or an EOT. The character '@' kills the entire line up to the point where it was typed, but not beyond an EOT. Both these characters operate on a keystroke basis independently of any backspacing or tabbing that may have been done. Either '@' or '#' may be entered literally by preceding it by '\'; the erase or kill character remains, but the '\' disappears.

In upper-case mode, all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. In addition, the following escape sequences are generated on output and accepted on input:

|     |          |
|-----|----------|
| for | use      |
| '   | '\'      |
| '`' | '`\'`'   |
| {   | '`\'`'{} |
| }   | '`\'`'}  |

In raw mode, the program reading is awakened on each character. No erase or kill processing is done; and the EOT, quit and interrupt characters are not treated specially. The input parity bit is passed back to the reader, but parity is still generated for output characters.

The ASCII EOT character may be used to generate an end of file from a typewriter. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file signal. The EOT is passed back unchanged in raw mode.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a *hangup* signal is sent to all processes with the typewriter as control typewriter. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any read returns with an end-of-file indication. Thus programs which read a typewriter and test for end-of-file on their input can terminate appropriately when hung up on.

Two characters have a special meaning when typed. The ASCII DEL character (sometimes called 'rubout') is not passed to a program but generates an *interrupt* signal which is sent to all processes with the associated control typewriter. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a simulated trap to an agreed-upon location. See signal (II).

The ASCII character FS generates the *quit* signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated. See signal (II). If you find it hard to type this character, try control-\ or control-shift-L.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is always generated on output. The EOT character is not transmitted (except in raw mode) to prevent terminals which respond to it from hanging up.

**SEE ALSO**

dc (IV), kl (IV), dh (IV), getty (VIII), stty (I, II), gtty (I, II), signal (II)

**BUGS**

Half-duplex terminals are not supported. On raw-mode output, parity should be transmitted as specified in the characters written.

**NAME**

vs — voice synthesizer interface

**DESCRIPTION**

Bytes written on *vs* drive a Federal Screw Works Votrax® voice synthesizer. The upper two bits encode an inflection, the other 6 specify a phoneme. The code is given in section vs (VII).

Touch-Tone® signals sent by a caller will be picked up during a *read* as the ASCII characters {0123456789#\*}.

**FILES**

/dev/vs

**SEE ALSO**

speak (VI), vs (VII)

**BUGS**

**NAME**

vt - 11/20 (vt01) interface

**DESCRIPTION**

The file *vt0* provides the interface to a PDP 11/20 which runs a VT01A-controlled Tektronix 611 storage display. The inter-computer interface is a pair of DR-11C word interfaces.

Although the display has essentially only two commands, namely "erase screen" and "display point", the 11/20 program will draw points, lines, and arcs, and print text on the screen. The 11/20 can also type information on the attached 33 TTY.

This special file operates in two basic modes. If the first byte written of the file cannot be interpreted as one of the codes discussed below, the rest of the transmitted information is assumed to ASCII and written on the screen. The screen has 33 lines (1/2 a standard page). The file simulates a 37 TTY: the control characters NL, CR, BS, and TAB are interpreted correctly. It also interprets the usual escape sequences for forward and reverse half-line motion and for full-line reverse. Greek is not available yet. Normally, when the screen is full (i.e. the 34th line is started) the screen is erased before starting a new page. To allow perusal of the displayed text, it is usual to assert bit 0 of the console switches. This causes the program to pause before erasing until this bit is lowered.

If the first byte written is recognizable, the display runs in graphic mode. In this case bytes written on the file are interpreted as display commands. Each command consists of a single byte usually followed by parameter bytes. Often the parameter bytes represent points in the plotting area. Each point coordinate consists of 2 bytes interpreted as a 2's complement 16-bit number. The plotting area itself measures  $(\pm 03777) \times (\pm 03777)$  (numbers in octal); that is, 12 bits of precision. Attempts to plot points outside the screen limits are ignored.

The graphic commands follow.

order (1); 1 parameter byte

The parameter indicates a subcommand, possibly followed by subparameter bytes, as follows:

erase (1)

The screen is erased. The program will wait until bit 0 of the console switches is down.

label (3); several subparameter bytes

The following bytes up to a null byte are printed as ASCII text on the screen. The origin of the text is the last previous point plotted; or the upper left hand of the screen if there were none.

point (2); 4 parameter bytes

The 4 parameter bytes are taken as a pair of coordinates representing a point to be plotted.

line (3); 8 parameter bytes

The parameter bytes are taken as 2 pairs of coordinates representing the ends of a line segment which is plotted. Only the portion lying within the screen is displayed.

frame (4); 1 parameter byte

The parameter byte is taken as a number of sixtieths of a second; an externally-available lead is asserted for that time. Typically the lead is connected to an automatic camera which advances its film and opens the shutter for the specified time.

circle (5); 6 parameter bytes

The parameter bytes are taken as a coordinate pair representing the origin, and a word representing the radius of a circle. That portion of the circle which lies within the screen is plotted.

arc (6); 12 parameter bytes

The first 4 parameter bytes are taken to be a coordinate-pair representing the center of a circle. The next 4 represent a coordinate-pair specifying a point on this circle.

The last 4 should represent another point on the circle. An arc is drawn counter-clockwise from the first circle point to the second. If the two points are the same, the whole circle is drawn. For the second point, only the smaller in magnitude of its two coordinates is significant; the other is used only to find the quadrant of the end of the arc. In any event only points within the screen limits are plotted.

**dot-line (7); at least 6 parameter bytes**

The first 4 parameter bytes are taken as a coordinate-pair representing the origin of a dot-line. The next byte is taken as a signed x-increment. The next byte is an unsigned word-count, with '0' meaning '256'. The indicated number of words is picked up. For each bit in each word a point is plotted which is visible if the bit is '1', invisible if not. High-order bits are plotted first. Each successive point (or non-point) is offset rightward by the given x-increment.

Asserting bit 3 of the console switches causes the display processor to throw away everything written on it. This sometimes helps if the display seems to be hung up.

**FILES**

/dev/vt0

**BUGS**

**NAME**

a.out – assembler and link editor output

**DESCRIPTION**

*A.out* is the output file of the assembler *as* and the link editor *ld*. Both programs make *a.out* executable if there were no errors and no unresolved external references.

This file has four sections: a header, the program and data text, a symbol table, and relocation bits (in that order). The last two may be empty if the program was loaded with the “-s” option of *ld* or if the symbols and relocation have been removed by *strip*.

The header always contains 8 words:

- 1 A magic number (407 or 410(8))
- 2 The size of the program text segment
- 3 The size of the initialized portion of the data segment
- 4 The size of the uninitialized (bss) portion of the data segment
- 5 The size of the symbol table
- 6 The entry location (always 0 at present)
- 7 Unused
- 8 A flag indicating relocation bits have been suppressed

The sizes of each segment are in bytes but are even. The size of the header is not included in any of the other sizes.

When a file produced by the assembler or loader is loaded into core for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number (word 0) is 407, it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. If the magic number is 410, the data segment begins at the first 0 mod 8K byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment.

The stack will occupy the highest possible locations in the core image: from 177776(8) and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by the *break* system call.

The start of the text segment in the file is 20(8); the start of the data segment is 20+S<sub>t</sub> (the size of the text) the start of the relocation information is 20+S<sub>t</sub>+S<sub>d</sub>; the start of the symbol table is 20+2(S<sub>t</sub>+S<sub>d</sub>) if the relocation information is present, 20+S<sub>t</sub>+S<sub>d</sub> if not.

The symbol table consists of 6-word entries. The first four words contain the ASCII name of the symbol, null-padded. The next word is a flag indicating the type of symbol. The following values are possible:

- 00 undefined symbol
- 01 absolute symbol
- 02 text segment symbol
- 03 data segment symbol
- 37 file name symbol (produced by *ld*)
- 04 bss segment symbol
- 40 undefined external (.globl) symbol
- 41 absolute external symbol
- 42 text segment external symbol
- 43 data segment external symbol
- 44 bss segment external symbol

Values other than those given above may occur if the user has defined some of his own instructions.

The last word of a symbol table entry contains the value of the symbol.

If the symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation bits for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the "suppress relocation" flag in the header is on.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

- 00 indicates the reference is absolute
- 02 indicates the reference is to the text segment
- 04 indicates the reference is to initialized data
- 06 indicates the reference is to bss (uninitialized data)
- 10 indicates the reference is to an undefined external symbol.

Bit 0 of the relocation word indicates if *on* that the reference is relative to the pc (e.g. "clr x"); if *off*, that the reference is to the actual symbol (e.g., "clr \*\$x").

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

**SEE ALSO**

as(I), ld(I), strip(I), nm(I)

**NAME**

archive (library) file format

**DESCRIPTION**

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld*.

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 177555(8) (it was chosen to be unlikely to occur anywhere else). The header of each file is 16 bytes long:

|       |                                     |
|-------|-------------------------------------|
| 0-7   | file name, null padded on the right |
| 8-11  | modification time of the file       |
| 12    | user ID of file owner               |
| 13    | file mode                           |
| 14-15 | file size                           |

If the file is an odd number of bytes long, it is padded with a null byte, but the size in the header is correct.

Notice there is no provision for empty areas in an archive file.

**SEE ALSO**

*ar*, *ld* (I)

**BUGS**

Names are only 8 characters, not 14. More important, there isn't enough room to store the proper mode, so *ar* always extracts in mode 666.

**NAME**

core — format of core image file

**DESCRIPTION**

UNIX writes out a core image of a terminated process when any of various errors occur. See *signal (II)* for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called "core" and is written in the process's working directory (provided it can be; normal access controls apply).

The first 1024 bytes of the core image are a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The remainder represents the actual contents of the user's core area when the core image was written. At the moment, if the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

The actual format of the information in the first 1024 bytes is complicated. A guru will have to be consulted if enlightenment is required. In general the debugger *db (I)* should be used to deal with core images.

**SEE ALSO**

*db (I)*, *signal (II)*

**NAME**

format of directories

**DESCRIPTION**

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry. Directory entries are 16 bytes long. The first word is the i-number of the file represented by the entry, if non-zero; if zero, the entry is empty.

Bytes 2-15 represent the (14-character) file name, null padded on the right. These bytes are not cleared for empty slots.

By convention, the first two entries in each directory are for “.” and “..”. The first is an entry for the directory itself. The second is for the parent directory. The meaning of “..” is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases “..” has the same meaning as “.”.

**SEE ALSO**

file system (V).

**NAME**

dump - incremental dump tape format

**DESCRIPTION**

The *dump* and *restor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of blocks of 512-bytes each. The first block has the following structure.

```
struct {
 int isize;
 int fsize;
 int date[2];
 int ddate[2];
 int tsize;
};
```

*Isize*, and *fsize* are the corresponding values from the super block of the dumped file system. (See file system (V).) *Date* is the date of the dump. *Ddate* is the incremental dump date. The incremental dump contains all files modified between *ddate* and *date*. *Tsize* is the number of blocks per reel. This block checksums to the octal value 31415.

Next there are enough whole tape blocks to contain one word per file of the dumped file system. This is  *isize* divided by 16 rounded to the next higher integer. The first word corresponds to i-node 1, the second to i-node 2, and so forth. If a word is zero, then the corresponding file was not dumped. A non-zero value for the word indicates that the file was dumped and the value is one more than the number of blocks it contains.

The rest of the tape contains for each dumped file a header block and the data blocks from the file. The header contains an exact copy of the i-node (see file system (V)) and also checksums to 031415. The number of data blocks per file is directly specified by the control word for the file and indirectly specified by the size in the i-node. If these numbers differ, the file was dumped with a 'phase error'.

**SEE ALSO**

dump (VIII), restor (VIII), file system(V)

**NAME**

fs — format of file system volume

**DESCRIPTION**

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the *super block*. Starting from its first word, the format of a super-block is

```
struct {
 int isize;
 int fsize;
 int nfree;
 int free[100];
 int ninode;
 int inode[100];
 char flock;
 char ilock;
 char fmod;
 int time[2];
};
```

*Isize* is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. *Fsize* is the first block not potentially available for allocation to a file. This number is unused by the system, but is used by programs like *check (I)* to test for bad block numbers. The free list for each volume is maintained as follows. The *free* array contains, in *free[1], ..., free[nfree-1]*, up to 99 numbers of free blocks. *Free[0]* is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain block is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 blocks is the link to the next member of the chain. To allocate a block: decrement *nfree*, and the new block is *free[nfree]*. If the new block number is 0, there are no blocks left, so give an error. If *nfree* became 0, read in the block named by the new block number, replace *nfree* by its first word, and copy the block numbers in the next 100 words into the *free* array. To free a block, check if *nfree* is 100; if so, copy *nfree* and the *free* array into it, write it out, and set *nfree* to 0. In any event set *free[nfree]* to the freed block's number and increment *nfree*.

*Ninode* is the number of free i-numbers in the *inode* array. To allocate an i-node: if *ninode* is greater than 0, decrement it and return *inode[ninode]*. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *inode* array, then try again. To free an i-node, provided *ninode* is less than 100, place its number into *inode[ninode]* and increment *ninode*. If *ninode* is already 100, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

*Flock* and *ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

*Time* is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 0000 Jan. 1 1970 (GMT). During a reboot, the *time* of the super-block for the root file system is used to set the system's idea of the time.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 32 bytes long, so 16 of them fit into a block. Therefore, i-node *i* is located in block  $(i + 31) / 16$ , and begins  $32 \cdot ((i + 31) \bmod 16)$  bytes from its start. I-node 1 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node is as follows.

```

struct {
 int flags; /* +0: see below */
 char nlinks; /* +2: number of links to file */
 char uid; /* +3: user ID of owner */
 char gid; /* +4: group ID of owner */
 char size0; /* +5: high byte of 24-bit size */
 int size1; /* +6: low word of 24-bit size */
 int addr[8]; /* +8: block numbers or device number */
 int actime[2]; /* +24: time of last access */
 int modtime[2]; /* +28: time of last modification */
};

```

The flags are as follows:

|        |                               |
|--------|-------------------------------|
| 100000 | i-node is allocated           |
| 060000 | 2-bit file type:              |
| 000000 | plain file                    |
| 040000 | directory                     |
| 020000 | character-type special file   |
| 060000 | block-type special file.      |
| 010000 | large file                    |
| 004000 | set user-ID on execution      |
| 002000 | set group-ID on execution     |
| 000400 | read (owner)                  |
| 000200 | write (owner)                 |
| 000100 | execute (owner)               |
| 000070 | read, write, execute (group)  |
| 000007 | read, write, execute (others) |

Special files are recognized by their flags and not by i-number. A block-type special file is basically one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files the high byte of the first address word specifies the type of device; the low byte specifies one of several devices of that type. The device type numbers of block and character special files overlap.

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large).

Byte number  $n$  of a file is accessed as follows.  $N$  is divided by 512 to find its logical block number (say  $b$ ) in the file. If the file is small (flag 010000 is 0), then  $b$  must be less than 8, and the physical block number is  $addr[b]$ .

If the file is large,  $b$  is divided by 256 to yield  $i$ , and  $addr[i]$  is the physical block number of the indirect block. The remainder from the division yields the word in the indirect block which contains the number of the block for the sought-for byte.

For block  $b$  in a file to exist, it is not necessary that all blocks less than  $b$  exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

#### SEE ALSO

check (VIII)

**NAME**

**mtab** — mounted file system table

**DESCRIPTION**

*Mtab* resides in directory */etc* and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last "/" is thrown away.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

**FILES**

*/etc/mtab*

**SEE ALSO**

*mount* (VIII), *umount* (VIII)

**BUGS**

**NAME**

passwd — password file

**DESCRIPTION**

*Passwd* contains for each user the following information:

- name (login name, contains no upper case)
- encrypted password
- numerical user ID
- numerical group ID (for now, always 1)
- GCOS job number and box number
- initial working directory
- program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The job and box numbers are separated by a comma. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

**SEE ALSO**

login (I), crypt (III), passwd (I)

**NAME**

speak.m — voice synthesizer vocabulary

**SYNOPSIS**

```
struct {
 int n;
 struct {
 int key;
 int phon;
 } entry[vocab.nl];
 int m;
 char strings[vocab.ml];
} vocab;
```

**DESCRIPTION**

This type of file is created and used only by *speak*. The keys are the words, word fragments and letters of the vocabulary.

The *i*th key is stored as a null-terminated string at `&vocab.strings[entry[i].key]`. Its phonetic string is similarly stored at `&vocab.strings[entry[i].phon]`. The keys are ordered lexicographically. The contents of certain parts of *vocab* are invariant: `vocab.entry[0].key = vocab.entry[0].phon = vocab.strings[0] = 0`.

Each key is maintained exactly as entered into *speak*, so that fragments are recognized by an initial '%' and letters by an initial '\*'.

Each phoneme of a phonetic string is maintained in *vs* code. A '%' in a phonetic string is represented as octal 001, and all following characters are kept literally.

**SEE ALSO**

*speak* (VI), *vs* (VII)

**BUGS**

The coding 001 for '%' precludes the use of phoneme 3-1, which is no particular loss on the model 5 Votrax, but will pinch on the model 6. 0200 would be a safe choice.

**NAME**

tp - DEC/mag tape formats

**DESCRIPTION**

The command *tp* dumps and extracts files to and DECTape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See boot procedures (VIII).

Blocks 1 through 24 for DECTape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

|               |          |
|---------------|----------|
| path name     | 32 bytes |
| mode          | 2 bytes  |
| uid           | 1 byte   |
| gid           | 1 byte   |
| unused        | 1 byte   |
| size          | 3 bytes  |
| time modified | 4 bytes  |
| tape address  | 2 bytes  |
| unused        | 16 bytes |
| check sum     | 2 bytes  |

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (file system (V)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies  $(\text{size}+511)/512$  blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks 25 (resp. 63) on are available for file storage.

A fake entry (see tp(I)) has a size of zero.

**SEE ALSO**

file system(V), tp(I)

**NAME**

ttys — typewriter initialization data

**DESCRIPTION**

The *ttys* file is read by the *init* program and specifies which typewriter special files are to have a process created for them which will allow people to log in. It consists of lines of 3 characters each.

The first character is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is the last character in the name of a typewriter; e.g. x refers to the file '/dev/ttyx'. The third character is the offset in a table contained in *init* which selects an initialization program for the line; currently it must be '0' and the only such program is */etc/getty*.

**FILES**

found in /etc

**SEE ALSO**

*init* (VIII)

**NAME**

utmp — user information

**DESCRIPTION**

This file allows one to discover information about who is currently using UNIX. The file is binary; each entry is 16(10) bytes long. The first eight bytes contain a user's login name or are null if the table slot is unused. The low order byte of the next word contains the last character of a typewriter name. The next two words contain the user's login time. The last word is unused.

This file resides in directory /tmp.

**SEE ALSO**

init (VIII) and login (I), which maintain the file; who (I), which interprets it.

**NAME**

wtmp — user login history

**DESCRIPTION**

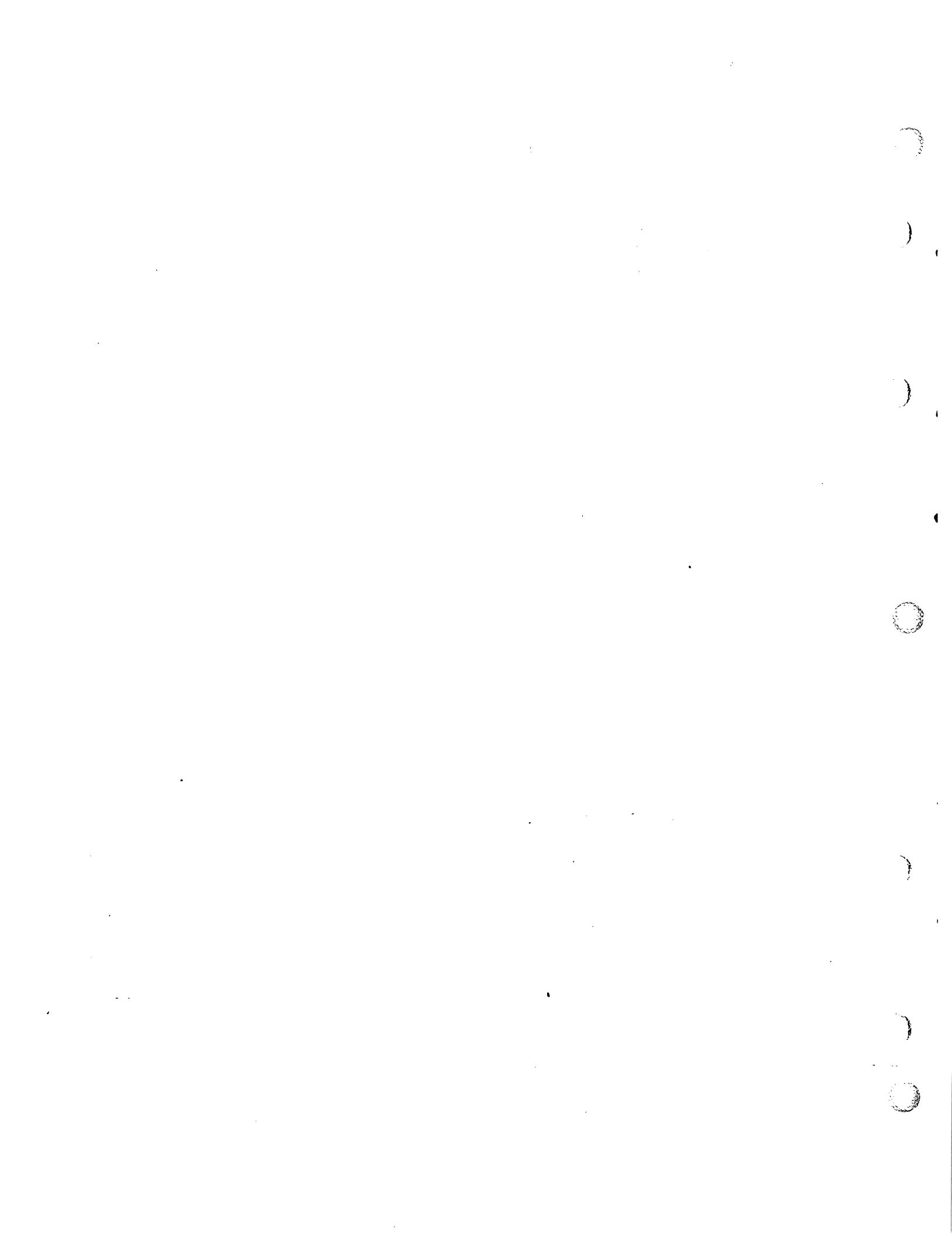
This file records all logins and logouts. Its format is exactly like utmp (V) except that a null user name indicates a logout on the associated typewriter. Furthermore, the typewriter name “” indicates that the system was rebooted at the indicated time; the adjacent pair of entries with typewriter names ‘‘ and ‘’ indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

*Wtmp* is maintained by login (I) and init (VII). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by ac (VIII).

This file resides in directory */usr/adm*.

**SEE ALSO**

login (I), init (VIII), ac (VIII), who (I)



**NAME**

apl - APL interpreter

**SYNOPSIS**

apl

**DESCRIPTION**

*Apl* is an interpreter for the language APL described in the reference. The interpreter maintains its workspace on disk rather than in core. This has two consequences: there is the potential of a million byte workspace; it takes a week to access that much data.

**Not Implemented (never)**

1. Lamination (except for scalar,scalar)
2. 0 div 0 is a domain error.
3. 0 mod x is a domain error.
4. No function definition — use 'edit fname' to enter the system editor; type "w" when done editing to write the function out in a place where apl can pick it up. Type "w file" to save it.
5. Indexing is off in character vectors containing overstrikes.

**Under Implemented (later)**

1. Negative numbers raised to fractional powers are handled incorrectly.
2. No trace or SI.
3. Incomplete set of I-beams and system calls.

**Over Implemented (over zealous)**

1. Ravel[i] — obvious extension of cat.
2. Grade up and grade down extend to matrices.
3. Arbitrary overstriking is allowed in characters.

**FILES**

/usr/lib/apl/\* programs  
alloc.d workspace  
apl\_ed editor intermediate

**SEE ALSO**

IBM GH20-0906-1 "APL User's Manual"  
/usr/pub/apl ASCII APL character set

**BUGS**

**NAME**

*azel* — obtain satellite predictions

**SYNOPSIS**

*azel* [ **-d** ] [ **-l** ] *satellite1* [ **-d** ] [ **-l** ] *satellite2* ...

**DESCRIPTION**

*Azel* predicts, in convenient form, the apparent trajectories of Earth satellites whose orbital elements are given in the argument files. If a given satellite name cannot be read, an attempt is made to find it in a directory of satellites maintained by the program's author. The **-d** option causes *azel* to ask for a date and read line 1 data (see below) from the standard input. The **-l** option causes *azel* to ask for the observer's latitude, west-longitude, and height above sea level.

For each satellite given the program types its full name, the date, and a sequence of lines each containing a time, an azimuth, an elevation, a distance, and a visual magnitude. Each such line indicates that: at the indicated time, the satellite may be seen from Murray Hill (or provided location) at the indicated azimuth and elevation, and that its distance and apparent magnitude are as given. Predictions are printed only when the sky is dark (sun more than 5 degrees below the horizon) and when the satellite is not eclipsed by the earth's shadow. Satellites which have not been seen and verified will not have had their visual magnitude level set correctly.

All times input and output by *azel* are GMT (Universal Time).

The satellites for which elements are maintained are:

sla,b,e,f,k Skylab A through Skylab K. Skylabs A and B are the laboratory and its rocket respectively; the remainder are various other objects attendant upon its launch and subsequent activities. A, B, and probably K have been sighted and verified.  
cop Copernicus I. Never verified.  
oao Orbiting Astronomical Observatory. Seen and verified.  
pag Pageos I. Seen and verified; fairly dim (typically 2nd-3rd magnitude), but elements are extremely accurate.  
exp19 Explorer 19; seen and verified, but quite dim (4th-5th magnitude) and fast-moving.  
c103b, c156b, c184b, c206b, c220b, c461b, c500b Various of the USSR Cosmos series; none seen.  
7276a Unnamed (satellite # 72-76A); not seen.

The element files used by *azel* contain five lines. The first line gives a year, month number, day, hour, and minute at which the program begins its consideration of the satellite, followed by a number of minutes and an interval in minutes. If the year, month, and day are 0, they are taken to be the current date (taken to change at 6 A.M. local time). The output report starts at the indicated epoch and prints the position of the satellite for the indicated number of minutes at times separated by the indicated interval. This line is ended by two numbers which specify options to the program governing the completeness of the report; they are ordinarily both "1". The first option flag suppresses output when the sky is not dark; the second suppresses output when the satellite is eclipsed by the earth's shadow. The next line of an element file is the full name of the satellite. The next three are the elements themselves (including certain derivatives of the elements).

**FILES**

/usr/jfo/el/\* — orbital element files

**SEE ALSO**

sky (VI)

**AUTHOR**

J. F. Ossanna

**NAME****bas - basic****SYNOPSIS****bas [ file ]****DESCRIPTION**

*Bas* is a dialect of Basic. If a file argument is provided, the file is used for input before the console is read. *Bas* accepts lines of the form:

statement  
integer statement

Integer numbered statements (known as internal statements) are stored for later execution. They are stored in sorted ascending order. Non-numbered statements are immediately executed. The result of an immediate expression statement (that does not have '=' as its highest operator) is printed.

Statements have the following syntax:

**expression**

The expression is executed for its side effects (assignment or function call) or for printing as described above.

**done**

Return to system level.

**draw expression expression expression**

A line is drawn on the Tektronix 611 display '/dev/vt0' from the current display position to the XY co-ordinates specified by the first two expressions. The scale is zero to one in both X and Y directions. If the third expression is zero, the line is invisible. The current display position is set to the end point.

**display list**

The list of expressions and strings is concatenated and displayed (i.e. printed) on the 611 starting at the current display position. The current display position is not changed.

**dump**

The name and current value of every variable is printed.

**erase**

The 611 screen is erased.

**for name = expression expression statement**

**for name = expression expression**

...

**next**

The *for* statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression.

**goto expression**

The expression is evaluated, truncated to an integer and execution goes to the corresponding integer numbered statement. If executed from immediate mode, the internal statements are compiled first.

**if expression statement**

The statement is executed if the expression evaluates to non-zero.

**list [expression [expression]]**

is used to print out the stored internal statements. If no arguments are given, all internal statements are printed. If one argument is given, only that internal statement is listed. If two arguments are given, all internal statements inclusively between the arguments are printed.

**print list**

The list of expressions and strings are concatenated and printed. (A string is delimited by " characters.)

**prompt list**

*Prompt* is the same as *print* except that no newline character is printed.

**return [expression]**

The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

**run**

The internal statements are compiled. The symbol table is re-initialized. The random number generator is reset. Control is passed to the lowest numbered internal statement.

**save [expression [expression]]**

*Save* is like *list* except that the output is written on the *file* argument. If no argument is given on the command, **b.out** is used.

Expressions have the following syntax:

**name**

A name is used to specify a variable. Names are composed of a letter followed by letters and digits. The first four characters of a name are significant.

**number**

A number is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an e followed by a possibly signed exponent.

**( expression )**

Parentheses are used to alter normal order of evaluation.

**- expression**

The result is the negation of the expression.

**expression operator expression**

Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. A complete list of operators is given below.

**expression ( [expression [ , expression] ... ] )**

Functions of an arbitrary number of arguments can be called by an expression followed by the arguments in parentheses separated by commas. The expression evaluates to the line number of the entry of the function in the internally stored statements. This causes the internal statements to be compiled. If the expression evaluates negative, a builtin function is called. The list of builtin functions appears below.

**name [ expression [ , expression] ... ]**

Each expression is truncated to an integer and used as a specifier for the name. The result is syntactically identical to a name. **a[1,2]** is the same as **a[1][2]**. The truncated expressions are restricted to values between 0 and 32767.

The following is the list of operators:

**=**

= is the assignment operator. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left, all other operators bind left to right.

**& |**

& (logical and) has result zero if either of its arguments are zero. It has result one if both its arguments are non-zero. | (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments are non-zero.

< <= > >= == <>

The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, == equal to, <> not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: a>b>c is the same as a>b&b>c.

+ -

Add and subtract.

\* /

Multiply and divide.

Exponentiation.

The following is a list of builtin functions:

**arg(i)**

is the value of the *i*-th actual parameter on the current level of function call.

**exp(x)**

is the exponential function of *x*.

**log(x)**

is the natural logarithm of *x*.

**sin(x)**

is the sine of *x* (radians).

**cos(x)**

is the cosine of *x* (radians).

**atn(x)**

is the arctangent of *x*. Its value is between  $-\pi/2$  and  $\pi/2$ .

**rnd( )**

is a uniformly distributed random number between zero and one.

**expr( )**

is the only form of program input. A line is read from the input and evaluated as an expression. The resultant value is returned.

**int(x)**

returns *x* truncated to an integer.

#### FILES

|           |           |
|-----------|-----------|
| /tmp/btm? | temporary |
| b.out     | save file |

#### DIAGNOSTICS

Syntax errors cause the incorrect line to be typed with an underscore where the parse failed. All other diagnostics are self explanatory.

#### BUGS

Has been known to give core images.

**NAME**

bj — the game of black jack

**SYNOPSIS**

/usr/games/bj

**DESCRIPTION**

*Bj* is a serious attempt at simulating the dealer in the game of black jack (or twenty-one) as might be found in Reno. The following rules apply:

The bet is \$2 every hand.

A player 'natural' (black jack) pays \$3. A dealer natural loses \$2. Both dealer and player naturals is a 'push' (no money exchange).

If the dealer has an ace up, the player is allowed to make an 'insurance' bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins \$2 if the dealer has a natural and loses \$1 if the dealer does not.

If the player is dealt two cards of the same value, he is allowed to 'double'. He is allowed to play two hands, each with one of these cards. (The bet is doubled also; \$2 on each hand.)

If a dealt hand has a total of ten or eleven, the player may 'double down'. He may double the bet (\$2 to \$4) and receive exactly one more card on that hand.

Under normal play, the player may 'hit' (draw a card) as long as his total is not over twenty-one. If the player 'busts' (goes over twenty-one), the dealer wins the bet.

When the player 'stands' (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.

If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by y followed by a new line for 'yes', or just new line for 'no'.

? (means, "do you want a hit?")

Insurance?

Double down?

Every time the deck is shuffled, the dealer so states and the 'action' (total bet) and 'standing' (total won or lost) is printed. To exit, hit the interrupt key (DEL) and the action and standing will be printed.

**BUGS**

**NAME**

cal — print calendar

**SYNOPSIS**

cal [ month ] year

**DESCRIPTION**

*Cal* prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

**BUGS**

The year is always considered to start in January even though this is historically naive.

**NAME**

**catsim** — phototypesetter simulator

**SYNOPSIS**

**catsim**

**DESCRIPTION**

*Catsim* will interpret its standard input as codes for the phototypesetter (*cat*). The output of *cat-sim* is output to the display (*vt*).

About the only use of *catsim* is to save time and paper on the phototypesetter by the following command:

```
troff -t file ... | catsim
```

**FILES**

/dev/vt0

**SEE ALSO**

troff (I), cat (IV), vt (IV)

**BUGS**

Point sizes are not correct. The vt character set is restricted to one font of ASCII.

**NAME**

chess — the game of chess

**SYNOPSIS**

/usr/games/chess

**DESCRIPTION**

*Chess* is a computer program that plays class D chess. Moves may be given either in standard (descriptive) notation or in algebraic notation. The symbol '+' is used to specify check and is not required; 'o-o' and 'o-o-o' specify castling. To play black, type 'first'; to print the board, type an empty line.

Each move is echoed in the appropriate notation followed by the program's reply.

**FILES**

/usr/lib/book                    opening 'book'

**DIAGNOSTICS**

The most cryptic diagnostic is 'eh?' which means that the input was syntactically incorrect.

**WARNING**

Over-use of this program will cause it to go away.

**BUGS**

Pawns may be promoted only to queens.

**NAME**  
col — filter reverse line feeds

**SYNOPSIS**  
col

**DESCRIPTION**  
*Col* reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ascii code ESC-7). *Col* is particularly useful for filtering multicolumn output made with the 'rt' command of *nroff*.

**SEE ALSO**  
*nroff* (I)

**BUGS**  
Can't back up more than 102 lines.

**NAME**

cubic — three dimensional tic-tac-toe

**SYNOPSIS**

/usr/games/cubic

**DESCRIPTION**

*Cubic* plays the game of three dimensional  $4 \times 4 \times 4$  tic-tac-toe. Moves are given by the three digits (each 1-4) specifying the coordinate of the square to be played.

**WARNING**

Too much playing of the game will cause it to disappear.

**BUGS**

**NAME**

factor — discover prime factors of a number

**SYNOPSIS**

factor

**DESCRIPTION**

When *factor* is invoked, it types out 'Enter:' at you. If you type in a positive number less than  $2^{56}$  (about  $7.2 \times 10^{16}$ ) it will repeat the number back at you and then its prime factors each one printed the proper number of times. Then it says 'Enter:' again. To exit, feed it an EOT or a delete.

Maximum time to factor is proportional to  $\sqrt{n}$  and occurs when *n* is prime or the square of a prime. It takes 1 minute to factor a prime near  $10^{13}$ .

**DIAGNOSTICS**

'Ouch.' for input out of range or for garbage input.

**BUGS**

**NAME**

**graf** — draw graph on GSI terminal

**SYNOPSIS**

**graf | nroff | gsi**

**DESCRIPTION**

*Graf* is a preprocessor to *nroff* (q.v.) for producing plots imbedded in documents. The standard input is copied to the standard output except for plots, which are inserted whenever a line beginning ".GR" is found. The remainder of the line should be the arguments to *plog*, normally including a "<file>" to point off to the data for the graph. *Graf* itself reads its standard input.

*Graf* and *neqn* can be used together:

**neqn files | graf | nroff | gsi**

produces a memo with figures and equations and text all intermixed. There is no typesetter equivalent of *graf*, nor are there any plans for one.

**SEE ALSO**

*plog* (VI), *neqn* (I), *nroff* (I), *gsi* (VI)

**BUGS**

Same as *plog* (VI). Axes and labels are not scaled down correctly for smaller graphs. It should recognize .so commands.

**NAME**

*gsi* — interpret funny characters on GSI terminal

**SYNOPSIS**

*gsi*

**DESCRIPTION**

*Gsi* interprets commands specific to the GSI terminal. It converts half line forward and reverse motions into the right vertical motion. It also attempts to draw Greek letters and other special symbols of the Model 37 extended character set. (These are normally preceded by shift-out and followed by shift-in.) *Gsi* is most often used to print equations neatly, in the sequence

*neqn file ... | nroff | gsi*

*Gsi* also interprets the plot control characters ACK and BEL. This makes it useful in the sequence

*graf | nroff | gsi*

**FILES****SEE ALSO**

*neqn (I), graf (VI), greek (VII)*

**BUGS**

Some funny characters can't be correctly printed in column 1 because you can't move to the left from there.

**NAME**

hyphen — find hyphenated words

**SYNOPSIS**

**hyphen** file ...

**DESCRIPTION**

It finds all of the words in a document which are hyphenated across lines and prints them back at you in a convenient format.

If no arguments are given, the standard input is used. Thus *hyphen* may be used as a filter.

**BUGS**

Yes, it gets confused, but with no ill effects other than spurious extra output.

**NAME**

**ibm** — submit off-line job to HO IBM 370

**SYNOPSIS**

**ibm [ -j ] file ...**

**DESCRIPTION**

*ibm* arranges to have the 201 data phone daemon submit a job to the IBM 370 at Holmdel via the Murray Hill H6070. Normally the job is submitted with enough "JCL" (the IBM version of the shell) to return the output to your box at Murray Hill. You can supply your own if you dare— the **-j** option suppresses all JCL.

If there are no arguments, the standard input is read and submitted. Thus *ibm* may be used as a filter.

**FILES**

|             |                      |
|-------------|----------------------|
| /usr/dpd/*  | spool area           |
| /etc/passwd | personal ident cards |
| /etc/dpd    | daemon               |

**SEE ALSO**

*dpd* (VIII), *passwd* (V)

**BUGS**

Stuff is sent on 6-bit cards, so lower case vanishes, as do some of the special characters.

**NAME**

lc - LIL compiler

**SYNOPSIS**

lc [ -c ] [ -p ] [ -P ] file ...

**DESCRIPTION**

*Lc* is the UNIX LIL compiler. It accepts three types of arguments:

Arguments whose names end with '.l' are taken to be LIL source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.l'. The '.o' file is normally deleted, however, if a single LIL program is compiled and loaded all at one go.

The following flags are interpreted by *lc*.

- c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.
- p If loading takes place, replace the standard startup routine by one which automatically calls the *monitor* subroutine (III) at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof(I)*.
- P Run only the string preprocessor on the named LIL programs, and leave the output on corresponding files suffixed '.i'.

Other arguments are taken to be object programs or perhaps libraries of routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out**.

**FILES**

|              |                                  |
|--------------|----------------------------------|
| file.l       | input file                       |
| file.o       | object file                      |
| a.out        | loaded output                    |
| /tmp/ctm?    | temporary                        |
| /lib/l[01]   | compiler                         |
| /lib/crt0.o  | runtime startoff                 |
| /lib/mcrt0.o | runtime startoff for monitoring. |
| /lib/libc.a  | builtin functions, etc.          |

**SEE ALSO**

"Programming in LIL: a tutorial," LIL Reference Manual,  
*monitor* (III), *prof* (I), *cdb* (I), *ld* (I).

**DIAGNOSTICS**

The diagnostics produced by LIL itself are intended to be self-explanatory.

**BUGS**

Creates temporary symbols of the form '..octal', which might conflict with user defined symbols.

**NAME**

m6 — general purpose macroprocessor

**SYNOPSIS**

**m6 [ name ]**

**DESCRIPTION**

*M6* copies the standard input to the standard output, with substitutions for any macro calls that appear. When a file name argument is given, that file is read before the standard input.

The processor is as described in the reference with these exceptions:

**#def,arg1,arg2,arg3:** causes *arg1* to become a macro with defining text *arg2* and (optional) built-in serial number *arg3*.

**#del,arg1:** deletes the definition of macro *arg1*.

**#end:** is not implemented.

**#list,arg1:** sends the name of the macro designated by *arg1* to the current destination without recognition of any warning characters; *arg1* is 1 for the most recently defined macro, 2 for the next most recent, and so on. The name is taken to be empty when *arg1* doesn't make sense.

**#warn,arg1,arg2:** replaces the old warning character *arg1* by the new warning character *arg2*.

**#quote,arg1:** sends the definition text of macro *arg1* to the current destination without recognition of any warning characters.

**#serial,arg1:** delivers the built-in serial number associated with macro *arg1*.

**#source,arg1:** is not implemented.

**#trace,arg1:** with *arg1* = '1' causes a reconstruction of each later call to be placed on the standard output with a call level number; other values of *arg1* turn tracing off.

The built-in 'warn' may be used to replace inconvenient warning characters. The example below replaces '#':<'> by '[' ']' '{ '}'.

```
#warn,<#>,:
[warn,<:>,:
[warn,[substr,<<>>,1,1;,{
[warn,[substr,{>>,2,1;,{
[now,{calls look like this}]]
```

Every built-in function has a serial number, which specifies the action to be performed before the defining text is expanded. The serial numbers are: 1 gt, 2 eq, 3 ge, 4 lt, 5 ne, 6 le, 7 seq, 8 sne, 9 add, 10 sub, 11 mpy, 12 div, 13 exp, 20 if, 21 def, 22 copy, 23 warn, 24 size, 25 substr, 26 go, 27 gobk, 28 del, 29 dnl, 32 quote, 33 serial, 34 list, 35 trace. Serial number 0 specifies no built-in action.

**SEE ALSO**

A. D. Hall, M6 Reference Manual. Computer Science Technical Report #2, Bell Laboratories, 1969.

**DIAGNOSTICS**

Various table overflows and "impossible" conditions result in comment and dump. There are no diagnostics for poorly formed input.

**AUTHOR**

M. D. McIlroy

**BUGS**

Provision should be made to extend tables as needed, instead of wasting a big fixed core allocation. You get what the PDP11 gives you for arithmetic.

**NAME**

maze — generate a maze problem

**SYNOPSIS**

maze

**DESCRIPTION**

*Maze* asks a few questions and then prints a maze.

**BUGS**

Some mazes (especially small ones) have no solutions.

**NAME**

moo — guessing game

**SYNOPSIS**

/usr/games/moo

**DESCRIPTION**

*Moo* is a guessing game imported from England. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits being scored on each guess. A 'cow' is a correct digit in an incorrect position. A 'bull' is a correct digit in a correct position. The game continues until the player guesses the number (a score of four bulls).

**BUGS**

**NAME**

npr — print file on Spider line-printer

**SYNOPSIS**

npr file ...

**DESCRIPTION**

*Npr* prints files on the line-printer in the Spider room, sending them over the Spider loop.

If there are no arguments, the standard input is read and submitted. Thus *npr* may be used as a filter.

**FILES**

/dev/tiu/d2      tiu to loop

**BUGS**

**NAME**

plog — make a graph on the gsi terminal

**SYNOPSIS**

**plog [ option ] ...**

**DESCRIPTION**

*Plog* is almost the same as *plot* (q.v.) but the plot is written on the standard output using the control sequences for the GSI terminal. The following changes have been made:

- The default for grids is no grid at all.
- The 'a' option can be followed by two arguments; the second is the starting point for automatic abscissas.
- There is a new option 'h' which must be followed by a numerical argument: it specifies the height desired for the plot.
- There is a new option 'w' similar to 'h', except that the width is specified. If only one of 'h' and 'w' is given, the plot is made square of the indicated size. If neither is given, the plot is made six inches square.
- There is a new option 'r' to be followed by a number which locates the plot that many inches to the right on the page.

**SEE ALSO**

*plot* (VI)

**BUGS**

Same as *plot* (VI). Drawing lines is not yet done exactly right. If you store the output in a file, before printing with cat you must turn off delays and turn off CR-NL echo (e.g. "stty -delay nl");

**NAME**

**plot** — make a graph

**SYNOPSIS**

**plot [ option ] ...**

**DESCRIPTION**

*Plot* takes pairs of numbers from the standard input as abscissas and ordinates of a graph. The graph is plotted on the storage scope, /dev/vt0.

The following options are recognized, each as a separate argument.

- a** Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.
- c** Place character string given by next argument at each point.
- d** Omit connections between points. (Disconnect.)
- gn** Grid style:
  - n=0**, no grid
  - n=1**, axes only
  - n=2**, complete grid (default).
- s** Save screen, don't erase before plotting.
- x** Next 1 (or 2) arguments are lower (and upper) x limits.
- y** Next 1 (or 2) arguments are lower (and upper) y limits.

Points are connected by straight line segments in the order they appear in input. If a specified lower limit exceeds the upper limit, or if the automatic increment is negative, the graph is plotted upside down. Automatic abscissas begin with the lower x limit, or with 0 if no limit is specified. Grid lines and automatically determined limits fall on round values, however roundness may be subverted by giving an inappropriately rounded lower limit. Plotting symbols specified by c are placed so that a small initial letter, such as + o x, will fall approximately on the plotting point.

**FILES**

/dev/vt0

**SEE ALSO**

spline (VI), plog (VI)

**BUGS**

A limit of 1000 points is enforced silently.

**NAME**

*ptx* — permuted index

**SYNOPSIS**

*ptx* [ **-t** ] *input* [ *output* ]

**DESCRIPTION**

*Ptx* generates a permuted index from file *input* on file *output*. It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally the sorted lines are rotated so the keyword comes at the middle of the page.

*Input* should be edited to remove useless lines. The following words are suppressed: 'a', 'an', 'and', 'as', 'is', 'for', 'of', 'on', 'or', 'the', 'to', 'up'.

The optional argument **-t** causes *ptx* to prepare its output for the phototypesetter.

The index for this manual was generated using *ptx*.

**FILES**

/bin/sort

**NAME**

sfs — structured file scanner

**SYNOPSIS**

sfs filename [ - ]

**DESCRIPTION**

*Sfs* provides an interactive program for scanning and patching a structured file. If the second argument is supplied, the file is block addressed.

Some features of *sfs* include.

1. It provides interactive and preprogrammed operation.
2. It provides expression evaluation (32 bit precision) and branching.
3. It provides the ability to define a large set of hierarchical structure definitions.
4. It provides the ability to locate, to dump, and to patch specific instances of structure in the file. Furthermore, in the dump and patch operations the external form of the structure is selected by the user.
5. It provides the ability to escape to the UNIX command level to allow the use of other UNIX debugging aids.

**SEE ALSO**

“SFS reference manual” (internal memorandum)

**BUGS**

**NAME**

sky — obtain ephemerides

**SYNOPSIS**

sky

**DESCRIPTION**

*Sky* predicts the apparent locations of the Sun, the Moon, the planets out to Saturn, stars of magnitude at least 2.5, and certain other celestial objects. *Sky* reads the standard input to obtain a GMT time typed on one line with blanks separating year, month number, day, hour, and minute; if the year is missing the current year is used. If a blank line is typed the current time is used. The program prints the azimuth, elevation, and magnitude of objects which are above the horizon at the ephemeris location of Murray Hill at the indicated time.

Placing a "1" input after the minute entry causes the program to print out the Greenwich Sidereal Time at the indicated moment and to print for each body its right ascension and declination as well as its azimuth and elevation. Also, instead of the magnitude, the geocentric distance of the body, in units the program considers convenient, is printed. (For planets the unit is essentially A. U.)

The magnitudes of Solar System bodies are not calculated and are given as 0. The effects of atmospheric extinction are not included; the mean magnitudes of variable stars are marked with "##".

For all bodies, the program takes into account precession and nutation of the equinox, annual (but not diurnal) aberration, diurnal parallax, and the proper motion of stars (but not annual parallax). In no case is refraction included.

The program takes into account perturbations of the Earth due to the Moon, Venus, Mars, and Jupiter. The expected accuracies are: for the Sun and other stellar bodies a few tenths of seconds of arc; for the Moon (on which particular care is lavished) likewise a few tenths of seconds. For the Sun, Moon and stars the accuracy is sufficient to predict the circumstances of eclipses and occultations to within a few seconds of time. The planets may be off by several minutes of arc.

**FILES**

/usr/lib/startab, /usr/lib/moontab

**SEE ALSO**

azel (VI)

*American Ephemeris and Nautical Almanac*, for the appropriate years; also, the *Explanatory Supplement to the American Ephemeris and Nautical Almanac*.

**AUTHOR**

R. Morris

**BUGS**

**NAME**

sno — Snobol interpreter

**SYNOPSIS**

sno [ file ]

**DESCRIPTION**

*Sno* is a Snobol III (with slight differences) compiler and interpreter. *Sno* obtains input from the concatenation of *file* and 'the standard input. All input through a statement containing the label 'end' is considered program and is compiled. The rest is available to 'syspit'.

*Sno* differs from Snobol III in the following ways.

There are no unanchored searches. To get the same effect:

a \*\* b      unanchored search for b  
a \*x\* b = x c      unanchored assignment

There is no back referencing.

x = "abc"  
a \*x\* x      is an unanchored search for 'abc'

Function declaration is different. The function declaration is done at compile time by the use of the label 'define'. Thus there is no ability to define functions at run time and the use of the name 'define' is preempted. There is also no provision for automatic variables other than the parameters. For example:

define f()

or

define f(a,b,c)

All labels except 'define' (even 'end') must have a non-empty statement.

If 'start' is a label in the program, program execution will start there. If not, execution begins with the first executable statement. 'define' is not an executable statement.

There are no builtin functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators '/' and '\*' must be set off by space.

The right side of assignments must be non-empty.

Either ' or " may be used for literal quotes.

The pseudo-variable 'sysppt' is not available.

**SEE ALSO**

Snobol III manual. (JACM; Vol. 11 No. 1; Jan 1964; pp 21)

**BUGS**

**NAME**

**speak** — word to voice translator

**SYNOPSIS**

**speak [ -epsv ] [ vocabulary [ output ] ]**

**DESCRIPTION**

*Speak* turns a stream of words into utterances and outputs them to a voice synthesizer, or to a specified output file. It has facilities for maintaining a vocabulary. It receives, from the standard input

- working lines: text of words separated by blanks
- phonetic lines: strings of phonemes for one word preceded and separated by commas. The phonemes may be followed by comma-percent then a 'replacement part' — an ASCII string with no spaces. The phonetic code is given in *bs* (VII).
- empty lines
- command lines: beginning with !. The following command lines are recognized:

|                |                                                    |
|----------------|----------------------------------------------------|
| <b>!r file</b> | replace coded vocabulary from file                 |
| <b>!w file</b> | write coded vocabulary on file                     |
| <b>!p</b>      | print phonetics for working word                   |
| <b>!l</b>      | list vocabulary on standard output with phonetics  |
| <b>!c word</b> | copy phonetics from working word to specified word |
| <b>!d</b>      | print decomposition into substrings                |

Each working line replaces its predecessor. Its first word is the 'working word'. Each phonetic line replaces the phonetics stored for the working word. In particular, a phonetic line of comma only deletes the entry for the working word. Each working line, phonetic line or empty line causes the working line to be uttered. The process terminates at the end of input.

Unknown words are pronounced by rules, and failing that, are spelled. Spelling is done by taking each character of the word, prefixing it with '\*', and looking it up. Unspellable words burp.

*Speak* is initialized with a coded vocabulary stored in file /usr/lib/speak.m. The vocabulary option substitutes a different file for /usr/lib/speak.m.

A set of single letter options may appear in any order preceded by -. Their meanings are:

- e** suppress English preprocessing
- p** suppress pronunciation by rule
- s** suppress spelling
- v** suppress voice output

The following input will reconstitute a coded vocabulary, 'speak.m', from an ascii listing, 'speak.v', that was created using !l. 'Null' names a nonexistent vocabulary file.

```
cat speak.v - | speak -v null
!w speak.m
```

**FILES**

/usr/lib/speak.m

**SEE ALSO**

M. D. McIlroy, "Synthetic English Speech by Rule," Computing Science Technical Report #14, Bell Laboratories, 1973  
*vs* (VII), *vs* (IV)

**BUGS**

Excessively long words cause dumps.

Space is not reclaimed from deleted entries; use !w and !r to effect reclamation.  
The first phoneme is sometimes dropped when !p is used after !d.

**NAME**

spline — interpolate smooth curve

**SYNOPSIS**

spline [ option ] ...

**DESCRIPTION**

*Spline* takes pairs of numbers from the standard input as abscissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output (R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed., 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by *plot* (I).

The following options are recognized, each as a separate argument.

- a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.
- k The constant  $k$  used in the boundary value computation

$$y_0'' = ky_1'', \quad y_n'' = ky_{n-1}''$$

is set by the next argument. By default  $k = 0$ .

- n Space output points so that approximately  $n$  points occur between the lower and upper  $x$  limits. (Default  $n = 100$ .)
- p Make output periodic, i.e. match derivatives at ends. First and last input values should normally agree.
- x Next 1 (or 2) arguments are lower (and upper)  $x$  limits. Normally these limits are calculated from the data. Automatic abscissas start at lower limit (default 0).

**SEE ALSO**

*plot* (I)

**AUTHOR**

M. D. McIlroy

**BUGS**

A limit of 1000 input points is enforced silently.

**NAME**

tmg — compiler-compiler

**SYNOPSIS**

tmg name

**DESCRIPTION**

*Tmg* produces a translator for the language whose parsing and translation rules are described in file name.t. The new translator appears in a.out and may be used thus:

a.out input [ output ]

Except in rare cases input must be a randomly addressable file. If no output file is specified, the standard output file is assumed.

**FILES**

/sys/tmg/tmgl.o the compiler-compiler  
/sys/tmg[abc] libraries  
alloc.d table storage

**SEE ALSO**

A Manual for the Tmg Compiler-writing Language, internal memorandum.

**DIAGNOSTICS**

Syntactic errors result in "???" followed by the offending line.

Situations such as space overflow with which the Tmg processor or a Tmg-produced processor can not cope result in a descriptive comment and a dump.

**AUTHOR**

M. D. McIlroy

**BUGS**

9.2 footnote 1 is not enforced, causing trouble.

Restrictions (7.) against mixing bundling primitives should be listed.

Certain hidden reserved words exist: gpar, classtab, trans.

Octal digits include 8=10 and 9=11.

**NAME**

ttt - tic-tac-toe

**SYNOPSIS**

/usr/games/ttt

**DESCRIPTION**

*Ttt* is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to completely know the game.

**FILES**

/usr/games/ttt.k learning file

**BUGS**

**NAME**

wump — hunt the wumpus

**SYNOPSIS**

/usr/games/wump

**DESCRIPTION**

*Wump* plays the game of "Hunt the Wumpus." A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company*, 2, 2 (November 1973).

**BUGS**

It will never replace Space War.

**NAME**

yacc — yet another compiler-compiler

**SYNOPSIS**

**yacc [ -v ] [ grammar ]**

**DESCRIPTION**

*Yacc* converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output is *y.tab.c*, which must be compiled by the C compiler and loaded with any other routines required (perhaps a lexical analyzer) and the Yacc library:

```
cc y.tab.c other.o -ly
```

If the **-v** flag is given, the file *y.output* is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

**SEE ALSO**

"LR Parsing", by A. V. Aho and S. C. Johnson, to appear in Computing Surveys. "The YACC Compiler-compiler", internal memorandum.

**AUTHOR**

S. C. Johnson

**FILES**

|             |                              |
|-------------|------------------------------|
| y.output    |                              |
| y.tab.c     |                              |
| /lib/liby.a | runtime library for compiler |

**DIAGNOSTICS**

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the *y.output* file.

**BUGS**



**NAME**

ascii — map of ASCII character set

**SYNOPSIS**

cat /usr/pub/ascii

**DESCRIPTION***Ascii* is a map of the ASCII character set, to be printed as needed. It contains:

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 000 | nul | 001 | soh | 002 | stx | 003 | etx | 004 | eot | 005 | enq | 006 | ack | 007 | bel |
| 010 | bs  | 011 | ht  | 012 | nl  | 013 | vt  | 014 | np  | 015 | cr  | 016 | so  | 017 | si  |
| 020 | dle | 021 | dcl | 022 | dc2 | 023 | dc3 | 024 | dc4 | 025 | nak | 026 | syn | 027 | etb |
| 030 | can | 031 | em  | 032 | sub | 033 | esc | 034 | fs  | 035 | gs  | 036 | rs  | 037 | us  |
| 040 | sp  | 041 | !   | 042 | "   | 043 | #   | 044 | \$  | 045 | %   | 046 | &   | 047 | '   |
| 050 | (   | 051 | )   | 052 | *   | 053 | +   | 054 | ,   | 055 | -   | 056 | .   | 057 | /   |
| 060 | 0   | 061 | 1   | 062 | 2   | 063 | 3   | 064 | 4   | 065 | 5   | 066 | 6   | 067 | 7   |
| 070 | 8   | 071 | 9   | 072 | :   | 073 | :   | 074 | <   | 075 | =   | 076 | >   | 077 | ?   |
| 100 | @   | 101 | A   | 102 | B   | 103 | C   | 104 | D   | 105 | E   | 106 | F   | 107 | G   |
| 110 | H   | 111 | I   | 112 | J   | 113 | K   | 114 | L   | 115 | M   | 116 | N   | 117 | O   |
| 120 | P   | 121 | Q   | 122 | R   | 123 | S   | 124 | T   | 125 | U   | 126 | V   | 127 | W   |
| 130 | X   | 131 | Y   | 132 | Z   | 133 | [   | 134 | \   | 135 | ]   | 136 | ^   | 137 | _   |
| 140 |     | 141 | a   | 142 | b   | 143 | c   | 144 | d   | 145 | e   | 146 | f   | 147 | g   |
| 150 | h   | 151 | i   | 152 | j   | 153 | k   | 154 | l   | 155 | m   | 156 | n   | 157 | o   |
| 160 | p   | 161 | q   | 162 | r   | 163 | s   | 164 | t   | 165 | u   | 166 | v   | 167 | w   |
| 170 | x   | 171 | y   | 172 | z   | 173 | {   | 174 |     | 175 | }   | 176 | -   | 177 | del |

**FILES**

found in /usr/pub

**NAME**

greek — graphics for extended ascii type-box

**SYNOPSIS**

**cat /usr/pub/greek**

**DESCRIPTION**

*Greek* gives the mapping from ascii to the "shift out" graphics in effect between SO and SI on model 37 Teletypes with a 128-character type-box. It contains:

|         |            |   |          |          |   |        |           |   |
|---------|------------|---|----------|----------|---|--------|-----------|---|
| alpha   | $\alpha$   | A | beta     | $\beta$  | B | gamma  | $\gamma$  | \ |
| GAMMA   | $\Gamma$   | G | delta    | $\delta$ | D | DELTA  | $\Delta$  | W |
| epsilon | $\epsilon$ | S | zeta     | $\zeta$  | Q | eta    | $\eta$    | N |
| THETA   | $\Theta$   | T | theta    | $\theta$ | O | lambda | $\lambda$ | L |
| LAMBDA  | $\Lambda$  | E | mu       | $\mu$    | M | nu     | $\nu$     | @ |
| xi      | $\xi$      | X | pi       | $\pi$    | J | PI     | $\Pi$     | P |
| rho     | $\rho$     | K | sigma    | $\sigma$ | Y | SIGMA  | $\Sigma$  | R |
| tau     | $\tau$     | I | phi      | $\phi$   | U | PHI    | $\Phi$    | F |
| psi     | $\psi$     | V | PSI      | $\Psi$   | H | omega  | $\omega$  | C |
| OMEGA   | $\Omega$   | Z | nabla    | $\nabla$ | [ | not    | $\neg$    | - |
| partial | $\partial$ | ] | integral | $\int$   | ^ |        |           |   |

**SEE ALSO**

ascii (VII)

**NAME**  
tabs — set tab stops

**SYNOPSIS**  
cat /usr/pub/tabs

**DESCRIPTION**  
When printed on a suitable terminal, this file will set tab stops every 8 columns. Suitable terminals include the Teletype model 37 and the GE TermiNet 300.  
These tab stop settings are desirable because UNIX assumes them in calculating delays.

**NAME**

tmheader — TM cover sheet

**SYNOPSIS**

ed /usr/pub/tmheader

**DESCRIPTION**

*/usr/pub/tmheader* contains a prototype for making a *troff(1)* formatted cover sheet for a technical memorandum. Parameters to be filled in by the user are marked by self-explanatory names beginning with “—”.

**BUGS**

God help you on two-page abstracts. Try to write less.

**NAME**

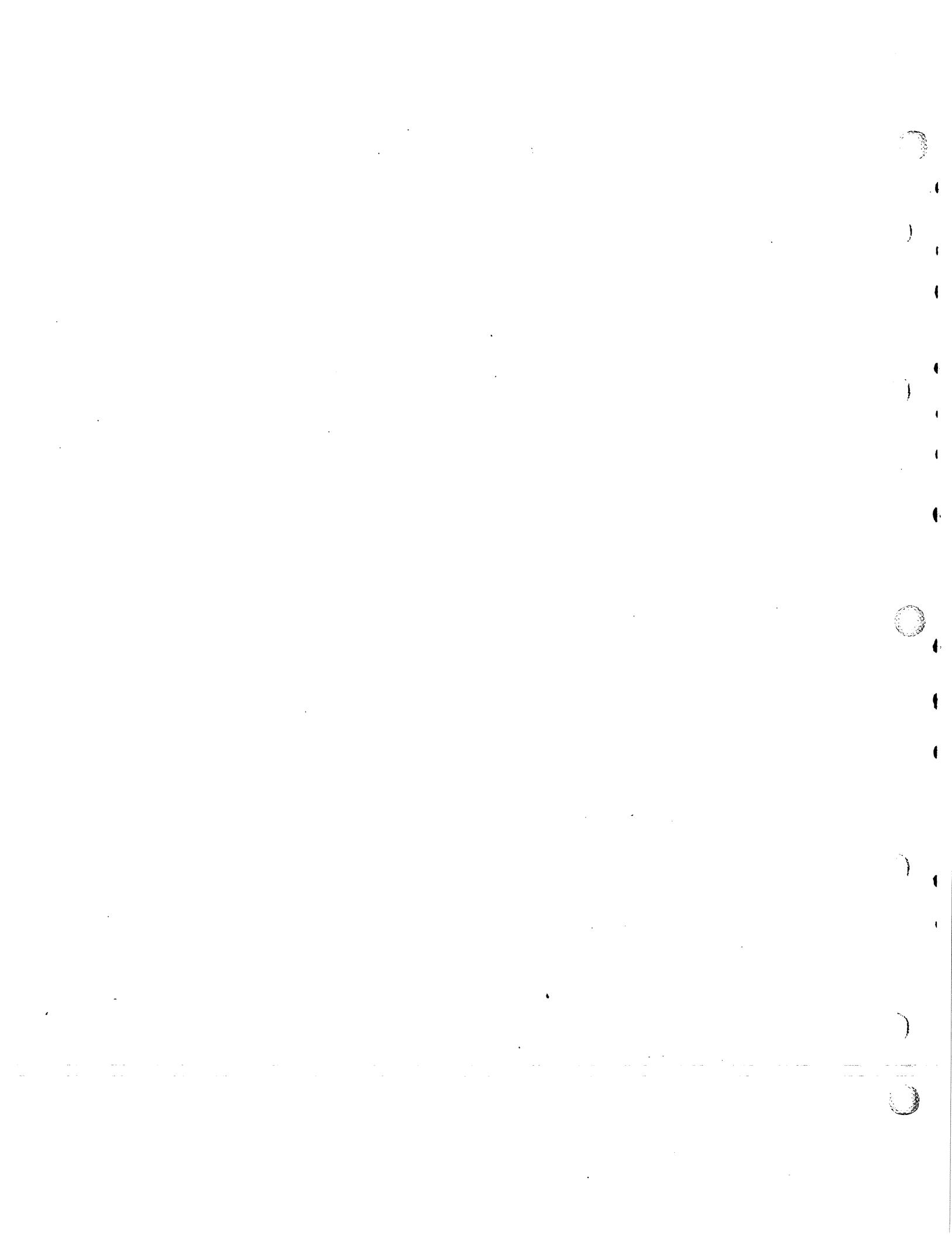
vs — voice synthesizer code

**DESCRIPTION**

The octal codes below are understood by the Votrax® voice synthesizer. Inflection and phonemes are or-ed together. The mnemonics in the first column are used by *speak* (VI); the upper case mnemonics are used by the manufacturer.

|    |     |                     |    |     |                     |
|----|-----|---------------------|----|-----|---------------------|
| 0  | 300 | 4—strong inflection | u0 | 014 | UH—but              |
| 1  | 200 | 3                   | u1 | 015 | UH1—uncle           |
| 2  | 100 | 2                   | u2 | 016 | UH2—stirrup         |
| 3  | 000 | 1—weak inflection   | u3 | 034 | UH3—app_le ab_le    |
| a0 | 033 | AH—contact          | yu | 027 | U—use               |
| a1 | 052 | AH1—connect         | iu | 010 | U1—unite(y1,iu,...) |
| aw | 002 | AW—law(l,u2,aw)     | ju | 011 | IU—new              |
| au | 054 | AW1—fault           | b  | 061 | B                   |
| ae | 021 | AE—cat              | d  | 041 | D                   |
| ea | 020 | AE1—antenna         | f  | 042 | F                   |
| ai | 037 | A—name(n,ai,y0,m)   | g  | 043 | G                   |
| aj | 071 | A1—namely           | h  | 044 | H                   |
| e0 | 004 | EH—met enter        | k  | 046 | K                   |
| e1 | 076 | EH1—seven           | l  | 047 | L                   |
| e2 | 077 | EH2—seven           | m  | 063 | M                   |
| er | 005 | ER—weather          | n  | 062 | N                   |
| eu | 073 | OOH—Goethe cheveux  | p  | 032 | P                   |
| eh | 067 | EHH—le cheveux      | q  | 075 | Q                   |
| y0 | 023 | EE—three            | r  | 024 | R                   |
| y1 | 026 | Y—sixty             | s  | 040 | S                   |
| y2 | 035 | Y1—yes              | t  | 025 | T                   |
| ay | 036 | AY—may              | v  | 060 | V                   |
| i0 | 030 | I—six               | w  | 022 | W                   |
| i1 | 064 | I1—inept inside     | z  | 055 | Z                   |
| i2 | 065 | I2—static           | sh | 056 | SH—show ship        |
| iy | 066 | IY—cry(k,r,a0,iy)   | zh | 070 | ZH—pleasure         |
| ie | 003 | IE—zero             | j  | 045 | J—edge              |
| ih | 072 | IH—station          | ch | 057 | CH—batch            |
| o0 | 031 | O—only no           | th | 006 | TH—thin             |
| o1 | 012 | O1—hello            | dh | 007 | THV—then            |
| o2 | 013 | O2—notice           | ng | 053 | NG—long ink         |
| ou | 051 | OO1—good should     | -0 | 017 | PA2—long pause      |
| oo | 050 | OO—look             | -1 | 001 | PA1                 |
|    |     |                     | -2 | 074 | PA0—short pause     |

**SEE ALSO***speak* (VI), vs (IV)



**NAME**

20boot — install new 11/20 system

**SYNOPSIS**

**20boot**

**DESCRIPTION**

This shell command file copies the current version of the 11/20 program used to run the VT01 display onto the /dev/vt0 file. The 11/20 should have been started at its ROM location 773000.

**FILES**

/dev/vt0, /usr/mdec/20.o (11/20 program)

**SEE ALSO**

vt (IV)

**NAME**

ac — login accounting

**SYNOPSIS**

ac [ -w wtmp ] [ -p ] [ -d ] people

**DESCRIPTION**

*Ac* produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. **-w** is used to specify an alternate *wtmp* file. **-p** prints individual totals; without this option, only totals are printed. **-d** causes a printout for each midnight to midnight period. Any *people* will limit the printout to only the specified login names. If no *wtmp* file is given, */usr/adm/wtmp* is used.

The accounting file */usr/adm/wtmp* is maintained by *init* and *login*. Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

**FILES**

*/usr/adm/wtmp*

**SEE ALSO**

*init* (VIII), *login* (I), *wtmp* (V).

**BUGS**

**NAME**

boot procedures – UNIX startup

**DESCRIPTION**

*How to start UNIX.* UNIX is started by placing it in core starting at location zero and transferring to zero. There are various ways to do this. If UNIX is still intact after it has been running, the most obvious method is simply to transfer to zero, but this is not recommended if the system has crashed.

The *tp* command places a bootstrap program on the otherwise unused block zero of the tape. The DECtape version of this program is called *tboot*, the magtape version *mboot*. If *tboot* or *mboot* is read into location zero and executed there, it will type '=' on the console, read in a *tp* entry name, load that entry into core, and transfer to zero. Thus the next easiest way to run UNIX is to maintain the UNIX code on a tape using *tp*. Then when a boot is required, execute (somehow) a program which reads in and jumps to the first block of the tape. In response to the '=' prompt, type the entry name of the system on the tape (we use plain 'unix'). It is strongly recommended that a current version of the system be maintained in this way, even if the first or third methods of booting the system are usually used.

The standard DEC ROM which loads DECtape is sufficient to read in *tboot*, but the magtape ROM loads block one, not zero. If no suitable ROM is available, magtape and DECtape programs are presented below which may be manually placed in core and executed.

A third method of rebooting the system involves the otherwise unused block zero of each UNIX file system. The single-block program *uboot* will read a UNIX pathname from the console, find the corresponding file on a device, load that file into core location zero, and transfer to it. The current version of this boot program reads a single character (either p or k for RP or RK, both drive 0) to specify which device is to be searched. *Uboot* operates under very severe space constraints. It supplies no prompts, except that it echos a carriage return and line feed after the p or k. No diagnostic is provided if the indicated file cannot be found, nor is there any means of correcting typographical errors in the file name except to start the program over. *Uboot* can reside on any of the standard file systems or may be loaded from a *tp* tape as described above.

The standard DEC disk ROMs will load and execute *uboot* from block zero.

*The switches.* The console switches play an important role in the use and especially the booting of UNIX. During operation, the console switches are examined 60 times per second, and the contents of the address specified by the switches are displayed in the display register. (This is not true on the 11/40 since there is no display register on that machine.) If the switch address is even, the address is interpreted in kernel (system) space; if odd, the rounded-down address is interpreted in the current user space.

If any diagnostics are produced by the system, they are printed on the console only if the switches are non-zero. Thus it is wise to have a non-zero value in the switches at all times.

During the startup of the system, the *init* program (VIII) reads the switches and will come up single-user if the switches are set to 173030.

It is unwise to have a non-existent address in the switches. This causes a bus error in the system (displayed as 177777) at the rate of 60 times per second. If there is a transfer of more than 16ms duration on a device with a data rate faster than the bus error timeout (about 10 $\mu$ s) then a permanent disk non-existent-memory error will occur.

*ROM programs.* Here are some programs which are suitable for installing in read-only memories, or for manual keying into core if no ROM is present. Each program is position-independent but should be placed well above location 0 so it will not be overwritten. Each reads a block from the beginning of a device into core location zero. The octal words constituting the program are listed on the left.

## DECtape (drive 0) from endzone:

|        |     |               |                                 |
|--------|-----|---------------|---------------------------------|
| 012700 | mov | \$tcba,r0     |                                 |
| 177346 |     |               |                                 |
| 010040 | mov | r0,-(r0)      | / use tc addr for wc            |
| 012710 | mov | \$3,(r0)      | / read bn forward               |
| 000003 |     |               |                                 |
| 105710 | 1:  | tstb (r0)     | / wait for ready                |
| 002376 |     | bge 1b        |                                 |
| 112710 |     | movb \$5,(r0) | / read (forward)                |
| 000005 |     |               |                                 |
| 000777 |     | br            | / loop; now halt and start at 0 |

## DECtape (drive 0) with search:

|        |    |               |                        |                      |
|--------|----|---------------|------------------------|----------------------|
| 012700 | 1: | mov           | \$tcba,r0              |                      |
| 177346 |    |               |                        |                      |
| 010040 |    | mov           | r0,-(r0)               | / use tc addr for wc |
| 012740 |    | mov           | \$4003,-(r0)           | / read bn reverse    |
| 004003 |    |               |                        |                      |
| 005710 | 2: | tst (r0)      |                        |                      |
| 002376 |    | bge 2b        | / wait for error       |                      |
| 005760 |    | tst -2(r0)    | / loop if not end zone |                      |
| 177776 |    |               |                        |                      |
| 002365 |    | bge 1b        |                        |                      |
| 012710 |    | mov \$3,(r0)  | / read bn forward      |                      |
| 000003 |    |               |                        |                      |
| 105710 | 2: | tstb (r0)     | / wait for ready       |                      |
| 002376 |    | bge 2b        |                        |                      |
| 112710 |    | movb \$5,(r0) | / read (forward)       |                      |
| 000005 |    |               |                        |                      |
| 105710 | 2: | tstb (r0)     | / wait for ready       |                      |
| 002376 |    | bge 2b        |                        |                      |
| 005007 |    | clr pc        | / transfer to zero     |                      |

Caution: both of these DECtape programs will (literally) blow a fuse if 2 drives are dialed to zero.

## Magtape from load point:

|        |     |               |                                 |
|--------|-----|---------------|---------------------------------|
| 012700 | mov | \$mtcma,r0    |                                 |
| 172526 |     |               |                                 |
| 010040 | mov | r0,-(r0)      | / usr mt addr for wc            |
| 012740 | mov | \$60003,-(r0) | / read 9-track                  |
| 060003 |     |               |                                 |
| 000777 | br  |               | / loop; now halt and start at 0 |

## RK (drive 0):

|        |     |           |                         |
|--------|-----|-----------|-------------------------|
| 012700 | mov | \$rkda,r0 |                         |
| 177412 |     |           |                         |
| 005040 | clr | -(r0)     | / rkda cleared by start |
| 010040 | mov | r0,-(r0)  |                         |
| 012740 | mov | \$5,-(r0) |                         |
| 000005 |     |           |                         |
| 105710 | 1:  | tstb (r0) |                         |
| 002376 |     | bge 1b    |                         |
| 005007 |     | clr pc    |                         |

## RP (drive 0)

|        |     |           |  |
|--------|-----|-----------|--|
| 012700 | mov | \$rpmr,r0 |  |
| 176726 |     |           |  |
| 005040 | clr | -(r0)     |  |
| 005040 | clr | -(r0)     |  |
| 005040 | clr | -(r0)     |  |

|        |    |      |           |
|--------|----|------|-----------|
| 010040 |    | mov  | r0,-(r0)  |
| 012740 |    | mov  | \$5,-(r0) |
| 000005 |    |      |           |
| 105710 | 1: | tstb | (r0)      |
| 002376 |    | bge  | 1b        |
| 005007 |    | clr  | pc        |

**FILES**

/unix - UNIX code  
/usr/mdec/mboot - *tp* magtape bootstrap  
/usr/mdec/tboot - *tp* DECTape bootstrap  
/usr/mdec/u-boot - file system bootstrap

**SEE ALSO**

*tp* (I), init (VIII)

**NAME**

**check** — file system consistency check

**SYNOPSIS**

**check [ -lsvub [ numbers ] ] [ filesystem ]**

**DESCRIPTION**

*Check* examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. It also reads directories and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a check of a default file system is performed. The normal output of *check* includes a report of

- The number of blocks missing; i.e. not in any file nor in the free list,
- The number of special files,
- The total number of files,
- The number of large files,
- The number of directories,
- The number of indirect blocks,
- The number of blocks used in files,
- The highest-numbered block appearing in a file,
- The number of free blocks.

The **-l** flag causes *check* to produce as part of its output report a list of all the path names of files on the file system. The list is in i-number order; the first name for each file gives the i-number while subsequent names (i.e. links) have the i-number suppressed. The entries **“.”** and **“..”** for directories are also suppressed. If the flag is given as **-ll**, the listing will include the accessed and modified times for each file. The **-l** option supersedes **-s**.

The **-s** flag causes *check* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The **-s** flag causes the normal output reports to be suppressed.

With the **-u** flag, *check* examines the directory structure for connectivity. A list of all i-node numbers that cannot be reached from the root is printed. This is exactly the list of i-nodes that should be cleared (see *clri* (VIII)) after a series of incremental restores. (See the bugs section of *restor* (VIII).) The **-u** option supersedes **-s**.

The occurrence of **i n** times in a flag argument **-ii...i** causes *check* to store away the next **n** arguments which are taken to be i-numbers. When any of these i-numbers is encountered in a directory a diagnostic is produced, as described below, which indicates among other things the entry name.

Likewise, **n** appearances of **b** in a flag like **-bb...b** cause the next **n** arguments to be taken as block numbers which are remembered; whenever any of the named blocks turns up in a file, a diagnostic is produced.

**FILES**

Currently, **/dev/rp0** is the default file system.

**SEE ALSO**

**fs** (V), **clri** (VIII), **restor**(VIII)

**DIAGNOSTICS**

If a read error is encountered, the block number of the bad block is printed and *check* exits. “Bad freeblock” means that a block number outside the available space was encountered in the free list. “**n** dups in free” means that **n** blocks were found in the free list which duplicate blocks

either in some file or in the earlier part of the free list.

An important class of diagnostics is produced by a routine which is called for each block which is encountered in an i-node corresponding to an ordinary file or directory. These have the form

*b# complaint ; i= i# (class )*

Here *b#* is the block number being considered; *complaint* is the diagnostic itself. It may be

- blk** if the block number was mentioned as an argument after **-b**;
- bad** if the block number has a value not inside the allocatable space on the device, as indicated by the device's super-block;
- dup** if the block number has already been seen in a file;
- din** if the block is a member of a directory, and if an entry is found therein whose i-number is outside the range of the i-list on the device, as indicated by the i-list size specified by the super-block. Unfortunately this diagnostic does not indicate the offending entry name, but since the i-number of the directory itself is given (see below) the problem can be tracked down.

The *i#* in the form above is the i-number in which the named block was found. The *class* is an indicator of what type of block was involved in the difficulty:

- sdir** indicates that the block is a data block in a small file;
- ldir** indicates that the block is a data block in a large file (the indirect block number is not available);
- idir** indicates that the block is an indirect block (pointing to data blocks) in a large file;
- free** indicates that the block was mentioned after **-b** and is free;
- urk** indicates a malfunction in *check*.

When an i-number specified after **-i** is encountered while reading a directory, a report in the form

*i# ino; i= d# (class ) name*

where *i#* is the requested i-number, *d#* is the i-number of the directory, *class* is the class of the directory block as discussed above (virtually always "sdir") and *name* is the entry name. This diagnostic gives enough information to find a full path name for an i-number without using the **-l** option: use **-b n** to find an entry name and the i-number of the directory containing the reference to *n*, then recursively use **-b** on the i-number of the directory to find its name.

Another important class of file system diseases indicated by *check* is files for which the number of directory entries does not agree with the link-count field of the i-node. The diagnostic is hard to interpret. It has the form

*i# delta*

Here *i#* is the i-number affected. *Delta* is an octal number accumulated in a byte, and thus can have the value 0 through 377(8). The easiest way (short of rewriting the routine) of explaining the significance of *delta* is to describe how it is computed.

If the associated i-node is allocated (that is, has the *allocated* bit on) add 100 to *delta*. If its link-count is non-zero, add another 100 plus the link-count. Each time a directory entry specifying the associated i-number is encountered, subtract 1 from *delta*. At the end, the i-number and *delta* are printed if *delta* is neither 0 nor 200. The first case indicates that the i-node was unallocated and no entries for it appear; the second that it was allocated and that the link-count and the number of directory entries agree.

Therefore (to explain the symptoms of the most common difficulties) *delta* = 377 (-1 in 8-bit, 2's complement octal) means that there is a directory entry for an unallocated i-node. This is somewhat serious and the entry should be found and removed forthwith. *Delta* = 201 usually means that a normal, allocated i-node has no directory entry. This difficulty is much less serious. Whatever blocks there are in the file are unavailable, but no further damage will occur if nothing is done. A *crls* followed by a *check -s* will restore the lost space at leisure.

In general, values of *delta* equal to or somewhat above 0, 100, or 200 are relatively innocuous; just below these numbers there is danger of spreading infection.

**BUGS**

*Check -l or -u* on large file systems takes a great deal of core.

Since *check* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

It believes even preposterous super-blocks and consequently can get core images.

**NAME**

clri — clear i-node

**SYNOPSIS**

clri *i-number* [ *filesystem* ]

**DESCRIPTION**

*Clri* writes zeros on the 32 bytes occupied by the i-node numbered *i-number*. If the *file system* argument is given, the i-node resides on the given device, otherwise on a default file system. The file system argument must be a special file name referring to a device containing a file system. After *clri*, any blocks in the affected file will show up as "missing" in a *check* of the file system.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

**BUGS**

Whatever the default file system is, it is likely to be wrong. Specify the file system explicitly.

If the file is open, *clri* is likely to be ineffective.

**NAME**

df — disk free

**SYNOPSIS**

**df [ filesystem ]**

**DESCRIPTION**

*Df* prints out the number of free blocks available on a file system. If the file system is unspecified, the free space on all of the normally mounted file systems is printed.

**FILES**

/dev/rf?, /dev/rk?, /dev/rp?

**SEE ALSO**

check (VIII)

**BUGS**

**NAME**

dpd — spawn data phone daemon

**SYNOPSIS**

/etc/dpd

**DESCRIPTION**

*Dpd* is the 201 data phone daemon. It is designed to submit jobs to the Honeywell 6070 computer via the GRITS interface.

*Dpd* uses the directory /usr/dpd. The file lock in that directory is used to prevent two daemons from becoming active. After the daemon has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with df. Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

- S directs *dpd* to generate a unique snumb card. This card is generated by incrementing the first word of the file /usr/dpd/snnumb and converting that to three-digit octal concatenated with the station ID.
- L specifies that the remainder of the line is to be sent as a literal.
- B specifies that the rest of the line is a file name. That file is to be sent as binary cards.
- F is the same as B except a form feed is prepended to the file.
- U specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.

Any error encountered will cause the daemon to drop the call, wait up to 20 minutes and start over. This means that an improperly constructed df file may cause the same job to be submitted every 20 minutes.

While waiting, the daemon checks to see that the lock file still exists. If it is gone, the daemon will exit.

**FILES**

/dev/dn0, /dev/dp0, /usr/dpd/\*

**SEE ALSO**

opr (I)

**NAME**

dump — incremental file system dump

**SYNOPSIS**

dump [ key [ arguments ] filesystem ]

**DESCRIPTION**

*Dump* will make an incremental file system dump on magtape of all files changed after a certain date. The argument *key*, specifies the date and other options about the dump. *Key* consists of characters from the set **iu0hds**.

- i** the dump date is taken from the file */etc/ddate*.
- u** the date just prior to this dump is written on */etc/ddate* upon successful completion of this dump.
- 0** the dump date is taken as the epoch (beginning of time). Thus this option causes an entire file system dump to be taken.
- h** the dump date is some number of hours before the current date. The number of hours is taken from the next argument in *arguments*.
- d** the dump date is some number of days before the current date. The number of days is taken from the next argument in *arguments*.
- s** the size of the dump tape is specified in feet. The number of feet is taken from the next argument in *arguments*. It is assumed that there are 9 standard UNIX records per foot. When the specified size is reached, the dump will wait for reels to be changed. The default size is 1700 feet.

If no arguments are given, the *key* is assumed to be **i** and the file system is assumed to be */dev/rp0*.

Full dumps should be taken on quiet file systems as follows:

```
dump 0u /dev/rp0
check -l /dev/rp0
```

The *check* will come in handy in case it is necessary to restore individual files from this dump. Incremental dumps should then be taken when desired by:

```
dump
```

When the incremental dumps get cumbersome, a new complete dump should be taken. In this way, a restore requires loading of the complete dump tape and only the latest incremental tape.

**FILES**

|            |                     |
|------------|---------------------|
| /dev/mt0   | magtape             |
| /dev/rp0   | default file system |
| /etc/ddate |                     |

**SEE ALSO**

restor (VIII), check (VIII), dump (V)

**BUGS**

**NAME**

getty — set typewriter mode

**SYNOPSIS**

/etc/getty

**DESCRIPTION**

*Getty* is invoked by init (VIII) immediately after a typewriter is opened following a dial-up. The user's login name is read and the *login* (I) command is called with this name as an argument. While reading this name *getty* attempts to adapt the system to the speed and type of terminal being used.

*Getty* initially sets the speed of the interface to 300 baud, specifies that raw mode is to be used (break on every character), that echo is to be suppressed, and either parity allowed. It types the "login:" message (which includes the characters which put the Terminet 300 terminal into full-duplex and return the GSI terminal to non-graphic mode. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the "break" ("interrupt") key. The speed is then changed to 150 baud and the "login:" is typed again, this time including the character sequence which puts a Teletype 37 into full-duplex. If a subsequent null character is received, the speed is changed back to 300 baud.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *stty* (II)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters. Thus UNIX is usable from upper-case-only terminals.

Finally, *login* is called with the user's name as argument.

**SEE ALSO**

init (VIII), *login* (I), *stty* (II)

**BUGS**

**NAME**

**init** — process control initialization

**SYNOPSIS**

/etc/init

**DESCRIPTION**

*Init* is invoked inside UNIX as the last step in the boot procedure. Generally its role is to create a process for each typewriter on which a user may log in.

First, *init* checks to see if the console switches contain 173030. (This number is likely to vary between systems.) If so, the console typewriter /dev/tty8 is opened for reading and writing and the Shell is invoked immediately. This feature is used to bring up a single-user system. When the system is brought up in this way, the *getty* and *login* routines mentioned below and described elsewhere are not used. If the Shell terminates, *init* starts over looking for the console switch setting.

Otherwise, *init* invokes a Shell, with input taken from the file /etc/rc. This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then *init* reads the file /etc/ttys and forks several times to create a process for each typewriter specified in the file. Each of these processes opens the appropriate typewriter for reading and writing. These channels thus receive file descriptors 0 and 1, the standard input and output. Opening the typewriter will usually involve a delay, since the *open* is not completed until someone is dialed up and carrier established on the channel. Then the process executes the program specified by its line in ttys; the only program currently specifiable is /etc/getty (q.v.). *Getty* reads the user's name and invokes *login* (q.v.) to log in the user and execute the Shell.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of *init*, which has been waiting for such an event, wakes up and removes the appropriate entry from the file utmp, which records current users, and makes an entry in /usr/adm/wtmp, which maintains a history of logins and logouts. Then the appropriate typewriter is reopened and *getty* is reinvoked.

*Init* catches the *hangup* signal (signal #1) and interprets it to mean that the switches should be examined as in a reboot: if they indicate a multi-user system, the /etc/ttys file is read again. The Shell process on each line which used to be active in ttys but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add phone lines without rebooting the system by changing the ttys file and sending a *hangup* signal to the *init* process: use "kill -1 1."

**FILES**

/dev/tty?, /tmp/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

**SEE ALSO**

login (I), kill (I), sh (I), ttys (V), getty (VIII)

**NAME**

lpd — line printer daemon

**SYNOPSIS**

/etc/lpd

**DESCRIPTION**

*Lpd* is the line printer daemon (spool area handler) invoked by *lpr*. It uses the directory */usr/lpd*. The file *lock* in that directory is used to prevent two daemons from becoming active simultaneously. After the daemon has successfully set the lock, it scans the directory for files beginning with "df." Lines in each *df* file specify files to be printed in the same way as is done by the data-phone daemon *dpd* (VIII).

**FILES**

|            |            |
|------------|------------|
| /usr/lpd/* | spool area |
| /dev/lp    | printer    |

**SEE ALSO**

*dpd* (III), *lpr* (VI)

**BUGS**

**NAME**

mount — mount file system

**SYNOPSIS**

/etc/mount special file [ -r ]

**DESCRIPTION**

*Mount* announces to the system that a removable file system is present on the device corresponding to special file *special* (which must refer to a disk or possibly DECtape). The *file* must exist already; it becomes the name of the root of the newly mounted file system.

*Mount* maintains a table of mounted devices; if invoked without an argument it prints the table.

The optional last argument indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

**SEE ALSO**

mount (II), mtab (V), umount (VIII)

**BUGS**

Mounting file systems full of garbage will crash the system.

**NAME**

msh — mini-shell

**SYNOPSIS**

/etc/msh

**DESCRIPTION**

*Msh* is a heavily simplified version of the Shell. It reads one line from the standard input file, interprets it as a command, and calls the command.

The mini-shell supports few of the advanced features of the Shell; none of the following characters is special:

> < \$ \ ; & | ^

However, “\*”, “[”, and “?” are recognized and *glob* is called. The main use of *msh* is to provide a command-executing facility for various interactive sub-systems.

**SEE ALSO**

sh (I), glob (VIII)

**NAME**

**sync** — update the super block

## SYNOPSIS

sync

## **DESCRIPTION**

*Sync* executes the `sync` system primitive. If the system is to be stopped, `sync` must be called to insure file system integrity. See `sync(II)` for details.

**SEE ALSO**

5

**NAME**

umount — dismount file system

**SYNOPSIS**

/etc/umount special

**DESCRIPTION**

*Umount* announces to the system that the removable file system previously mounted on special file *special* is to be removed.

**SEE ALSO**

mount (VIII), umount (II), mtab (V)

**FILES**

/etc/mtab mounted device table

**DIAGNOSTICS**

It complains if the special file is not mounted or if it is busy. The file system is busy if there is an open file on it or if someone has his current directory there.

**BUGS**

**NAME**

**update** — periodically update the super block

**SYNOPSIS**

**update**

**DESCRIPTION**

*Update* is a program that executes the *sync* primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file. See *sync* (II) for details.

**SEE ALSO**

*sync* (II), *init* (VIII)

**BUGS**

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when DPR requests fail. A fix would be to have *sync* temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.