C In Depth

Pointers and Addresses

A pointer is a variable that contains the address of a variable. Pointers and arrays are closely related.

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.

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.)

The `&` Operator

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 `*` Operator

The unary operator * is the indirection or dereferencing operator; when applied to

a pointer, it accesses the object the pointer points to.

```
int *x;
```

Declares `x` as a being a pointer of type `int`.

```
x = &a;
```

Makes `x` point to the memory address of where the value of the variable `a` is stored.

```
*x = 10
```

will now change the value of `a`. Must be used with `*`, otherwise C will throw an error.

I.e. If `x` points to the integer `a`, then `*x` can occur in any context where `a` could. E.g.

```
(*x)++;
++*x;
```

The parentheses are necessary in the top example; without them, the expression would increment `x` instead of what it points to, because unary operators like * and ++ associate right to left.

In functions `atof(char *)` says that the argument of `atof` is a pointer to `char`.

Since pointers are variables, they can be used without dereferencing. For example, if `y` is another pointer to `int`,

```
y = x;
```

copies the contents of `x` into `y`, thus making `y` point to whatever `x` pointed to.

Pointers and Functions 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.

Pointer arguments enable a function to access and change objects in the function that called it (e.g. main).

Pointers and Arrays

In C, there is a strong relationship between pointers and arrays. 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.

```
int a[10], *x;
x = &a[0];
```

This means that `x` points to `a[0]`, so using `*x` will change `a[0]`.

Furthermore, `*(x+1)` refers to the contents of `a[1]`, so `x+i` is the address of `a[i]`, and `*(x+i)` is the contents of `a[i]`.

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. So:

```
x = &a[0];
```

can also be written as

```
x = a;
```

Even more surprising is that `a[i]` can also be written as `*(a+i)`. In evaluating a[i], C converts it to *(a+i) immediately. So `&a[i]` and `a+i` are

identical.

Conversely, if `x` is a pointer to `a[i]` as above, `x[i]` is identical to `*(x+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.

As formal parameters in a function definition, `char s[]` and `char *s` are equivalent.

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.

Address Arithmetic

If p is a pointer to some element of an array, then p++ increments p to point to the next element.

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>.

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.

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.

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.

A string can be initialized by one of the following methods:

```
char d[] = "hello dude";        #an array
char *s =  "hello snood";       #a pointer
```

`d[]` is an array that contains the string `*s` is a pointer that points to a string

To mess around with the strings via pointers, use:

```
while (*s != '\0') {
    printf("%c", *s);
    s++;
}
```

Using `*s` in the `printf` statement means that you're targeting the contents of the specific character or array index that `s` is currently pointing to.

Conversely, `s` is the pointer to the beginning of the string, so if you just wanted to print out the string without a loop, you'd need to use `s` without the `*`, like so:

```
printf("%s", s);
```

You could also try to write the above loop as:

```
while (*s++ != '\0')
    printf("%c", s)         #will start 1 letter after 1st
```

Since we're using `*s++`, and not `++(*s)`, the value of `*s` is retrieved first, and THEN it is incremented. BUT... because the `*s++` is executed in the test before we perform the first `printf` statement, it means that `*s` already points to the second position in the string by the time you read the `printf` statement.

You could also omit the `= '\0'` since it is essentially redundant, because loops test for zero as a failure, so you could just rewrite the loop as:

```
while (*s) {
    printf("%c", *s);
    s++;
}
```

Other ways to write loops using pointers and arrays:

```
 for ( ; *s == *t; s++, t++)
 for (i = 0; s[i] == t[i]; i++)
```

Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, they can be stored in arrays just as other variables can.

Need help with this one

To initialize the array:

```
char *names[] = {
    "ben",
    "bob",
    "shamir",
    "ralph"
};
```

Since the size of the array name is not specified, the compiler counts the initializers and fills in the correct number.

Multi-dimensional Arrays

In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array.

```
daytab[i][j]    /* [row][col] */
```

Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

To declare and initialize a multi-dimensional array:

```
int days[2][5];         #declaration
days[1][5] = 22;        #initialization
```

The difference between pointer arrays and multi-dimensional arrays

The important advantage of the pointer array over a regular multi-dimensional array is that the rows of the array may be of different lengths. That is, each element of `a` need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all.

```
//Declare variables
char *a[] = {
    "hello there",
    "how are you today",
    "what is your name"
};

printf("\nanswer: %c\n", a[1][2]);
```

The difference is that you can't set a mult-dimensional array like we do above, so that it expands to fit each string in a new array position, and each letter of each string in a sub-array position automatically. I.e. you can't do:

```
a[][] = {
    "hello",
    "how are you",
    "nice tie"
};
```

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.

This is a huge chapter, and I don't want to write notes about it all. To learn all about it, go to Page 105.

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.

When you have `void *` as a parameter, it will take any datatype.

`*` is a prefix operator and it has lower precedence than `()`, so parentheses are necessary to force the proper association.

Structures (precursor to objects?)

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. 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.

Structures may be copied and assigned to, passed to functions, and returned by functions.

To declare a structure

```
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 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;
```

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.

```
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 };
```

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. E.g.

```
printf("%d,%d", pt.x, 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;
   };

screen.pt1.x
```

`screen.pt1.x` refers to the x coordinate of the pt1 member of screen.

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.

When working with structures and functions, there are at least three possible approaches: pass components separately, pass an entire structure, or pass a pointer to it.

```
/* 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;
}
```

This is a function that returns a point structure and takes 2 ints.

Notice that there is no conflict between the argument names and the members with the same names. Indeed the re-use of the names stresses the relationship.

`makepoint` can now be used to initialize any structure dynamically

Structures and Pointers

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. E.g.

```
struct point origin, *pp;

pp = &origin;
```

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.

The parentheses are necessary in `(*pp).x` because the precedence of the structure member operator `.` is higher then `*`. 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. E.g.

```
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
```

(++p)->len increments p before accessing len, and (p++)->len increments p afterward. (This last set of parentheses is unnecessary.)

`*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.

Arrays of Structures

The structure declaration:

```
struct key {
    char *word;
    int count;
};

struct key 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.

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
}
```

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.

Self-referential Structures

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.

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 occurances
    struct tnode *left;     #left child
    struct tnode *right;    #right child
};
```

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.

Advanced Info about `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.

Table Lookup

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.

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:

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;
```

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.

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.

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.

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 (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.

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.

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.

Input and Output (I/O)

The simplest input mechanism is to read one character at a time from the standard input, normally the keyboard, with getchar:

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.

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 another prog.

Scanf

`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.

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.

There is also a function sscanf that reads from a string instead of the standard

input:

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).
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 %1s
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.

```
scanf("%d %s %d", &day, monthname, &year);
```

No & is used with monthname, since an array name is a pointer.

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.

File Access

The next step is to write a program that accesses a file that is not already connected to the program.

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.

The call to fopen in a program is

```
fp = fopen(name, mode);
```

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'.

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. concatenate t to end of s strcat(s,t) strncat(s,t,n) concatenate n characters of t to

end of s return negative, zero, or positive for s < t, s == t, s > t strcmp(s,t) strncmp(s,t,n) same as strcmp but only in first n characters copyttos strncpy(s,t,n) copy at most n characters of t to s strlen(s) strcpy(s,t) return length of s 147 strchr(s,c) strrchr(s,c) return pointer to first c in s, or NULL if not present return pointer to last c in s, or NULL if not present 7.8.2 Character Class Testing and Conversion Several functions from <ctype.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. isalpha(c) non-zero if c is alphabetic, 0 if not isupper(c) non-zero if c is upper case, 0 if not islower(c) non-zero if c is lower case, 0 if not isdigit(c) non-zero if c is digit, 0 if not isalnum(c) non-zero if isalpha(c) or isdigit(c), 0 if not isspace(c) non-zero if c is blank, tab, newline, return, formfeed, vertical tab toupper(c) return c converted to upper case tolower(c) return c converted to lower case

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.

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. E.g.

```
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

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))
```

System Calls

System Calls are in functions within the operating system that may be called by user programs.

This chapter is divided into three major parts: input/output, file system, and storage allocation.

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.

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.