```
#include <stdio.h>

/* print Fahrenheit-Celsius table
    for fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* lower limit of temperature table */
    upper = 300;    /* upper limit */
    step = 20;      /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

The two lines

```
/* print Fahrenheit-Celsius table
    for fahr = 0, 20, ..., 300 */
```

are a *comment*, which in this case explains briefly what the program does. Any characters between `/*` and `*/` are ignored by the compiler; they may be used freely to make a program easier to understand. Comments may appear anywhere a blank or tab or newline can.

In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements. A *declaration* announces the properties of variables; it consists of a type name and a list of variables, such as

```
int fahr, celsius;
int lower, upper, step;
```

The type `int` means that the variables listed are integers, by contrast with `float`, which means floating point, i.e., numbers that may have a fractional part. The range of both `int` and `float` depends on the machine you are using; 16-bit `int`s, which lie between −32768 and +32767, are common, as are 32-bit `int`s. A `float` number is typically a 32-bit quantity, with at least six significant digits and magnitude generally between about 10^{-38} and 10^{+38}.

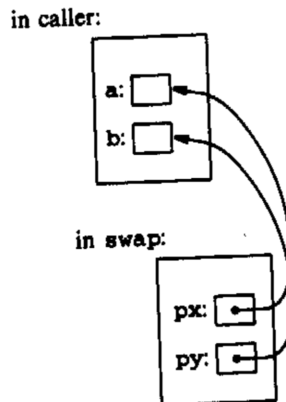
C provides several other basic data types besides `int` and `float`, including:

`char`	character—a single byte
`short`	short integer
`long`	long integer
`double`	double-precision floating point

```
#include <stdio.h>

#define MAXLINE 1000     /* maximum input line size */

int max;                 /* maximum length seen so far */
char line[MAXLINE];      /* current input line */
char longest[MAXLINE];   /* longest line saved here */

int getline(void);
void copy(void);

/* print longest input line; specialized version */
main()
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0)    /* there was a line */
        printf("%s", longest);
    return 0;
}

/* getline:  specialized version */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE-1
        && (c=getchar()) != EOF && c != '\n'; ++i)
            line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}
```

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

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

Pictorially:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for `EOF`, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf` as well; see Section 7.4.

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

```
int n, array[SIZE], getint(int *);

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

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint` returns `EOF` for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
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].

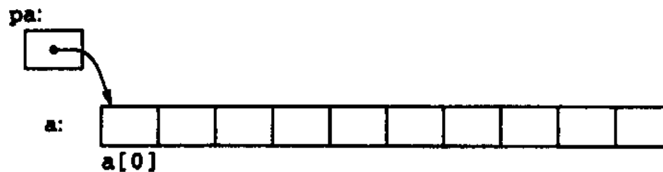
The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

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

sets pa to point to element zero of a; that is, pa contains the address of a[0].



Now the assignment

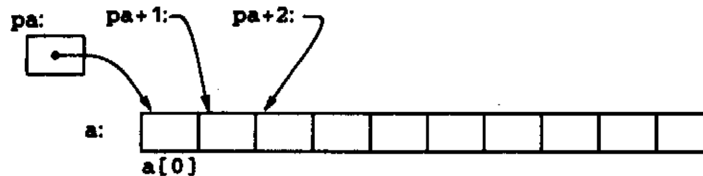
```
x = *pa;
```

will copy the contents of a[0] into x.

If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before. 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 or size of the variables in the array a. The meaning of "adding 1 to a pointer," and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th

characters, we need a loop. The array version is first:

```
/* strcpy:  copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

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

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

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

Because arguments are passed by value, `strcpy` can use the parameters `s` and `t` in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the `'\0'` that terminates `t` has been copied to `s`.

In practice, `strcpy` would not be written as we showed it above. Experienced C programmers would prefer

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

This moves the increment of `s` and `t` into the test part of the loop. The value of `*t++` is the character that `t` pointed to before `t` was incremented; the postfix `++` doesn't change `t` until after this character has been fetched. In the same way, the character is stored into the old `s` position before `s` is incremented. This character is also the value that is compared against `'\0'` to control the loop. The net effect is that characters are copied from `t` to `s`, up to and including the terminating `'\0'`.

As the final abbreviation, observe that a comparison against `'\0'` is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

- An `h` if the integer is to be printed as a `short`, or `l` (letter ell) if as a `long`.

Conversion characters are shown in Table 7-1. If the character after the `%` is not a conversion specification, the behavior is undefined.

TABLE 7-1. BASIC PRINTF CONVERSIONS

CHARACTER	ARGUMENT TYPE; PRINTED AS
`d`, `i`	`int`; decimal number.
`o`	`int`; unsigned octal number (without a leading zero).
`x`, `X`	`int`; unsigned hexadecimal number (without a leading `0x` or `0X`), using `abcdef` or `ABCDEF` for 10, ..., 15.
`u`	`int`; unsigned decimal number.
`c`	`int`; single character.
`s`	`char *`; print characters from the string until a `'\0'` or the number of characters given by the precision.
`f`	`double`; [-]*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
`e`, `E`	`double`; [-]*m.dddddd*`e`±*xx* or [-]*m.dddddd*`E`±*xx*, where the number of *d*'s is given by the precision (default 6).
`g`, `G`	`double`; use `%e` or `%E` if the exponent is less than -4 or greater than or equal to the precision; otherwise use `%f`. Trailing zeros and a trailing decimal point are not printed.
`p`	`void *`; pointer (implementation-dependent representation).
`%`	no argument is converted; print a `%`.

A width or precision may be specified as `*`, in which case the value is computed by converting the next argument (which must be an `int`). For example, to print at most `max` characters from a string `s`,

```
printf("%.*s", max, s);
```

Most of the format conversions have been illustrated in earlier chapters. One exception is precision as it relates to strings. The following table shows the effect of a variety of specifications in printing "hello, world" (12 characters). We have put colons around each field so you can see its extent.

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

A warning: `printf` uses its first argument to decide how many arguments

semantics of C. Conversion characters are shown in Table 7-2.

TABLE 7-2. BASIC SCANF CONVERSIONS

CHARACTER	INPUT DATA; ARGUMENT TYPE
d	decimal integer; int *.
i	integer; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); int *.
u	unsigned decimal integer; unsigned int *.
x	hexadecimal integer (with or without leading 0x or 0X); int *.
c	characters; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip over white space is suppressed; to read the next non-white space character, use %1s.
s	character string (not quoted); char *, pointing to an array of characters large enough for the string and a terminating '\0' that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; float *.
%	literal %; no assignment is made.

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list. Similarly, the conversion characters e, f, and g may be preceded by l to indicate that a pointer to double rather than float is in the argument list.

As a first example, the rudimentary calculator of Chapter 4 can be written with scanf to do the input conversion:

```
#include    <stdio.h>

main()  /* rudimentary calculator */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Suppose we want to read input lines that contain dates of the form

```
25 Dec 1988
```

The scanf statement is

```
    int day, year;
    char monthname[20];

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

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

Literal characters can appear in the `scanf` format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with this `scanf` statement:

```
    int day, month, year;

    scanf("%d/%d/%d", &month, &day, &year);
```

`scanf` ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with `sscanf`. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line);   /* 25 Dec 1988 form */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line);   /* mm/dd/yy form */
    else
        printf("invalid: %s\n", line); /* invalid form */
}
```

Calls to `scanf` can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by `scanf`.

A final warning: the arguments to `scanf` and `sscanf` *must* be pointers. By far the most common error is writing

```
    scanf("%d", n);
```

instead of

```
    scanf("%d", &n);
```

This error is not generally detected at compile time.

Exercise 7-4. Write a private version of `scanf` analogous to `minprintf` from the previous section. □

Exercise 7-5. Rewrite the postfix calculator of Chapter 4 to use `scanf` and/or `sscanf` to do the input and number conversion. □