and data cannot conflict even inadvertently. For example, `getch` and `ungetch` form a "module" for character input and pushback; `buf` and `bufp` should be `static` so they are inaccessible from the outside. In the same way, `push`, `pop` and `clear` form a module for stack manipulation; `val` and `sp` should also be external `static`.

4.7 Register Variables

The fourth and final storage class is called `register`. A `register` declaration advises the compiler that the variable in question will be heavily used. When possible, `register` variables are placed in machine registers, which may result in smaller and faster programs.

The `register` declaration looks like

```
register int   x;
register char  c;
```

and so on; the `int` part may be omitted. `register` can only be applied to automatic variables and to the formal parameters of a function. In this latter case, the declaration looks like

```
f(c, n)
register int c, n;
{
     register int i;
     ...
}
```

In practice, there are some restrictions on register variables, reflecting the realities of underlying hardware. Only a few variables in each function may be kept in registers, and only certain types are allowed. The word `register` is ignored for excess or disallowed declarations. And it is not possible to take the address of a register variable (a topic to be covered in Chapter 5). The specific restrictions vary from machine to machine; as an example, on the PDP-11, only the first three register declarations in a function are effective, and the types must be `int`, `char`, or pointer.

4.8 Block Structure

C is not a block-structured language in the sense of PL/I or Algol, in that functions may not be defined within other functions.

On the other hand, variables can be defined in a block-structured fashion. Declarations of variables (including initializations) may follow the left brace that introduces *any* compound statement, not just the one that begins a function. Variables declared in this way supersede any identically named variables in outer blocks, and remain in existence until the matching right brace. For example, in

```
if (n > 0) {
        int i;      /* declare a new i */
        for (i = 0; i < n; i++)
                ...
}
```

the scope of the variable `i` is the "true" branch of the `if`; this `i` is unrelated to any other `i` in the program.

Block structure also applies to external variables. Given the declarations

```
int x;

f()
{
        double x;
        ...
}
```

then within the function `f`, occurrences of `x` refer to the internal `double` variable; outside of `f`, they refer to the external integer. The same is true of the names of formal parameters:

```
int z;

f(z)
double z;
{
        ...
}
```

Within the function `f`, `z` refers to the formal parameter, not the external.

4.9 Initialization

Initialization has been mentioned in passing many times so far, but always peripherally to some other topic. This section summarizes some of the rules, now that we have discussed the various storage classes.

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic and register variables have undefined (i.e., garbage) values.

Simple variables (not arrays or structures) may be initialized when they are declared, by following the name with an equals sign and a constant expression:

```
int  x = 1;
char squote = '\'';
long day = 60 * 24; /* minutes in a day */
```

For external and static variables, the initialization is done once, conceptually at compile time. For automatic and register variables, it is done each time the function or block is entered.

For automatic and register variables, the initializer is not restricted to being a constant: it may in fact be any valid expression involving previously defined values, even function calls. For example, the initializations of the binary search program in Chapter 3 could be written as

```
binary(x, v, n)
int x, v[], n;
{
        int low = 0;
        int high = n - 1;
        int mid;
        ...
}
```

instead of

```
binary(x, v, n)
int x, v[], n;
{
        int low, high, mid;

        low = 0;
        high = n - 1;
        ...
}
```

In effect, initializations of automatic variables are just shorthand for assignment statements. Which form to prefer is largely a matter of taste. We have generally used explicit assignments, because initializers in declarations are harder to see.

Automatic arrays may not be initialized. External and static arrays may be initialized by following the declaration with a list of initializers enclosed in braces and separated by commas. For example, the character counting program of Chapter 1, which began

```
main()    /* count digits, white space, others */
{
        int c, i, nwhite, nother;
        int ndigit[10];

        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
                ndigit[i] = 0;
        ...
}
```

can be written instead as

```
int  nwhite = 0;
int  nother = 0;
int  ndigit[10] ={ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

main()    /* count digits, white space, others */
{
     int c, i;
     ...
}
```

These initializations are actually unnecessary since all are zero, but it's good form to make them explicit anyway. If there are fewer initializers than the specified size, the others will be zero. It is an error to have too many initializers. Regrettably, there is no way to specify repetition of an initializer, nor to initialize an element in the middle of an array without supplying all the intervening values as well.

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

```
char pattern[] = "the";
```

This is a shorthand for the longer but equivalent

```
char pattern[] = { 't', 'h', 'e', '\0' };
```

When the size of an array of any type is omitted, the compiler will compute the length by counting the initializers. In this specific case, the size is 4 (three characters plus the terminating `\0`).

4.10 Recursion

C functions may be used recursively; that is, a function may call *itself* either directly or indirectly. One traditional example involves printing a number as a character string. As we mentioned before, the digits are generated in the wrong order: low-order digits are available before high-order digits, but they have to be printed the other way around.

There are two solutions to this problem. One is to store the digits in an array as they are generated, then print them in the reverse order, as we did in Chapter 3 with `itoa`. The first version of `printd` follows this pattern.

```
printd(n) /* print n in decimal */
int n;
{
    char s[10];
    int i;

    if (n < 0) {
        putchar('-');
        n = -n;
    }
    i = 0;
    do {
        s[i++] = n % 10 + '0';    /* get next char */
    } while ((n /= 10) > 0); /* discard it */
    while (--i >= 0)
        putchar(s[i]);
}
```

The alternative is a recursive solution, in which each call of `printd` first calls itself to cope with any leading digits, then prints the trailing digit.

```
printd(n) /* print n in decimal (recursive) */
int n;
{
    int i;

    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if ((i = n/10) != 0)
        printd(i);
    putchar(n % 10 + '0');
}
```

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, quite independent of the previous set. Thus in `printd(123)` the first `printd` has `n = 123`. It passes 12 to a second `printd`, then prints 3 when that one returns. In the same way, the second `printd` passes 1 to a third (which prints it), then prints 2.

Recursion generally provides no saving in storage, since somewhere a stack of the values being processed has to be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand. Recursion is especially convenient for recursively defined data structures like trees; we will see a nice example in Chapter 6.

Exercise 4-7. Adapt the ideas of `printd` to write a recursive version of `itoa`; that is, convert an integer into a string with a recursive routine. □

Exercise 4-8. Write a recursive version of the function `reverse(s)`, which reverses the string `s`. □

4.11 The C Preprocessor

C provides certain language extensions by means of a simple macro preprocessor. The `#define` capability which we have used is the most common of these extensions; another is the ability to include the contents of other files during compilation.

File Inclusion

To facilitate handling collections of `#define`'s and declarations (among other things) C provides a file inclusion feature. Any line that looks like

```
#include "filename"
```

is replaced by the contents of the file *filename*. (The quotes are mandatory.) Often a line or two of this form appears at the beginning of each source file, to include common `#define` statements and `extern` declarations for global variables. `#include`'s may be nested.

`#include` is the preferred way to tie the declarations together for a large program. It guarantees that all the source files will be supplied with the same definitions and variable declarations, and thus eliminates a particularly nasty kind of bug. Of course, when an included file is changed, all files that depend on it must be recompiled.

Macro Substitution

A definition of the form

```
#define   YES  1
```

calls for a macro substitution of the simplest kind — replacing a name by a string of characters. Names in `#define` have the same form as C identifiers; the replacement text is arbitrary. Normally the replacement text is the rest of the line; a long definition may be continued by placing a \ at the end of the line to be continued. The "scope" of a name defined with `#define` is from its point of definition to the end of the source file. Names may be redefined, and a definition may use previous definitions. Substitutions do not take place within quoted strings, so, for example, if `YES` is a defined name, there would be no substitution in `printf("YES")`.

Since implementation of `#define` is a macro prepass, not part of the compiler proper, there are very few grammatical restrictions on what can be defined. For example, Algol fans can say

```
    #define then
    #define begin  {
    #define end    ;}
```

and then write

```
    if (i > 0) then
         begin
              a = 1;
              b = 2
         end
```

It is also possible to define macros with arguments, so the replacement text depends on the way the macro is called. As an example, define a macro called **max** like this:

```
    #define  max(A, B)  ((A) > (B) ? (A) : (B))
```

Now the line

```
    x = max(p+q, r+s);
```

will be replaced by the line

```
    x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

This provides a "maximum function" that expands into in-line code rather than a function call. So long as the arguments are treated consistently, this macro will serve for any data type; there is no need for different kinds of **max** for different data types, as there would be with functions.

Of course, if you examine the expansion of **max** above, you will notice some pitfalls. The expressions are evaluated twice; this is bad if they involve side effects like function calls and increment operators. Some care has to be taken with parentheses to make sure the order of evaluation is preserved. (Consider the macro

```
    #define  square(x)  x * x
```

when invoked as `square(z+1)`.) There are even some purely lexical problems: there can be no space between the macro name and the left parenthesis that introduces its argument list.

Nonetheless, macros are quite valuable. One practical example is the standard I/O library to be described in Chapter 7, in which `getchar` and `putchar` are defined as macros (obviously `putchar` needs an argument), thus avoiding the overhead of a function call per character processed.

Other capabilities of the macro processor are described in Appendix A.

Exercise 4-9. Define a macro `swap(x, y)` which interchanges its two `int` arguments. (Block structure will help.) □

CHAPTER 5: 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;
```

note to help memorize 技巧

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;
        }
```

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;
        *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 `getint` which performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found, 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 `getint` 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 `getint`:

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

for (n = 0; n < SIZE && getint(&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 `getint`. Using plain `v` is likely to cause an addressing error, since `getint` believes it has been handed a valid pointer.

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

```
getint(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);
}
```

Throughout `getint`, `*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 `getint`. What type does `getfloat` return as its function value? □

5.3 Pointers and Arrays

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

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

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

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.

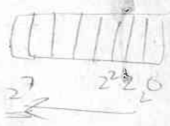
The other information needed is how much of `allocbuf` has been used. We use a pointer to the next free element, called `allocp`. When `alloc` is asked for `n` characters, it checks to see if there is enough room left in `allocbuf`. If so, `alloc` returns the current value of `allocp` (i.e., the beginning 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`.

```
#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;
}
```



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

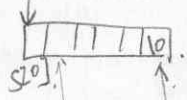
```
    p + n
```

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++;
}
```

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++;
	}
}
```

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')
		;
}
```

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++)
          ;
}
```

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')
               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);
}
```

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 `getline` (Chapters 1 and 4), `atoi`, `itoa`, and their variants (Chapters 2, 3, and 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