

CHAPTER 5 POINTERS AND ARRAYS

Chapter 5



Before we start Chapter 5, a quick note from your narrator. From time to time I have been adding some of my interpretation to this book. But I won't be adding anything to the first part of this chapter. I think that sections 5.1 through 5.6 contain some of the most elegantly written text in the book. The concepts are clearly stated and the example code is short, direct, and easy to understand. Pointers are the essential difference between C and any other modern programming languages. So pay close attention to this chapter and make sure you understand it before continuing.

This chapter is as strong now as it was in 1978 so without further ado, we read as Kernighan and Ritchie teach us about pointers and arrays.

A pointer is a variable that contains the address of another variable. Pointers are very 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 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.

5.1 Pointers and Addresses

Since a pointer contains the address of an object, it is possible to access the object "indirectly" through the pointer. Suppose that `x` is a variable, say an `int`, and that `px` is a pointer, created in some as yet unspecified way. The unary operator `&` gives the *address* of an object, so the statement

```
px = &x;
```

assigns the address of `x` to the variable `px`; `px` is now said to "point to" `x`. The `&` operator can be applied only to variables and array elements; constructs like `&(x+1)` and `&3` are illegal. It is also illegal to take the address of a `register` variable.

The unary operator `*` treats its operand as the address of the ultimate target, and accesses that address to fetch the contents. Thus if `y` is also an `int`,

```
y = *px;
```

assigns to `y` the contents of whatever `px` points to. So the sequence

```
px = &x;  
y = *px;
```

assigns the same value to `y` as does

```
y = x;
```

It is also necessary to declare the variables that participate in all of this:

```
int x, y;  
int *px;
```

The declaration of `x` and `y` is what we've seen all along. The declaration of the pointer `px` is new.

```
int *px;
```

is intended as a mnemonic; it says that the combination `*px` is an `int`, that is, if `px` occurs in the context `*px`, it is equivalent to a variable of type `int`. In effect, the syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning is useful in all cases involving complicated declarations. For example,

```
double atof(), *dp;
```

says that in an expression `atof()` and `*dp` have values of type `double`. You should also note the implication in the declaration that a pointer is constrained to point to a particular kind of object.

Pointers can occur in expressions. For example, if `px` points to the integer `x`, then `*px` can occur in any context where `x` could.

```
y = *px + 1
```

sets `y` to 1 more than `x`;

```
printf("%d\n", *px)
```

prints the current value of `x`; and

```
d = sqrt ((double) *px)
```

produces in `d` the square root of `x`, which is coerced into a `double` before being passed to `sqrt`. (See [Chapter 2](#).)

In expressions like

```
y = *px + 1
```

the unary operators `*` and `&` bind more tightly than arithmetic operators, so this expression takes whatever `px` points at, adds 1, and assigns it to `y`. We will return shortly to what

```
y = *(px + 1)
```

might mean.

Pointer references can also occur on the left side of assignments. If `px` points to `x`, then

```
*px = 0
```

sets `x` to zero, and

```
*px += 1
```

increments it, as does

```
(*px)++
```

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

Finally, since pointers are variables, they can be manipulated as other variables can. If `py` is another pointer to `int`, then

```
py = px
```

copies the contents of `px` into `py`, thus making `py` point to whatever `px` points to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by "call by value," there is no direct way for the called function to alter a variable in the calling function. What do you do if you really have to change an ordinary argument? For example, a sorting routine might exchange two out-of-order elements with a function called `swap`. It is not enough to write

```
swap(a, b);
```

where the `swap` function is defined as

```
swap(x, y) /* WRONG */
int x, y;
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

c_091_01

Copy

Edit

Because of call by value, `swap` *can't* affect the arguments `a` and `b` in the routine that called it.

Fortunately, there is a way to obtain the desired effect. The calling program passes *pointers* to the values to be changed:

```
swap(&a, &b);
```

Since the operator `&` gives the address of a variable, `&a` is a pointer to `a`. In `swap` itself, the arguments are declared to be pointers, and the actual operands are accessed through them.

```
swap(px, py) /* interchange *px and *py */
int *px, *py;
{
    int temp;

    temp = *px;
```

c_092_01

Copy

Edit

```
*px = *py;
*py = temp;
}
```

One common use of pointer arguments is in functions that must return more than a single value. (You might say that `swap` returns two values, the new values of its arguments.) As an example, consider a function `get_int` which performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `get_int` is to return the value it found, or an end of file signal when there is no more input. These values have to be returned as separate objects, for no matter what value is used for EOF, that could also be the value of an input integer.

One solution, which is based on the input function `scanf` that we will describe in [Chapter 7](#), is to have `get_int` return EOF as its function value if it found end of file; any other returned value signals a normal integer. The numeric value of the integer it found is returned through an argument, which must be a pointer to an integer. This organization separates end of file status from numeric values.

The following loop fills an array with integers by calls to `get_int`:

```
int n, v, array[SIZE];

for (n = 0; n < SIZE && get_int(&v) != EOF; n++)
    array[n] = v;
```

Each call sets `v` to the next integer found in the input. Notice that it is essential to write `&v` instead of `v` as the argument of `get_int`. Using plain `v` is likely to cause an addressing error, since `get_int` believes it has been handed a valid pointer.

`get_int` itself is an obvious modification of the `atoi` we wrote earlier:

```
#include <stdio.h>

get_int(pn) /* get next integer from input */
int *pn;
{
    int c, sign;

    while ((c = getch()) == ' ' || c == '\n' || c == '\t')
        ; /* skip white space */
    sign = 1;
    if (c == '+' || c == '-') { /* record sign */
        sign = (c == '+') ? 1 : -1;
        c = getch();
    }
    for (*pn = 0; c >= '0' && c <= '9'; c = getch())
        *pn = 10 * *pn + c - '0';
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return(c);
}
```

c_093_01

Copy

Edit

Throughout `get_int`, `*pn` is used as an ordinary `int` variable. We have also used `getch` and `ungetch` (described in [Chapter 4](#)) so the one extra character that must be read can be pushed back onto the input.

Exercise 5-1. Write `getfloat`, the floating point analog of `get_int`. 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 really should be treated simultaneously. Any operation which 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 grasp immediately.

The declaration

```
int a[10]
```

defines an array `a` of size 10, that is a block of 10 consecutive objects named `a[0]`, `a[1]`, ..., `a[9]`. The notation `a[i]` means the element of the array `i` positions from the beginning. If `pa` is a pointer to an integer, declared as

```
int *pa
```

then the assignment

```
pa = &a[0]
```

sets `pa` to point to the zeroth element 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 `a`, then *by definition* `pa+1` points to the next element, and in general `pa+i` points `i` elements before `pa`, and `pa+i` points `i` elements after. Thus, if `pa` points to `a[0]`,

```
*(pa+1)
```

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 of the variables in the array `a`. The definition of "adding 1 to a pointer," and by extension, all pointer arithmetic, is that the increment is scaled by the size in storage of the object that is pointed to. Thus in `pa+i`, `i` is multiplied by the size of the objects that `pa` points to before being added to `pa`.

The correspondence between indexing and pointer arithmetic is evidently very close. In fact, a reference to an array is converted by the compiler to a pointer to the beginning of the array. The effect is that an array name *is* a pointer expression. This has quite a few useful implications. Since the name of an array is a synonym for the location of the zeroth element, the assignment

```
pa = &a[0]
```

can also be written as

```
pa = a
```

Rather more surprising, at least 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 completely 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 may use it with a subscript: `pa[i]` is identical to `*(pa+i)`. In short, any array and index expression can be written as a pointer and offset, and vice versa, even in the same statement.

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 sensible operations. But an array name is a *constant*, not a variable: constructions like `a=pa` or `a++` or `p=&a` are illegal.

When an array name is passed to a function, what is passed is the location of the beginning of the array. Within the called function, this argument is a variable, just like any other variable, and so an array name argument is truly a pointer, that is, a variable containing an address. We can use this fact to write a new version of `strlen`, which computes the length of a string.

```
int strlen(s) /* return length of string s */
char *s;
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return (n);
}
```

c_095_01

Copy

Edit

Incrementing `s` is perfectly legal, since it is a pointer variable; `s++` has no effect on the character string in the function that called `strlen`, but merely increments `strlen`'s private copy of the address.

As formal parameters in a function definition,

```
char s[];
```

and

```
char *s;
```

are exactly equivalent; which one should be written is determined largely by how expressions will be written in the function. 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 kinds of operations 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 element `a[2]`, because `&a[2]` and `a+2` are both pointer expressions that refer to the third element of `a`. Within `f`, the argument declaration can read

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

or

```
f(arr)
int *arr;
{
    ...
}
```

So as far as `f` is concerned, the fact that the argument really refers to part of a larger array is of no consequence.

5.4 Address Arithmetic

If `p` is a pointer, then `p++` increments `p` to point to the next element of whatever kind of object `p` points to, and `p+=i` increments `p` to point `i` elements beyond where it currently does. These and similar constructions are the simplest and most common 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 major strengths of the language. Let us illustrate some of its properties by writing a rudimentary storage allocator (but useful in spite of its simplicity). There are two routines: `alloc(n)` returns a pointer `p` to `n` consecutive character positions, which can be used by the caller of `alloc` for storing characters; `free(p)` releases the storage thus acquired so it can be later re-used. The routines are "rudimentary" because the calls to `free` must be made in the opposite order to the calls made on `alloc`. That is, the storage managed by `alloc` and `free` is a stack, or last-in, first-out list. The standard C library provides analogous functions which have no such restrictions, and in [Chapter 8](#) we will show improved versions as well. In the meantime, however, many applications really only need a trivial `alloc` to dispense little pieces of storage of unpredictable sizes at unpredictable times.

The simplest implementation is to have `alloc` hand out pieces of a large character array which we will call `allocbuf`. This array is private to `alloc` and `free`. Since they deal in pointers, not array indices, no other routine need know the name of the array, which can be declared external `static`, that is, local to the source file containing `alloc` and `free`, and invisible outside it. In practical implementations, the array may well not even have a name; it might instead be obtained 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 to the next free element, called `allocp`. 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. `free(p)` merely sets `allocp` to `p` if `p` is inside `allocbuf`.

```
#include <stdio.h>
#define NULL 0 /* pointer value for error report */
#define ALLOCSIZE 1000 /* size of available space */

static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf; /* next free position */

char *alloc(n) /* return pointer to n characters */
int n;
{
    if (allocp + n <= allocbuf + ALLOCSIZE) { /* fits */
        allocp += n;
        return(allocp - n); /* old p */
    } else /* not enough room */
        return (NULL);
}

free(p) /* free storage pointed to by p */
char *p;
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

c_097_01

Copy

Edit

Some explanations. In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are `NULL` (discussed below) or an expression involving addresses 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 `allocbuf`, which is the next free position when the program starts. This could have also been written

```
static char *allocp = &allocbuf[0];
```

since the array name *is* the address of the zeroth element; use whichever is more natural.

The test

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```

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 normal pointer (notice the declaration of the function itself). If not, `alloc` must return some signal that no space is left. C guarantees that no pointer that validly points at data will contain zero, so a return value of zero can be used to signal an abnormal event, in this case, no space. We write `NULL`, instead of zero, however, to indicate more clearly that this is a special value for a pointer. In general, integers cannot meaningfully be assigned to pointers; zero is a special case.

Tests like

```
if (allocp + n <= allocbuf + ALLOCSIZE)
```


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.

```
p > q
```

is true, for example, if `p` points to an earlier member of the array than does `q`. The relations `==` and `!=` also work. Any pointer can be meaningfully compared for equality or inequality with `NULL`. But all bets are off if you do arithmetic or comparisons with pointers pointing to different arrays. If you're lucky, you'll get obvious nonsense on all machines. If you're unlucky, your code will work on one machine but collapse mysteriously on another.

Second, we have already observed that a pointer and an integer may be added or subtracted. The construction

```
p + n
```

means the `n`-th object beyond the one `p` currently points to. This is true regardless of the kind of object `p` is declared to point at; the compiler scales `n` according to the size of the objects `p` points to, which is determined by the declaration of `p`. For example, on the PDP-11, the scale factors are 1 for `char`, 2 for `int` and `short`, 4 for `long` and `float`, and 8 for `double`.

Pointer subtraction is also valid: if `p` and `q` point to members of the same array, `p-q` is the number of elements between `p` and `q`. This fact can be used to write yet another version of `strlen`:

```
strlen(s) /* return length of string s */
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.

In the `while` loop, each character in turn is examined until the `\0` at the end is seen. Since `\0` is zero, and since `while` tests only whether the expression is zero, it is possible to omit the explicit test, and such loops are often written as

```
while (*p)
    p++;
```

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. Pointer arithmetic is consistent: if we had been dealing with `float`'s, which occupy more storage than `char`'s, and if `p` were a pointer to `float`, `p++` would advance to the next float. Thus we could write another version of `alloc` which maintains, let us say, `float`'s instead of `char`'s, merely by changing `char` to `float` throughout `alloc` and `free`. All the pointer manipulations automatically take into account the size

of the object pointed to, so nothing else has to be altered.

Other than the operations mentioned here (adding or subtracting a pointer and an integer; subtracting or comparing two pointers), all other pointer arithmetic is illegal. It is not permitted to add two pointers, or to multiply or divide or shift or mask them, or to add `float` or `double` to them.

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 compiler terminates the array with the 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; what `printf` receives is a pointer to the character array.

Character arrays of course need not be function arguments. If `message` is declared as

```
char *message;
```

then the statement

```
message = "now is the time";
```

assigns to `message` a pointer to the actual characters. 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.

We will illustrate more aspects of pointers and arrays by studying two useful functions from the standard I/O library to be discussed in [Chapter 7](#).

The first function is `strcpy(s, t)`, which copies the string `t` to the string `s`. The arguments are written in this order by analogy to assignment, where one would say

```
s = t
```

to assign `t` to `s`. The array version is first:

```
strcpy(s, t) /* copy t to s */
char s[], t[];
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

c_100_01

Copy

Edit

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

```
strcpy(s, t) /* copy t to s; pointer version 1 */
char *s, *t;
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

c_100_02

Copy

Edit

Because arguments are passed by value, `strcpy` can use `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` which terminates `t` has been copied to `s`.

In practice, `strcpy` would not be written as we showed it above. A second possibility might be

```
strcpy(s, t) /* copy t to s; pointer version 2 */
char *s, *t;
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

c_100_03

Copy

Edit

This moves the increment of `s` and `t` into the test part. 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 to and including the terminating `\0`.

As the final abbreviation, we again observe that a comparison against `\0` is redundant, so the function is often written as

```
strcpy(s, t) /* copy t to s; pointer version 3 */
char *s, *t;
{
    while (*s++ = *t++)
        ;
}
```

c_101_01

Copy

Edit

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, if for no other reason than that you will see it frequently in C programs.

The second routine is `strcmp(s, t)`, which compares the character strings `s` and `t`, and returns negative, zero or positive according as `s` is lexicographically less than, equal to, or greater than `t`. The value returned is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```
strcmp(s, t) /* return <0 if s<t, 0 if s==t, >0 if s>t */
char s[], t[];
{
    int i;

    i = 0;

    while (s[i] == t[i])
        if (s[i++] == '\0')
```

c_101_02

Copy

Edit

```

    return (0);
    return(s[i] - t[i]);
}

```

The pointer version of `strcmp`:

```

strcmp(s, t) /* return <0 if s<t, 0 if s==t, >0 if s>t */
char *s, *t;
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return (0);
    return(*s - *t);
}

```

c_102_01

Copy

Edit

Since `++` and `--` are either prefix or postfix operators, other combinations of `*` and `++` and `--` occur, although less frequently. For example,

`*++p`

increments `p` *before* fetching the character that `p` points to;

`*--p`

decrements `p` first.

Exercise 5-2. Write a pointer version of the function `strcat` which we showed in [Chapter 2](#): `strcat(s, t)` copies the string `t` to the end of `s`.

Exercise 5-3. Write a macro for `strcpy`.

Exercise 5-4. Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include `get_line` ([Chapter 1](#) and [4](#)), `atoi`, `itoa`, and their variants ([Chapter 2](#), [3](#), and [Chapter 4](#)), `reverse` ([Chapter 3](#)), and `index` and `getop` ([Chapter 4](#)).

5.6 Pointers are not Integers

You may notice in older C programs a rather cavalier attitude toward copying pointers. It has generally been true that on most machines a pointer may be assigned to an integer and back again without changing it; no scaling or conversion takes place, and no bits are lost. Regrettably, this has led to the taking of liberties with routines that return pointers which are then merely passed to other routines - the requisite pointer declarations are often left out. For example, consider the function `strsave(s)`, which copies the string `s` into a safe place, obtained by a call on `alloc`, and returns a pointer to it. Properly, this should be written as

```

#include <stdlib.h>

char *strsave(s) /* save string s somewhere */
char *s;
{
    char *p, *alloc();

    if ((p = alloc(strlen(s)+1)) != NULL)
        strcpy(p, s);
    return(p);
}

```

c_103_01

Copy

Edit

```
}

```

In practice, there would be a strong tendency to omit declarations:

```
#include <stdlib.h>

strsave(s) /* save string s somewhere */
{
    char *p;

    if ((p = alloc(strlen(s)+1)) != NULL)
        strcpy(p, s);
    return(p);
}
```

c_103_02

Copy

Edit

This will work on many machines, since the default type for functions and arguments is `int`, and `int` and pointer can usually be safely assigned back and forth. Nevertheless this kind of code is inherently risky, for it depends on details of implementation and machine architecture which may not hold for the particular compiler you use. It's wiser to be complete in all declarations. (The program *lint* will warn of such constructions, in case they creep in inadvertently.)

Making sure that programmers did not blithely convert from integers to pointers was an important part of what would become ANSI C. The introduction of the `void` type as the lack of a type also allowed for a pointer to a void types object (`void *`). When a function like `alloc` wants to return a generic pointer that can be cast to become a `char *`, the return type of the function is `void *` in modern C dialects.

In general C assumes that all pointers are the same size in terms of storage allocation and can be converted back and forth. This means if you get an address of an integer, you can re-cast it to be an address of a character. An address is an address.

In the pre-1977 days of C, the duality between integers and pointers worked well enough because the amount of memory in early computers was relatively small. And addresses were generally small positive numbers that grew from zero up to possibly the maximum amount of memory in a machine. So if you were on a tiny system with 16-bit addresses and 32-bit integers, if you temporarily copied a positive 16-bit address value into a 32-bit integer variable and then back into a 16-bit address value - no accuracy would be lost in the conversion.

If we ended up with computers with more than 2GB of memory and integers were 32-bits you could find a situation where a 64-bit address could not be copied through a 32-bit integer without the loss of data. So C needed to quit returning pointers as integers sometime before 2005. Most C compilers supported the `void` type by 1979 so it was fixed with plenty of time to spare.

In this 1978 book, they are explicitly moving away from the previous use of integers to store pointers and instead having generic routines like `alloc()` return a `char *` pointer. This was then converted to its ultimate type such as `int *`. This way at least the conversion is from one address to another. By 1979, most of the C compilers supported the `void *` pattern and it became the modern way to return a generic address from a function like `alloc()`

5.7 Multi-Dimensional Arrays

In general, rectangular multi-dimensional arrays are used in computational programs like a weather simulation and were a way (back in the day) to write C code that could accept FORTRAN multi-dimensional arrays as parameters so that computational or statistical libraries could be written in C.

Arrays of pointers are a better mapping to the typical operating system and string manipulation use cases that is more the core of C applications. We also call these "ragged arrays" because each row can be a different length. This also works well as data is dynamically allocated in C - as compared to the more typical static allocation in FORTRAN's multi-dimensional arrays.

C provides for rectangular multi-dimensional arrays, although in practice they tend to be 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 returns two values, the month and day arguments will be pointers:

```
month_day(1977, 60, &m, &d)
```

sets `m` to 3 and `d` to 1 (March 1st).

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 try to keep track of what happens to February during computation. The array and the functions for performing the transformations are as follows:

```
static int day_tab[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(year, month, day) /* set day of year */
int year, month, day;      /* from month & day */
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += day_tab[leap][i];
    return (day);
}

month_day(year, yearday, pmonth, pday) /* set month, day */
int year, yearday, *pmonth, *pday; /* from day of year */
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > day_tab[leap][i]; i++)
        yearday -= day_tab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

c_104_01

Copy

Edit

The array `day_tab` has to be external to both `day_of_year` and `month_day`, so they can both use it.

`day_tab` is the first two-dimensional array we have dealt with. In C, by definition a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

```
day_tab[i][j]
```

rather than

```
day_tab[i, j]
```

as in most languages. Other than this, a two-dimensional array can be treated in much the same way as in other languages. Elements are stored by rows, that is, the rightmost subscript 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 `day_tab` 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 easier than adjusting indices.

If a two-dimensional array is to be passed to a function, the argument declaration in the function *must* include the column dimension; the row dimension is irrelevant, since what is passed is, as before, a pointer. In this particular case, it is a pointer to objects which are arrays of 13 `int`'s. Thus if the array `day_tab` is to be passed to a function `f`, the declaration of `f` would be

```
f(day_tab)
int day_tab[2][13];
{
    ...
}
```

The argument declaration in `f` could also be

```
int day_tab[][13];
```

since the number of rows is irrelevant, or it could be

```
int (*day_tab)[13];
```

which says that the argument is a pointer to an array of 13 integers. The parentheses are necessary since brackets `[]` have higher precedence than `*`; without parentheses, the declaration

```
int *day_tab[13];
```

is an array of 13 pointers to integers, as we shall see in the next section.

5.8 Pointer Arrays; Pointers to Pointers

Since pointers are variables themselves, you might expect that there would be uses for arrays of pointers. This is indeed the case. 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 utility *sort*.

In [Chapter 3](#) we presented a Shell sort function that would sort an array of integers. The same algorithm 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 (maintained by `alloc`, perhaps), 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 actual lines.

The sorting process involves 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 things.

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 line 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.

```
/* This file combines three successive sample code segments
   for ease of viewing, editing, and executing */
```

c_106_01

Copy

Edit

```
#include <stdio.h>
#include <string.h>
#define LINES 100 /* max lines to be sorted */

main() /* sort input lines */
{
    char *lineptr[LINES]; /* pointers to text lines */
    int nlines; /* number of input lines read */

    if ((nlines = readlines(lineptr, LINES)) >= 0) {
        sort(lineptr, nlines);
        writelines(lineptr, nlines);
    }
    else
        printf("input too big to sort\n");
}

/* The first example code from page 107 of the text book */

#define MAXLEN 1000

readlines(lineptr, maxlines) /* read input lines */
char *lineptr[]; /* for sorting */
int maxlines;
{
    int len, nlines;
    char *p, *alloc(), line[MAXLEN];

    nlines = 0;
    while ((len = get_line(line, MAXLEN)) > 0)
        if (nlines >= maxlines)
            return(-1);
        else if ((p = alloc(len)) == NULL)
            return(-1);
        else {
            line[len-1] = '\0'; /* zap newline */
```

```

        strcpy(p, line);
        lineptr[nlines++] = p;
    }
    return (nlines);
}

/* The newline at the end of each line is deleted so it
   will not affect the order in which the lines are sorted. */

/* The second example code from page 107 of the text book */

writelines(lineptr, nlines) /* write output lines */
char *lineptr[];
int nlines;
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}

```

The main new thing is the declaration for lineptr:

```
char *lineptr[LINES];
```

says that lineptr is an array of LINES elements, each element of which is a pointer to a char. That is, lineptr[i] is a character pointer, and *lineptr[i] accesses a character.

Since lineptr is itself an array which is passed to writelines, it can be treated as a pointer in exactly the same manner as our earlier examples, and the function can be written instead as

```

writelines(lineptr, nlines) /* write output lines */
char *lineptr[];
int nlines;
{
    while (--nlines >= 0)
        printf("%s\n", *lineptr++);
}

```

c_108_01

Copy

Edit

*lineptr points initially to the first line; each increment advances it to the next line while nlines is counted down.

With input and output under control, we can proceed to sorting. The Shell sort from [Chapter 3](#) needs minor changes: the declarations have to be modified, and the comparison operation must be moved into a separate function. The basic algorithm remains the same, which gives us some confidence that it will still work.

```

sort(v, n) /* sort strings v[0] v[n-1] */
char *v[]; /* into increasing order */
int n;
{
    int gap, i, j;
    char *temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i-gap; j >= 0; j -= gap) {
                if (strcmp(v[j], v[j+gap]) <= 0)
                    break;
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}

```

c_108_02

Copy

Edit

```
}

```

Since any individual element of `v` (alias `lineptr`) is a character pointer, `temp` also should be, so one can be copied to the other.

We wrote the program about as straightforwardly as possible, so as to get it working quickly. It might be faster, for instance, to copy the incoming lines directly into an array maintained by `readlines`, rather than copying them into `line` and then to a hidden place maintained by `alloc`. But it's wiser to make the first draft something easy to understand, and worry about "efficiency" later. The way to make this program significantly faster is probably not by avoiding an unnecessary copy of the input lines. Replacing the Shell sort by something better, like Quicksort, is more likely to make a difference.

In [Chapter 1](#) we pointed out that because `while` and `for` loops test the termination condition *before* executing the loop body even once, they help to ensure that programs will work at their boundaries, in particular with no input. It is illuminating to walk through the functions of the sorting program, checking what happens if there is no input text at all.

Exercise 5-5. Rewrite `readlines` to create lines in an array supplied by `main`, rather than calling `alloc` to maintain storage. How much faster is the program?

5.9 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. The topic of this section is how that array of names is initialized.

The syntax is quite similar to previous initializations:

```
char *month_name(n) /* return name of n-th month */
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]);
}
```

c_109_01

Copy

Edit

The declaration of `name`, which is an array of character pointers, is the same as `lineptr` in the sorting example. The initializer is simply a list of character strings; each is assigned to the

corresponding position in the array. More precisely, the characters of the i -th string are placed somewhere else, and a pointer to them is stored in `name[i]`. Since the size of the array `name` is not specified, the compiler itself counts the initializers and fills in the correct number.

5.10 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 declarations

```
int a[10][10];
int *b[10];
```

the usage of `a` and `b` may be similar, in that `a[5][5]` and `b[5][5]` are both legal references to a single `int`. But `a` is a true array: all 100 storage cells have been allocated, and the conventional rectangular subscript calculation is done to find any given element. For `b`, however, the declaration only allocates 10 pointers; each must be set to point to an array of integers. Assuming that each does point to a ten-element array, then there will be 100 storage cells set aside, plus the ten cells for the pointers. Thus the array of pointers uses slightly more space, and may require an explicit initialization step. But it has two advantages: accessing an element is done by indirection through a pointer rather than by a multiplication and an addition, and the rows of the array may be of different lengths. That is, each element of `b` need not point to a ten-element vector; some may point to two elements, some to twenty, 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 like that shown in `month_name`: to store character strings of diverse lengths.

Exercise 5-6. Rewrite the routines `day_of_year` and `month_day` with pointers instead of indexing.

5.11 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 to begin execution, it is called with two arguments. The first (conventionally called `argc`) is the number of command-line arguments the program was invoked with; the second (`argv`) is a pointer to an array of character strings that contain the arguments, one per string. Manipulating these character strings is a common use of multiple levels of pointers.

Back in 1978, the two largest bodies of C code were likely the AT&T UNIX kernel and UNIX utilities like `grep`, `ls`, or the login shell. So writing an operating system was fresh on the mind of the authors while writing this book. These topics find their way into the text of the book.

In a sense, a likely second-order goal of the book was to train programmers that might learn C and then help build and maintain UNIX. The 1978 edition of this textbook fits nicely into a series of AT&T Bell Labs technical reports like "Portability of C Programs and the UNIX System" written by Steven C. Johnson and Dennis M. Ritchie, published in The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978, pp 2021-2048.

Portability of C Programs and the UNIX System

The simplest illustration of the necessary declarations and use is the program `echo`, which simply echoes its command-line arguments on a single line, separated by blanks. That is, if the command

```
echo hello, world
```

is given, the output is

```
hello, world
```

By convention, `argv[0]` is the name by which the program was invoked, so `argc` is at least 1. In the example above, `argc` is 3, and `argv[0]`, `argv[1]` and `argv[2]` are "echo", "hello,", and "world" respectively. The first real argument is `argv[1]` and the last is `argv[argc-1]`. If `argc` is 1, there are no command-line arguments after the program name. This is shown in `echo`:

```
#include <stdio.h>
#include <string.h>
main(argc, argv) /* echo arguments; 1st version */
int argc;
char *argv[];
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

c_111_01

Copy

Edit

Since `argv` is a pointer to an array of pointers, there are several ways to write this program that involve manipulating the pointer rather than indexing an array. Let us show two variations.

```
#include <stdio.h>
#include <string.h>
main(argc, argv) /* echo arguments; 2nd version */
int argc;
char *argv[];
{
    while (--argc > 0)
        printf("%s%c", *++argv, (argc > 1) ? ' ' : '\n');
}
```

c_111_02

Copy

Edit

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,

```
#include <stdio.h>
#include <string.h>
main(argc, argv) /* echo arguments; 3rd version */
int argc;
char *argv[];
{
    while (--argc > 0)
        printf((argc > 1) ? "%s " : "%s\n", *++argv);
}
```

c_111_03

Copy

Edit

This version shows that the format argument of `printf` can be an expression just like any of the others. This usage is not very frequent, but worth remembering.

As a second example, let us make some enhancements to the pattern-finding program from [Chapter 4](#). If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX utility *grep*, let us change 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

main(argc, argv) /* find pattern from first argument */
int argc;
char *argv[];
{
    char line[MAXLINE];

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (index(line, argv[1]) >= 0)
                printf("%s", line);
}
```

c_112_01

Copy

Edit

The basic model can now be elaborated to illustrate further pointer constructions. Suppose we want to allow two optional arguments. One says "print all lines *except* those that match the pattern;" the second says "precede each printed line with its line number."

A common convention for C programs is that an argument beginning 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 the
```

with the input

```
now is the time
for all good men
to come to the aid
of their party.
```

should produce the output

```
2: for all good men
```

Optional arguments should be permitted in any order, and the rest of the program should be insensitive to the number of arguments which were actually present. In particular, the call to `index` should not refer to `argv[2]` when there was a single flag argument and to `argv[1]` when there wasn't. Furthermore, it is convenient for users if option arguments can be concatenated, as in

```
find -nx the
```

Here is the program.

```

#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

main(argc, argv) /* find pattern from first argument */
int argc;
char *argv[];
{
    char line[MAXLINE], *s;
    long lineno = 0;
    int except = 0, number = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
            switch (*s) {
                case 'x':
                    except = 1;
                    break;
                case 'n':
                    number = 1;
                    break;
                default:
                    printf("find: illegal option %c\n", *s);
                    argc = 0;
                    break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((index(line, *argv) >= 0) != except) {
                if (number)
                    printf("%ld: ", lineno);
                printf("%s", line);
            }
        }
}

```

c_113_01

Copy

Edit

`argv` is incremented before each optional argument, and `argc` decremented. If there are no errors, at the end of the loop `argc` should be 1 and `*argv` should point at the pattern. Notice that `*++argv` is a pointer to an argument string; `(*++argv)[0]` is its first character. The parentheses are necessary, for without them the expression would be `*++(argv[0])`, which is quite different (and wrong). An alternate valid form would be `**++argv`.

Exercise 5-7. Write the program `add` which evaluates a reverse Polish expression from the command line. For example,

```
add 2 3 4 + *
```

evaluates $2 * (3+4)$.

Exercise 5-8. Modify the programs `entab` and `detab` (written as exercises in [Chapter 1](#)) to accept a list of tab stops as arguments. Use the normal tab settings if there are no arguments.

Exercise 5-9. Extend `entab` and `detab` to accept the shorthand

```
entab m +n
```

to mean tabs stops every n columns, starting at column m . Choose convenient (for the user) default behavior.

Exercise 5-10. Write the program `tail`, which prints the last n lines of its input. By default, n is 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 `sort`, not in a two-dimensional array of fixed size.

5.12 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define a *pointer to a function*, which can be manipulated, passed to functions, placed in arrays, 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* which determines the ordering of any pair of objects, an *exchange* which reverses their order, and a *sorting algorithm* which 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`.

The lexicographic comparison of two lines is done by `strcmp` and swapping by `swap` as before; we will also need a routine `numcmp` which compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does. These three functions are declared in `main` and pointers to them are passed to `sort`. `sort` in turn calls the functions via the pointers. We have skimmed on error processing for arguments, so as to concentrate on the main issues.

```
#include <stdio.h>
#include <string.h>
#define LINES 100 /* max number of lines to be sorted */

main(argc, argv) /* sort input lines */
int argc;
char *argv[];
{
    char *lineptr[LINES]; /* pointers to text lines */
    int nlines; /* number of input lines read */
    int strcmp(), numcmp(); /* comparison functions */
    int swap(); /* exchange function */
    int numeric = 0; /* 1 if numeric sort */

    if (argc>1 && argv[1][0] == '-' && argv[1][1] == 'n')
        numeric = 1;
    if ((nlines = readlines(lineptr, LINES)) >= 0) {
        if (numeric)
            sort(lineptr, nlines, numcmp, swap);
        else
            sort(lineptr, nlines, strcmp, swap);
        writelines(lineptr, nlines);
    } else
        printf("input too big to sort\n");
}
```

c_115_01

Copy

Edit

`strcmp`, `numcmp` and `swap` are addresses of functions; since they are known to be functions, the `&` operator is not necessary, in the same way that it is not needed before an array name. The compiler arranges for the address of the function to be passed.

The second step is to modify `sort`:

```
sort(v, n, comp, exch) /* sort strings v[0]...v[n-1] */
char *v[]; /* into increasing order */
int n;
int (*comp)(), (*exch)();
{
    int gap, i, j;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i-gap; j >= 0; j -= gap) {
                if ((*comp)(v[j], v[j+gap]) <= 0)
                    break;
                (*exch>(&v[j], &v[j+gap]));
            }
}
```

c_116_01

Copy

Edit

The declarations should be studied with some care.

```
int (*comp) ()
```

says that `comp` is a pointer to a function that returns an `int`. The first set of parentheses are necessary; without them,

```
int *comp()
```

would say that `comp` is a function returning a pointer to an `int`, which is quite a different thing.

The use of `comp` in the line

```
if ((*comp)(v[j], v[j+gap]) <= 0)
```

is consistent with the declaration: `comp` is a pointer to a function, `*comp` is the function, and

```
(*comp)(v[j], v[j+gap])
```

is the call to it. The parentheses are needed so the components are correctly associated.

We have already shown `strcmp`, which compares two strings. Here is `numcmp`, which compares two strings on a leading numeric value:

```
numcmp(s1, s2) /* compare s1 and s2 numerically */
char *s1, *s2;
{
    double atof(), v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return(-1);
    else if (v1 > v2)
        return(1);
    else
        return(0);
}
```

c_117_01

Copy

Edit

The final step is to add the function `swap` which exchanges two pointers. This is adapted directly from what we presented early in the chapter.

```
swap(px, py) /* interchange *px and *py */
char *px[], *py[];
{
    char *temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

c_117_02

Copy

Edit

There are a variety of other options that can be added to the sorting program; some make challenging exercises.

Exercise 5-11. Modify `sort` to handle a `-r` flag, which indicates sorting in reverse (decreasing) order. Of course `-r` must work with `-n`.

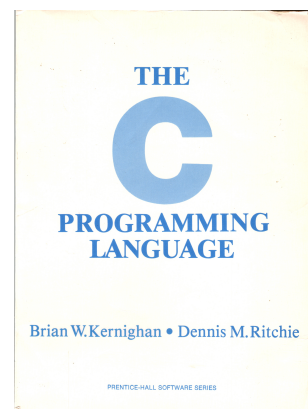
Exercise 5-12. Add the option `-f` to fold upper and lower case together, so that case distinctions are not made during sorting: upper and lower case data are sorted together, so that `a` and `A` appear adjacent, not separated by an entire case of the alphabet.

Exercise 5-13. Add the `-d` ("dictionary order") option, which makes comparisons only on letters, numbers and blanks. Make sure it works in conjunction with `-f`.

Exercise 5-14. Add a field-handling capability, so sorting may be done on fields within lines, each field 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.)

This work (www.cc4e.com) is based on the 1978 ["The C Programming Language"](#) book written by Brian W. Kernighan and Dennis M. Ritchie. Their book is copyright all rights reserved by AT&T but is being used in this work under "fair use" because of the book's historical and scholarly significance, its lack of availability, and lack of an accessible version of the book.

The book is augmented in places to help understand its rightful place in a historical context amidst the major changes of the 1970's and 1980's as Computer Science evolved from a hardware-first / vendor-centered approach to a software-centered approach where portable operating systems and applications written in C could run on any hardware.



This is not the ideal book to learn C programming because the 1978 edition does not reflect the modern C language. Using an obsolete book gives us an opportunity to take students back in time and understand how the C language was evolving as it laid the ground work for a future with portable applications.

If you want to learn modern C programming from a book that reflects the modern C language, I suggest you use the [C Programming Language, 2nd Edition](#), also written by Brian W. Kernighan and Dennis M. Ritchie, initially published in 1988 and based on the [ANSI C](#) (C89) version of the language.

The source code to this book is at <https://github.com/csev/cc4e>. You are welcome to help in the

creation and editing of the book.