LECTUER NOTE 30

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";
char *pmessage = "now is the time";
```

`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:

```
void strcpy(char *s, char *t)
{
inti;
 i = 0;
while ((s[i] = t[i]) != '\0')
i++;
```

```
 }
```

For contrast, here is a version of `strcpy` with pointers:

```
void strcpy(char *s, char *t)
{
 inti;
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

```
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

```
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.

```
int strcmp(char *s, char *t)
```

```
{
 inti;
 for (i = 0; s[i] == t[i]; i++)
if (s[i] == '\0')
 return 0;
return s[i] - t[i];
}
```

The pointer version of strcmp:

```
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;

val = *--p;
```

are the standard idiom for pushing and popping a stack;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.

Pointer Arrays; Pointers to Pointers

Syntax to declare pointer to an array is

*datatype (*pointer_variable)[size];*

For example

int (*ptr)[10]; ,Here ptr is a pointer that can point to an array of 10 integers, where we can initialize ptr with the base address of the array then by incre menting the value of ptr we can access different elements of array a[].
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.

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 bee 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
char *lineptr[MAXLINES];
intreadlines(char *lineptr[], intnlines);
```

```
void writelines(char *lineptr[], intnlines);
void qsort(char *lineptr[], int left, int right);
main()
{
 intnlines;
 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
 intgetline(char *, int);
 char *alloc(int);
intreadlines(char *lineptr[], intmaxlines)
{
intlen, 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;
}
void writelines(char *lineptr[], intnlines)
{
inti;
 for (i = 0; i<nlines; i++)
printf("%s\n", lineptr[i]);
}
```

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

```
void writelines(char *lineptr[], intnlines)
{
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.

```
void qsort(char *v[], int left, int right)
{
inti, last;
void swap(char *v[], inti, 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:

```
void swap(char *v[], inti, 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.

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}
};
intday_of_year(int year, int month, int day)
{
inti, leap;
leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
for (i = 1; i< month; i++)
day += daytab[leap][i];
return day;
}
void month_day(int year, intyearday, int *pmonth, int *pday)
{

inti, 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]` rather than `daytab[i,j]`

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 `int`s. In this particular case, it is a pointer to objects that are arrays of 13 `int`s. Thus if the array `daytab` is to be passed to a function `f`, the declaration of `f` would be: `f(intdaytab[2][13]) { ... }`

It could also be

```
f(intdaytab[][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.

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:

```
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.

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 * *row* +*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 aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

Array of Pointers

We can declare an array that contains pointers as its elements. Syntax to declare array of pointer:

*datatype *arrayname[size];*

For example to declare an array of size 20 that contains integer pointers we can write

int *a[10];

where we can initialize each element of a[] with addresses of variables.

Functions returning Pointer

We can have a function that returns a pointer. Syntax to declare such type of function is

*type *function_name(type1,type2,……);*

For Example:

```
void main()
{
 int *p;
p=fun();
--------------------------
}
int *fun()
{
int a=5;
int *q=&a;
--------------------------- return
q;
}
```

Pointers to Functions

How to declare a pointer to a function?

Syntax: *returntype_of_function (*pointer variable)(List of arguments);* For example:

*int (*p)(int,int);* can be interpreted as p is a pointer to function which takes two integers as argument and returntype is integer.

How to make a pointer to a function?

Syntax:
pointer_variable=function_name_without_parantheses;
For Example:
p=test;
can be read as p is a pointer to function test.

How to call a function using pointer?

Syntax:
pointer_variable(ist of arguments);
OR
*(*pointer_variable)(List of arguments);*

The following program illustrates pointer to function

```
int getc()
{
return 10;
}
void put(int a)
{
printf("%d",a);
}
void main()
{
 int k;
int (*p)();                    /*You can write void *p();*/
void (*q)(int);                /*You can write void *q(int);*/
p=get; q=put; k=p(); q(k);
}
```

NOTE:

(i) In C every function is stored into physical memory location and entry point of that function address is stored into function name. If you assign a pointer to to the function then the base address of the function is copied into the pointer. Using this control is shifting from calling function to called function.

(ii) By default all functions are global, we can access from wherever we want. So there is no such significance to make a pointer to function.

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[], intnlines);
void writelines(char *lineptr[], intnlines);
void qsort(void *lineptr[], int left, int right, int (*comp)(void *, void *));
int numcmp(char *, char *);
/* sort input lines */
main(intargc, char *argv[])
{
intnlines; /* 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.

```
void qsort(void *v[], int left, int right,
int (*comp)(void *, void *))
{
inti, 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 *)
```

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 */
intnumcmp(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 as follows

```
void swap(void *v[], inti, int j;)
{ void *temp;
temp = v[i];
v[i] = v[j];
v[j] = temp;
}
```