SECOND EDITION

THE C PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Table of Contents

Chapter 4 - Functions and Program Structure
4.6 Static Variables....13
4.7 Register Variables....13
4.8 Block Structure....14
4.9 Initialization....14
4.10 Recursion....15
4.11 The C Preprocessor....17
4.11.1 File Inclusion....17
4.11.2 Macro Substitution....17
4.11.3 Conditional Inclusion....19

Chapter 5 - Pointers and Arrays....1
5.1 Pointers and Addresses....1
5.2 Pointers and Function Arguments....2
5.3 Pointers and Arrays....5
5.4 Address Arithmetic....7
5.5 Character Pointers and Functions....10
5.6 Pointer Arrays; Pointers to Pointers....13
5.7 Multi-dimensional Arrays....15
5.8 Initialization of Pointer Arrays....17
5.9 Pointers vs. Multi-dimensional Arrays....17
5.10 Command-line Arguments....18
5.11 Pointers to Functions....22
5.12 Complicated Declarations....24

Chapter 6 - Structures....1
6.1 Basics of Structures....1
6.2 Structures and Functions....3
6.3 Arrays of Structures....5
6.4 Pointers to Structures....8
6.5 Self-referential Structures....9
6.6 Table Lookup....12
6.7 Typedef....14
6.8 Unions....16
6.9 Bit-fields....17

Chapter 7 - Input and Output....1
7.1 Standard Input and Output....1
7.2 Formatted Output - printf....2
7.3 Variable-length Argument Lists....3
7.4 Formatted Input - Scanf....4
7.5 File Access....7
7.6 Error Handling - Stderr and Exit....9
7.7 Line Input and Output....10
7.8 Miscellaneous Functions....11
7.8.1 String Operations....11
7.8.2 Character Class Testing and Conversion....12
7.8.3 Ungetc....12
7.8.4 Command Execution....12
7.8.5 Storage Management....12
7.8.6 Mathematical Functions....13
7.8.7 Random Number generation....13

Table of Contents

Chapter 8 - The UNIX System Interface....1
8.1 File Descriptors....1
8.2 Low Level I/O - Read and Write....1
8.3 Open, Creat, Close, Unlink....3
8.4 Random Access - Lseek....5
8.5 Example - An implementation of Fopen and Getc....5
8.6 Example - Listing Directories....8
8.7 Example - A Storage Allocator....12

Appendix A - Reference Manual....1
A.1 Introduction....1
A.2 Lexical Conventions....1
A.2.1 Tokens....1
A.2.2 Comments....1
A.2.3 Identifiers....1
A.2.4 Keywords....1
A.2.5 Constants....2
A.2.6 String Literals....3
A.3 Syntax Notation....4
A.4 Meaning of Identifiers....4
A.4.1 Storage Class....4
A.4.2 Basic Types....4
A.4.3 Derived types....5
A.4.4 Type Qualifiers....5
A.5 Objects and Lvalues....6
A.6 Conversions....6
A.6.1 Integral Promotion....6
A.6.2 Integral Conversions....6
A.6.3 Integer and Floating....6
A.6.4 Floating Types....6
A.6.5 Arithmetic Conversions....7
A.6.6 Pointers and Integers....7
A.6.7 Void....8
A.6.8 Pointers to Void....8
A.7 Expressions....8
A.7.1 Pointer Conversion....8
A.7.2 Primary Expressions....9
A.7.3 Postfix Expressions....9
A.7.4 Unary Operators....11
A.7.5 Casts....12
A.7.6 Multiplicative Operators....13
A.7.7 Additive Operators....13
A.7.8 Shift Operators....14
A.7.9 Relational Operators....14
A.7.10 Equality Operators....14
A.7.11 Bitwise AND Operator....15
A.7.12 Bitwise Exclusive OR Operator....15
A.7.13 Bitwise Inclusive OR Operator....15
A.7.14 Logical AND Operator....15
A.7.15 Logical OR Operator....15
A.7.16 Conditional Operator....16
A.7.17 Assignment Expressions....16

```
 40       4
 60      15
 80      26
100      37
...
```

The more serious problem is that because we have used integer arithmetic, the Celsius temperatures are not very accurate; for instance, 0ºF is actually about -17.8ºC, not -17. To get more accurate answers, we should use floating-point arithmetic instead of integer. This requires some changes in the program. Here is the second version:

```
#include <stdio.h>

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

  lower = 0;      /* lower limit of temperatuire scale */
  upper = 300;    /* upper limit */
  step = 20;      /* step size */

  fahr = lower;
  while (fahr <= upper) {
      celsius = (5.0/9.0) * (fahr-32.0);
      printf("%3.0f %6.1f\n", fahr, celsius);
      fahr = fahr + step;
  }
}
```

This is much the same as before, except that `fahr` and `celsius` are declared to be `float` and the formula for conversion is written in a more natural way. We were unable to use `5/9` in the previous version because integer division would truncate it to zero. A decimal point in a constant indicates that it is floating point, however, so `5.0/9.0` is not truncated because it is the ratio of two floating-point values.

If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done. If we had written `(fahr-32)`, the `32` would be automatically converted to floating point. Nevertheless, writing floating-point constants with explicit decimal points even when they have integral values emphasizes their floating-point nature for human readers.

The detailed rules for when integers are converted to floating point are in Chapter 2. For now, notice that the assignment

```
fahr = lower;
```

and the test

```
while (fahr <= upper)
```

also work in the natural way - the `int` is converted to `float` before the operation is done.

The `printf` conversion specification `%3.0f` says that a floating-point number (here `fahr`) is to be printed at least three characters wide, with no decimal point and no fraction digits. `%6.1f` describes another number (`celsius`) that is to be printed at least six characters wide, with 1 digit after the decimal point. The output looks like this:

The standard library provides several functions for reading or writing one character at a time, of which `getchar` and `putchar` are the simplest. Each time it is called, `getchar` reads the *next input character* from a text stream and returns that as its value. That is, after

```
c = getchar();
```

the variable `c` contains the next character of input. The characters normally come from the keyboard; input from files is discussed in Chapter 7.

The function `putchar` prints a character each time it is called:

```
putchar(c);
```

prints the contents of the integer variable `c` as a character, usually on the screen. Calls to `putchar` and `printf` may be interleaved; the output will appear in the order in which the calls are made.

1.5.1 File Copying

Given `getchar` and `putchar`, you can write a surprising amount of useful code without knowing anything more about input and output. The simplest example is a program that copies its input to its output one character at a time:

```
read a character
    while (charater is not end-of-file indicator)
        output the character just read
        read a character
```

Converting this into C gives:

```
"temp0019.html">Chapter 2.)
```

Exercsise 1-6. Verify that the expression `getchar() != EOF` is 0 or 1.

Exercise 1-7. Write a program to print the value of `EOF`.

1.5.2 Character Counting

The next program counts characters; it is similar to the copy program.

```
#include <stdio.h>

/* count characters in input; 1st version */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

The statement

```
++nc;
```

presents a new operator, ++, which means *increment by one*. You could instead write `nc = nc + 1` but

`++nc` is more concise and often more efficient. There is a corresponding operator `--` to decrement by 1. The operators `++` and `--` can be either prefix operators (`++nc`) or postfix operators (`nc++`); these two forms have different values in expressions, as will be shown in Chapter 2, but `++nc` and `nc++` both increment `nc`. For the moment we will will stick to the prefix form.

The character counting program accumulates its count in a `long` variable instead of an int. `long` integers are at least 32 bits. Although on some machines, `int` and `long` are the same size, on others an `int` is 16 bits, with a maximum value of 32767, and it would take relatively little input to overflow an `int` counter. The conversion specification `%ld` tells `printf` that the corresponding argument is a `long` integer.

It may be possible to cope with even bigger numbers by using a `double` (double precision `float`). We will also use a `for` statement instead of a `while`, to illustrate another way to write the loop.

```
    "%.0f\n", nc);
}
```

`printf` uses `%f` for both `float` and `double`; `%.0f` suppresses the printing of the decimal point and the fraction part, which is zero.

The body of this `for` loop is empty, because all the work is done in the test and increment parts. But the grammatical rules of C require that a `for` statement have a body. The isolated semicolon, called a *null statement*, is there to satisfy that requirement. We put it on a separate line to make it visible.

Before we leave the character counting program, observe that if the input contains no characters, the `while` or `for` test fails on the very first call to `getchar`, and the program produces zero, the right answer. This is important. One of the nice things about `while` and `for` is that they test at the top of the loop, before proceeding with the body. If there is nothing to do, nothing is done, even if that means never going through the loop body. Programs should act intelligently when given zero-length input. The `while` and `for` statements help ensure that programs do reasonable things with boundary conditions.

1.5.3 Line Counting

The next program counts input lines. As we mentioned above, the standard library ensures that an input text stream appears as a sequence of lines, each terminated by a newline. Hence, counting lines is just counting newlines:

```
#include <stdio.h>

/* count lines in input */
main()
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

The body of the `while` now consists of an `if`, which in turn controls the increment `++nl`. The `if` statement tests the parenthesized condition, and if the condition is true, executes the statement (or group of statements in braces) that follows. We have again indented to show what is controlled by what.

The double equals sign == is the C notation for ``is equal to'' (like Pascal's single = or Fortran's `.EQ.`). This symbol is used to distinguish the equality test from the single = that C uses for assignment. A word of caution: newcomers to C occasionally write = when they mean ==. As we will see in Chapter 2, the result is usually a

```
}
```

The parameter `n` is used as a temporary variable, and is counted down (a `for` loop that runs backwards) until it becomes zero; there is no longer a need for the variable `i`. Whatever is done to `n` inside `power` has no effect on the argument that `power` was originally called with.

When necessary, it is possible to arrange for a function to modify a variable in a calling routine. The caller must provide the *address* of the variable to be set (technically a *pointer* to the variable), and the called function must declare the parameter to be a pointer and access the variable indirectly through it. We will cover pointers in Chapter 5.

The story is different for arrays. When the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array - there is no copying of array elements. By subscripting this value, the function can access and alter any argument of the array. This is the topic of the next section.

1.9 Character Arrays

The most common type of array in C is the array of characters. To illustrate the use of character arrays and functions to manipulate them, let's write a program that reads a set of text lines and prints the longest. The outline is simple enough:

```
while (there's another line)
    if (it's longer than the previous longest)
        (save it)
        (save its length)
print longest line
```

This outline makes it clear that the program divides naturally into pieces. One piece gets a new line, another saves it, and the rest controls the process.

Since things divide so cleanly, it would be well to write them that way too. Accordingly, let us first write a separate function `getline` to fetch the next line of input. We will try to make the function useful in other contexts. At the minimum, `getline` has to return a signal about possible end of file; a more useful design would be to return the length of the line, or zero if end of file is encountered. Zero is an acceptable end-of-file return because it is never a valid line length. Every text line has at least one character; even a line containing only a newline has length 1.

When we find a line that is longer than the previous longest line, it must be saved somewhere. This suggests a second function, `copy`, to copy the new line to a safe place.

Finally, we need a main program to control `getline` and `copy`. Here is the result.

```
"%s", longest);
    return 0;
}

/* getline:  read a line into s, return length  */
int getline(char s[],int lim)
{
    int c, i;

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

are, so we have chosen not to add error checking to it.

Exercise 1-16. Revise the main routine of the longest-line program so it will correctly print the length of arbitrary long input lines, and as much as possible of the text.

Exercise 1-17. Write a program to print all input lines that are longer than 80 characters.

Exercise 1-18. Write a program to remove trailing blanks and tabs from each line of input, and to delete entirely blank lines.

Exercise 1-19. Write a function `reverse(s)` that reverses the character string `s`. Use it to write a program that reverses its input a line at a time.

1.10 External Variables and Scope

The variables in `main`, such as `line`, `longest`, etc., are private or local to `main`. Because they are declared within `main`, no other function can have direct access to them. The same is true of the variables in other functions; for example, the variable `i` in `getline` is unrelated to the `i` in copy. Each local variable in a function comes into existence only when the function is called, and disappears when the function is exited. This is why such variables are usually known as *automatic* variables, following terminology in other languages. We will use the term automatic henceforth to refer to these local variables. (Chapter 4 discusses the `static` storage class, in which local variables do retain their values between calls.)

Because automatic variables come and go with function invocation, they do not retain their values from one call to the next, and must be explicitly set upon each entry. If they are not set, they will contain garbage.

As an alternative to automatic variables, it is possible to define variables that are *external* to all functions, that is, variables that can be accessed by name by any function. (This mechanism is rather like Fortran COMMON or Pascal variables declared in the outermost block.) Because external variables are globally accessible, they can be used instead of argument lists to communicate data between functions. Furthermore, because external variables remain in existence permanently, rather than appearing and disappearing as functions are called and exited, they retain their values even after the functions that set them have returned.

An external variable must be *defined*, exactly once, outside of any function; this sets aside storage for it. The variable must also be *declared* in each function that wants to access it; this states the type of the variable. The declaration may be an explicit `extern` statement or may be implicit from context. To make the discussion concrete, let us rewrite the longest-line program with `line`, `longest`, and `max` as external variables. This requires changing the calls, declarations, and bodies of all three functions.

```
"%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';
```

normally be the natural size for a particular machine. `short` is often 16 bits long, and `int` either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the the restriction that `short`s and ints are at least 16 bits, `long`s are at least 32 bits, and `short` is no longer than `int`, which is no longer than `long`.

The qualifier `signed` or `unsigned` may be applied to `char` or any integer. `unsigned` numbers are always positive or zero, and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type. So, for instance, if `char`s are 8 bits, `unsigned char` variables have values between 0 and 255, while `signed char`s have values between -128 and 127 (in a two's complement machine.) Whether plain `char`s are signed or unsigned is machine-dependent, but printable characters are always positive.

The type `long double` specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; `float`, `double` and `long double` could represent one, two or three distinct sizes.

The standard headers `<limits.h>` and `<float.h>` contain symbolic constants for all of these sizes, along with other properties of the machine and compiler. These are discussed in Appendix B.

Exercise 2-1. Write a program to determine the ranges of `char`, `short`, `int`, and `long` variables, both `signed` and `unsigned`, by printing appropriate values from standard headers and by direct computation. Harder if you compute them: determine the ranges of the various floating-point types.

2.3 Constants

An integer constant like `1234` is an `int`. A `long` constant is written with a terminal `l` (ell) or `L`, as in `123456789L`; an integer constant too big to fit into an `int` will also be taken as a `long`. Unsigned constants are written with a terminal `u` or `U`, and the suffix `ul` or `UL` indicates `unsigned long`.

Floating-point constants contain a decimal point (`123.4`) or an exponent (`1e-2`) or both; their type is `double`, unless suffixed. The suffixes `f` or `F` indicate a `float` constant; `l` or `L` indicate a `long double`.

The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading `0` (zero) on an integer constant means octal; a leading `0x` or `0X` means hexadecimal. For example, decimal 31 can be written as `037` in octal and `0x1f` or `0x1F` in hex. Octal and hexadecimal constants may also be followed by `L` to make them `long` and `U` to make them `unsigned`: `0XFUL` is an *unsigned long* constant with value 15 decimal.

A `character constant` is an integer, written as one character within single quotes, such as `'x'`. The value of a character constant is the numeric value of the character in the machine's character set. For example, in the ASCII character set the character constant `'0'` has the value 48, which is unrelated to the numeric value 0. If we write `'0'` instead of a numeric value like 48 that depends on the character set, the program is independent of the particular value and easier to read. Character constants participate in numeric operations just as any other integers, although they are most often used in comparisons with other characters.

Certain characters can be represented in character and string constants by escape sequences like `\n` (newline); these sequences look like two characters, but represent only one. In addition, an arbitrary byte-sized bit pattern can be specified by

```
'\ooo'
```

where *ooo* is one to three octal digits (0...7) or by

```
'\xhh'
```

Another example of a similar construction comes from the `getline` function that we wrote in Chapter 1, where we can replace

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

by the more compact

```
if (c == '\n')
   s[i++] = c;
```

As a third example, consider the standard function `strcat(s,t)`, which concatenates the string `t` to the end of string `s`. `strcat` assumes that there is enough space in `s` to hold the combination. As we have written it, `strcat` returns no value; the standard library version returns a pointer to the resulting string.

```
/* strcat:  concatenate t to end of s; s must be big enough */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0') /* find end of s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copy t */
        ;
}
```

As each member is copied from `t` to `s`, the postfix ++ is applied to both `i` and `j` to make sure that they are in position for the next pass through the loop.

Exercise 2-4. Write an alternative version of `squeeze(s1,s2)` that deletes each character in `s1` that matches any character in the string `s2`.

Exercise 2-5. Write the function `any(s1,s2)`, which returns the first location in a string `s1` where any character from the string `s2` occurs, or `-1` if `s1` contains no characters from `s2`. (The standard library function `strpbrk` does the same job but returns a pointer to the location.)

2.9 Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, `char`, `short`, `int`, and `long`, whether signed or unsigned.

`&`	bitwise AND
`|`	bitwise inclusive OR
`^`	bitwise exclusive OR
`<<`	left shift
`>>`	right shift
`~`	one's complement (unary)

The bitwise AND operator `&` is often used to mask off some set of bits, for example

```
n = n & 0177;
```

sets to zero all but the low-order 7 bits of `n`.

The *statement* is executed, then *expression* is evaluated. If it is true, *statement* is evaluated again, and so on. When the expression becomes false, the loop terminates. Except for the sense of the test, `do-while` is equivalent to the Pascal `repeat-until` statement.

Experience shows that `do-while` is much less used than `while` and `for`. Nonetheless, from time to time it is valuable, as in the following function `itoa`, which converts a number to a character string (the inverse of `atoi`). The job is slightly more complicated than might be thought at first, because the easy methods of generating the digits generate them in the wrong order. We have chosen to generate the string backwards, then reverse it.

```
/* itoa:  convert n to characters in s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0)  /* record sign */
        n = -n;          /* make n positive */
    i = 0;
    do {       /* generate digits in reverse order */
        s[i++] = n % 10 + '0';  /* get next digit */
    } while ((n /= 10) > 0);    /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

The `do-while` is necessary, or at least convenient, since at least one character must be installed in the array `s`, even if `n` is zero. We also used braces around the single statement that makes up the body of the `do-while`, even though they are unnecessary, so the hasty reader will not mistake the `while` part for the *beginning* of a `while` loop.

Exercise 3-4. In a two's complement number representation, our version of `itoa` does not handle the largest negative number, that is, the value of `n` equal to $(2^{wordsize-1})$. Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs.

Exercise 3-5. Write the function `itob(n,s,b)` that converts the integer `n` into a base `b` character representation in the string `s`. In particular, `itob(n,s,16)` formats `s` as a hexadecimal integer in `s`.

Exercise 3-6. Write a version of `itoa` that accepts three arguments instead of two. The third argument is a minimum field width; the converted number must be padded with blanks on the left if necessary to make it wide enough.

3.7 Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The `break` statement provides an early exit from `for`, `while`, and `do`, just as from `switch`. A `break` causes the innermost enclosing loop or `switch` to be exited immediately.

The following function, `trim`, removes trailing blanks, tabs and newlines from the end of a string, using a `break` to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
/* trim:  remove trailing blanks, tabs, newlines */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
```

```
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

`strlen` returns the length of the string. The `for` loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when `n` becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters.

The `continue` statement is related to `break`, but less often used; it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin. In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. The `continue` statement applies only to loops, not to `switch`. A `continue` inside a `switch` inside a loop causes the next loop iteration.

As an example, this fragment processes only the non-negative elements in the array `a`; negative values are skipped.

```
for (i = 0; i < n; i++)
    if (a[i] < 0)    /* skip negative elements */
        continue;
    ... /* do positive elements */
```

The `continue` statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

3.8 Goto and labels

C provides the infinitely-abusable `goto` statement, and labels to branch to. Formally, the `goto` statement is never necessary, and in practice it is almost always easy to write code without it. We have not used `goto` in this book.

Nevertheless, there are a few situations where gotos may find a place. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The `break` statement cannot be used directly since it only exits from the innermost loop. Thus:

```
    for ( ... )
        for ( ... ) {
            ...
            if (disaster)
                goto error;
        }
    ...
error:
    /* clean up the mess */
```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several places.

A label has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the `goto`. The scope of a label is the entire function.

As another example, consider the problem of determining whether two arrays `a` and `b` have an element in common. One possibility is

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
```

4.2 Functions Returning Non-integers

So far our examples of functions have returned either no value (`void`) or an `int`. What if a function must return some other type? many numerical functions like `sqrt`, `sin`, and `cos` return `double`; other specialized functions return other types. To illustrate how to deal with this, let us write and use the function `atof(s)`, which converts the string `s` to its double-precision floating-point equivalent. `atof` if an extension of `atoi`, which we showed versions of in Chapters 2 and 3. It handles an optional sign and decimal point, and the presence or absence of either part or fractional part. Our version is *not* a high-quality input conversion routine; that would take more space than we care to use. The standard library includes an `atof`; the header `<stdlib.h>` declares it.

First, `atof` itself must declare the type of value it returns, since it is not `int`. The type name precedes the function name:

```
"\t%g\n", sum += atof(line));
    return 0;
}
```

The declaration

```
double sum, atof(char []);
```

says that `sum` is a `double` variable, and that `atof` is a function that takes one `char[]` argument and returns a `double`.

The function `atof` must be declared and defined consistently. If `atof` itself and the call to it in `main` have inconsistent types in the same source file, the error will be detected by the compiler. But if (as is more likely) `atof` were compiled separately, the mismatch would not be detected, `atof` would return a `double` that `main` would treat as an `int`, and meaningless answers would result.

In the light of what we have said about how declarations must match definitions, this might seem surprising. The reason a mismatch can happen is that if there is no function prototype, a function is implicitly declared by its first appearance in an expression, such as

```
sum += atof(line)
```

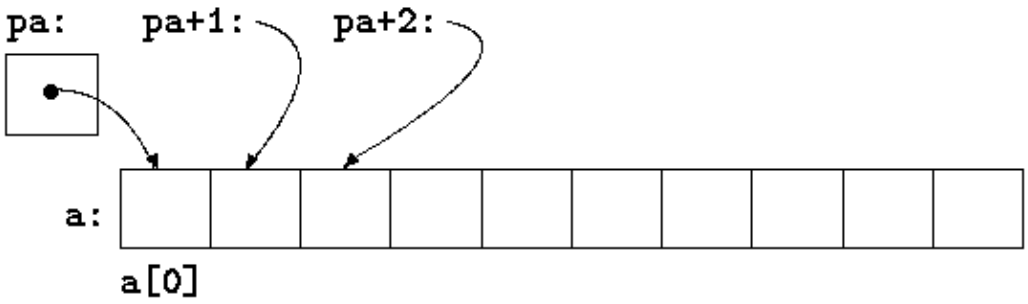
If a name that has not been previously declared occurs in an expression and is followed by a left parentheses, it is declared by context to be a function name, the function is assumed to return an `int`, and nothing is assumed about its arguments. Furthermore, if a function declaration does not include arguments, as in

```
double atof();
```

that too is taken to mean that nothing is to be assumed about the arguments of `atof`; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new C programs. If the function takes arguments, declare them; if it takes no arguments, use `void`.

Given `atof`, properly declared, we could write `atoi` (convert a string to `int`) in terms of it:

```
/* atoi:  convert string s to integer using atof */
int atoi(char s[])
{
    double atof(char s[]);

    return (int) atof(s);
}
```



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 object beyond `pa`.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

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

`pa` and `a` have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as

```
pa = a;
```

Rather more surprising, 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 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 might use it with a subscript; `pa[i]` is identical to `*(pa+i)`. In short, an array-and-index expression is equivalent to one written as a pointer and offset.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; constructions like `a=pa` and `a++` are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of `strlen`, which computes the length of a string.

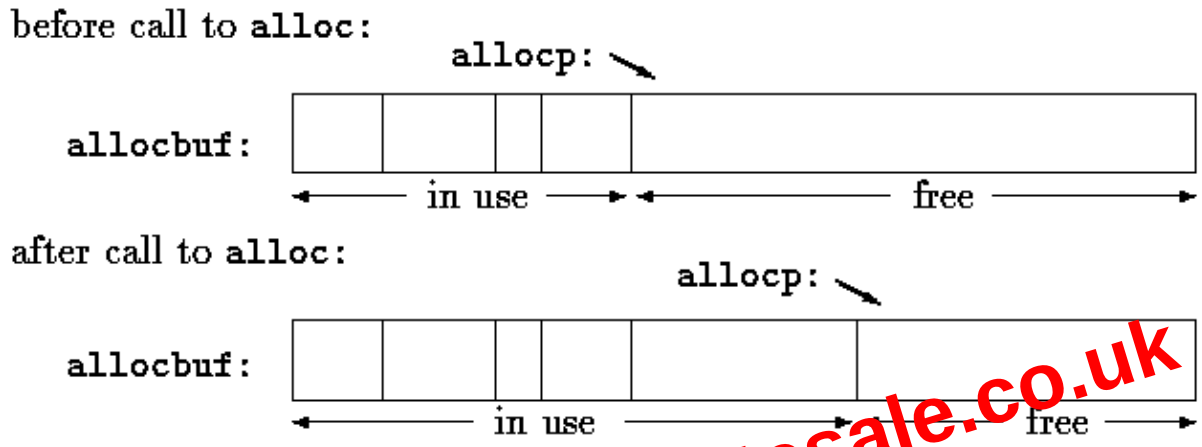
```
/* strlen:  return length of string s */
int strlen(char *s)
{
    int n;

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

Since `s` is a pointer, incrementing it is perfectly legal; `s++` has no effect on the character string in the function that called `strlen`, but merely increments `strlen`'s private copy of the pointer. That means that calls like

```
strlen("hello, world");    /* string constant */
strlen(array);             /* char array[100]; */
```

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




```
#define ALLOCSIZE 10000 /* size of available space */

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

char *alloc(int n)    /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) {  /* it fits */
        allocp += n;
        return allocp - n; /* old p */
    } else      /* not enough room */
        return 0;
}

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

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the address 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 the beginning of `allocbuf`, which is the next free position when the program starts. This could also have been written

is counted down.

With input and output under control, we can proceed to sorting. The quicksort from Chapter 4 needs minor changes: the declarations have to be modified, and the comparison operation must be done by calling `strcmp`. The algorithm remains the same, which gives us some confidence that it will still work.

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right)  /* do nothing if array contains */
        return;         /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Similarly, the swap routine needs only trivial changes:

```
/* swap:  interchange v[i] and v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```


Since any individual element of `v` (namely `v[i]`) is a character pointer, `temp` must be also, so one can be copied to the other.

Exercise 5-7. Rewrite `readlines` to store lines in an array supplied by `main`, rather than calling `alloc` to maintain storage. How much faster is the program?

5.7 Multi-dimensional Arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers. In this section, we will show some of their properties.

Consider the problem of date conversion, from day of the month to day of the year and vice versa. For example, March 1 is the 60th day of a non-leap year, and the 61st day of a leap year. Let us define two functions to do the conversions: `day_of_year` converts the month and day into the day of the year, and `month_day` converts the day of the year into the month and day. Since this latter function computes two values, the month and day arguments will be pointers:

```
month_day(1988, 60, &m, &d)
```

sets `m` to 2 and `d` to 29 (February 29th).

These functions both need the same information, a table of the number of days in each month (``thirty days hath September ..."). Since the number of days per month differs for leap years and non-leap years, it's easier

This shows that the format argument of `printf` can be an expression too.

As a second example, let us make some enhancements to the pattern-finding program from Section 4.1. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX program `grep`, let us enhance 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

int getline(char *line, int max);

/* find:  print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

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

The standard library function `strstr(s,t)` returns a pointer to the first occurrence of the string `t` in the string `s`, or `NULL` if there is none. It is declared in `<string.h>`.

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

A common convention for C programs on UNIX systems is that an argument that begins 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 -npattern
```

will print each line that doesn't match the pattern, preceded by its line number.

Optional arguments should be permitted in any order, and the rest of the program should be independent of the number of arguments that we present. Furthermore, it is convenient for users if option arguments can be combined, as in

```
find -nx pattern
```

Here is the program:

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

int getline(char *line, int max);

/* find: print lines that match pattern from 1st arg */
```

is syntactically analogous to

```
int x, y, z;
```

in the sense that each statement declares `x`, `y` and `z` to be variables of the named type and causes space to be set aside for them.

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or shape of a structure. If the declaration is tagged, however, the tag can be used later in definitions of instances of the structure. For example, given the declaration of `point` above,

```
struct point pt;
```

defines a variable `pt` which is a structure of type `struct point`. A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct maxpt = { 320, 200 };
```

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type.

A member of a particular structure is referred to in an expression by a construction of the form

structure-name.member

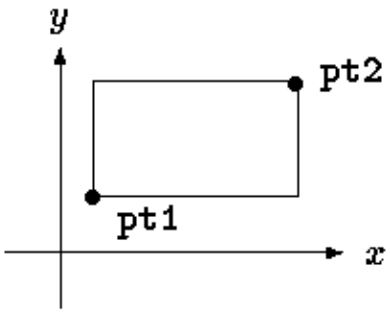
The structure member operator ``." connects the structure name and the member name. To print the coordinates of the point `pt`, for instance,

```
printf("%d,%d", pt.x, pt.y);
```

or to compute the distance from the origin (0,0) to `pt`,

```
double dist, sqrt(double);

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```


Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:




```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

The `rect` structure contains two `point` structures. If we declare `screen` as

```
{ "break", 0 },
{ "case", 0 },
...
```

but inner braces are not necessary when the initializers are simple variables or character strings, and when all are present. As usual, the number of entries in the array `keytab` will be computed if the initializers are present and the `[]` is left empty.

The keyword counting program begins with the definition of `keytab`. The main routine reads the input by repeatedly calling a function `getword` that fetches one word at a time. Each word is looked up in `keytab` with a version of the binary search function that we wrote in Chapter 3. The list of keywords must be sorted in increasing order in the table.

```
"%4d %s\n",
                keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch:  find word in tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```


We will show the function `getword` in a moment; for now it suffices to say that each call to `getword` finds a word, which is copied into the array named as its first argument.

The quantity `NKEYS` is the number of keywords in `keytab`. Although we could count this by hand, it's a lot easier and safer to do it by machine, especially if the list is subject to change. One possibility would be to terminate the list of initializers with a null pointer, then loop along `keytab` until the end is found.

But this is more than is needed, since the size of the array is completely determined at compile time. The size of the array is the size of one entry times the number of entries, so the number of entries is just

size of `keytab` / *size of* `struct key`

C provides a compile-time unary operator called `sizeof` that can be used to compute the size of any object. The expressions

```
sizeof object
```

and

```
sizeof (type name)
```

yield an integer equal to the size of the specified object or type in bytes. (Strictly, `sizeof` produces an unsigned integer value whose type, `size_t`, is defined in the header `<stddef.h>`.) An object can be a variable or array or structure. A type name can be the name of a basic type like `int` or `double`, or a derived type like a structure or a pointer.

In our case, the number of keywords is the size of the array divided by the size of one element. This computation is used in a `#define` statement to set the value of `NKEYS`:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Another way to write this is to divide the array size by the size of a specific element:

```
#define NKEYS (sizeof keytab / sizeof(keytab[0]))
```

This has the advantage that it does not need to be changed if the type changes.

A `sizeof` can not be used in a `#if` line, because the preprocessor does not parse type names. But the expression in the `#define` is not evaluated by the preprocessor, so the code here is legal.

Now for the function `getword`. We have written a more general `getword` than is necessary for this program, but it is not complicated. `getword` fetches the next ``word'' from the input, where a word is either a string of letters and digits beginning with a letter, or a single non-white space character. The function value is the first character of the word, or `EOF` for end of file, or the character itself if it is not alphabetic.

```
/* getword:  get next word or character from input */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```


`getword` uses the `getch` and `ungetch` that we wrote in Chapter 4. When the collection of an alphanumeric token stops, `getword` has gone one character too far. The call to `ungetch` pushes that character back on the input for the next call. `getword` also uses `isspace` to skip whitespace, `isalpha` to identify letters, and `isalnum` to identify letters and digits; all are from the standard header `<ctype.h>`.

Exercise 6-1. Our version of `getword` does not properly handle underscores, string constants, comments, or preprocessor control lines. Write a better version.

6.4 Pointers to Structures

To illustrate some of the considerations involved with pointers to and arrays of structures, let us write the keyword-counting program again, this time using pointers instead of array indices.

The external declaration of `keytab` need not change, but `main` and `binsearch` do need modification.

```
"%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struck key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

There are several things worthy of note here. First, the declaration of `binsearch` must indicate that it returns a pointer to `struct key` instead of an integer; this is declared both in the function prototype and in `binsearch`. If `binsearch` finds the word, it returns a pointer to it; if it fails, it returns `NULL`.

Second, the elements of `keytab` are now accessed by pointers. This requires significant changes in `binsearch`.

The initializers for `low` and `high` are now pointers to the beginning and just past the end of the table.

The computation of the middle element can no longer be simply

```
mid = (low+high) / 2    /* WRONG */
```

because the addition of pointers is illegal. Subtraction is legal, however, so `high-low` is the number of elements, and thus

```
mid = low + (high-low) / 2
```

sets `mid` to the element halfway between `low` and `high`.

The most important change is to adjust the algorithm to make sure that it does not generate an illegal pointer or attempt to access an element outside the array. The problem is that `&tab[-1]` and `&tab[n]` are both outside the limits of the array `tab`. The former is strictly illegal, and it is illegal to dereference the latter. The language definition does guarantee, however, that pointer arithmetic that involves the first element beyond the end of an array (that is, `&tab[n]`) will work correctly.

In `main` we wrote

To show that there is nothing special about functions like `fgets` and `fputs`, here they are, copied from the standard library on our system:

```
/* fgets:  get at most n chars from iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs:  put string s on file iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

For no obvious reason, the standard specifies different return values for `ferror` and `fputs`.

It is easy to implement our `getline` from `fgets`:

```
/* getline:  read a line, return length */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}
```


Exercise 7-6. Write a program to compare two files, printing the first line where they differ.

Exercise 7-7. Modify the pattern finding program of Chapter 5 to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found?

Exercise 7-8. Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file.

7.8 Miscellaneous Functions

The standard library provides a wide variety of functions. This section is a brief synopsis of the most useful. More details and many other functions can be found in Appendix B.

7.8.1 String Operations

We have already mentioned the string functions `strlen`, `strcpy`, `strcat`, and `strcmp`, found in `<string.h>`. In the following, `s` and `t` are `char *`'s, and `c` and `n` are `int`s.

```
#define S_IFCHR   0020000  /* character special */
#define S_IFBLK   0060000  /* block special */
#define S_IFREG   0010000  /* regular */
/* ... */
```

Now we are ready to write the program `fsize`. If the mode obtained from `stat` indicates that a file is not a directory, then the size is at hand and can be printed directly. If the name is a directory, however, then we have to process that directory one file at a time; it may in turn contain sub-directories, so the process is recursive.

The main routine deals with command-line arguments; it hands each argument to the function `fsize`.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>      /* flags for read and write */
#include <sys/types.h>  /* typedefs */
#include <sys/stat.h>   /* structure returned by stat */
#include "dirent.h"

void fsize(char *)

/* print file name */
main(int argc, char **argv)
{
    if (argc == 1)  /* default: current directory */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```


The function `fsize` prints the size of the file. If the file is a directory, however, `fsize` first calls `dirwalk` to handle all the files in it. Note how the flag names `S_IFMT` and `S_IFDIR` are used to decide if the file is a directory. Parenthesization matters, because the precedence of `&` is lower than that of `==`.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));

/* fsize:  print the name of file "name" */
void fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't access %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

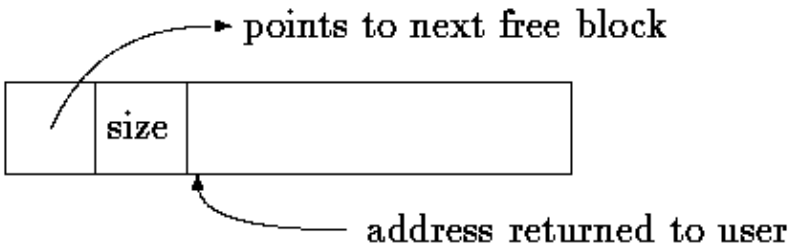
The function `dirwalk` is a general routine that applies a function to each file in a directory. It opens the directory, loops through the files in it, calling the function on each, then closes the directory and returns. Since `fsize` calls `dirwalk` on each directory, the two functions call each other recursively.

```
#define MAX_PATH 1024

/* dirwalk:  apply fcn to all files in dir */
void dirwalk(char *dir, void (*fcn)(char *))
```

```
        unsigned size;      /* size of this block */
    } s;
    Align x;            /* force alignment of blocks */
};

typedef union header Header;
```

The `Align` field is never used; it just forces each header to be aligned on a worst-case boundary.

In `malloc`, the requested size in characters is rounded up to the proper number of header-sized units; the block that will be allocated contains one more unit, for the header itself, and this is the value recorded in the `size` field of the header. The pointer returned by `malloc` points at the free space, not at the header itself. The user can do anything with the space requested, but if anything is written outside of the allocated space the list is likely to be scrambled.



A block returned by **malloc**

The size field is necessary because the blocks controlled by `malloc` need not be contiguous - it is not possible to compute sizes by pointer arithmetic.

The variable `base` is used to get started. If `freep` is `NULL`, as it is at the first call of `malloc`, then a degenerate free list is created; it contains one block of size zero and points to itself. In any case, the free list is then searched. The search for a free block of adequate size begins at the point (`freep`) where the last block was found; this strategy helps keep the list homogeneous. If a too-big block is found, the tail end is returned to the user; in this way the header of the original needs only to have its size adjusted. In all cases, the pointer returned to the user points to the free space within the block, which begins one unit beyond the header.

```
static Header base;       /* empty list to get started */
static Header *freep = NULL;     /* start of free list */

/* malloc:  general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *moreroce(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
    if ((prevp = freep) == NULL) {   /* no free list yet */
        base.s.ptr = freeptr = prevptr = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {  /* big enough */
            if (p->s.size == nunits)  /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {              /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
```

A.7.3.3 Structure References

A postfix expression followed by a dot followed by an identifier is a postfix expression. The first operand expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and its type is the type of the member. The expression is an lvalue if the first expression is an lvalue, and if the type of the second expression is not an array type.

A postfix expression followed by an arrow (built from − and >) followed by an identifier is a postfix expression. The first operand expression must be a pointer to a structure or union, and the identifier must name a member of the structure or union. The result refers to the named member of the structure or union to which the pointer expression points, and the type is the type of the member; the result is an lvalue if the type is not an array type.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in Par.A.8.3.

> In the first edition of this book, it was already the rule that a member name in such an expression had to belong to the structure or union mentioned in the postfix expression; however, a note admitted that this rule was not firmly enforced. Recent compilers, and ANSI, do enforce it.

A.7.3.4 Postfix Incrementation

A postfix expression followed by a ++ or −− operator is a postfix expression. The value of the expression is the value of the operand. After the value is noted, the operand is incremented ++ or decremented −− by 1. The operand must be an lvalue; see the discussion of additive operators (Par.A.7.7) and assignment (Par.A.7.17) for further constraints on the operand and details of the operation. The result is not an lvalue.

A.7.4 Unary Operators

Expressions with unary operators group right-to-left.

unary-expression:
postfix-expression
`++` *unary expression*
`--` *unary expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof(`*type-name*`)`

unary operator: one of
`& * + - ~ !`

A.7.4.1 Prefix Incrementation Operators

A unary expression followed by a ++ or −− operator is a unary expression. The operand is incremented ++ or decremented −− by 1. The value of the expression is the value after the incrementation (decrementation). The operand must be an lvalue; see the discussion of additive operators (Par.A.7.7) and assignment (Par.A.7.17) for further constraints on the operands and details of the operation. The result is not an lvalue.

A.7.4.2 Address Operator

The unary operator `&` takes the address of its operand. The operand must be an lvalue referring neither to a bit-field nor to an object declared as `register`, or must be of function type. The result is a pointer to the object or function referred to by the lvalue. If the type of the operand is *T*, the type of the result is ``pointer to *T*."

type specifier:
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
struct-or-union-specifier
enum-specifier
typedef-name

At most one of the words `long` or `short` may be specified together with `int`; the meaning is the same if `int` is not mentioned. The word `long` may be specified together with `double`. At most one of `signed` or `unsigned` may be specified together with `int` or any of its `short` or `long` varieties, or with `char`. Either may appear alone in which case `int` is understood. The `signed` specifier is useful for forcing `char` objects to carry a sign; it is permissible but redundant with other integral types.

Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be `int`.

Types may also be qualified, to indicate special properties of the objects being declared.

type-qualifier:
`const`
`volatile`

Type qualifiers may appear with any type specifier. A `const` object may be initialized, but not thereafter assigned to. There are no implementation-dependent semantics for `volatile` objects.

> The `const` and `volatile` properties are new with the ANSI standard. The purpose of `const` is to announce objects that may be placed in read-only memory, and perhaps to increase opportunities for optimization. The purpose of `volatile` is to force an implementation to suppress optimization that could otherwise occur. For example, for a machine with memory-mapped input/output, a pointer to a device register might be declared as a pointer to `volatile`, in order to prevent the compiler from removing apparently redundant references through the pointer. Except that it should diagnose explicit attempts to change `const` objects, a compiler may ignore these qualifiers.

A.8.3 Structure and Union Declarations

A structure is an object consisting of a sequence of named members of various types. A union is an object that contains, at different times, any of several members of various types. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ { *struct-declaration-list* }
struct-or-union identifier

struct-or-union:
`struct`
`union`

A struct-declaration-list is a sequence of declarations for the members of the structure or union:

struct-declaration-list:
struct declaration
struct-declaration-list struct declaration

struct-declaration: *specifier-qualifier-list struct-declarator-list*`;`

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct-declarator
struct-declarator-list `,` *struct-declarator*

Usually, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *bit-field*; its length is set off from the declarator for the field name by a colon.

struct-declarator:
declarator *declarator*$_{opt}$ `:` *constant-expression*

A type specifier of the form

struct-or-union identifier `{` *struct-declaration-list* `}`

declares the identifier to be the *tag* of the structure or union specified by the list. A subsequent declaration in the same or an inner scope may refer to the same type by using the tag in a specifier without the list:

struct-or-union identifier

If a specifier with a tag but without a list appears when the tag is not declared, an *incomplete type* is specified. Objects with an incomplete structure or union type may be mentioned in contexts where their size is not needed, for example in declarations (see definitions), for specifying a pointer, or for creating a `typedef`, but not otherwise. The type becomes complete on occurrence of a subsequent specifier with that tag, and containing a declaration list. Even in specifiers with a list, the structure or union type being declared is incomplete within the list, and becomes complete only at the `}` terminating the specifier.

A structure may not contain a member of incomplete type. Therefore, it is impossible to declare a structure or union containing an instance of itself. However, besides giving a name to the structure or union type, tags allow definition of self-referential structures; a structure or union may contain a pointer to an instance of itself, because pointers to incomplete types may be declared.

A very special rule applies to declarations of the form

struct-or-union identifier`;`

that declare a structure or union, but have no declaration list and no declarators. Even if the identifier is a structure or union tag already declared in an outer scope (Par.A.11.1), this declaration makes the identifier the tag of a new, incompletely-typed structure or union in the current scope.

> This recondite is new with ANSI. It is intended to deal with mutually-recursive structures declared in an inner scope, but whose tags might already be declared in the outer scope.

A structure or union specifier with a list but no tag creates a unique type; it can be referred to directly only in the declaration of which it is a part.

and the type of the identifier in the declaration `T D1` is ``*type-modifier* `T`," the type of the identifier of `D` is ``*type-modifier type-qualifier-list* pointer to `T`." Qualifiers following `*` apply to pointer itself, rather than to the object to which the pointer points.

For example, consider the declaration

```
int *ap[];
```

Here, `ap[]` plays the role of `D1`; a declaration ``int ap[]" (below) would give `ap` the type ``array of int," the type-qualifier list is empty, and the type-modifier is ``array of." Hence the actual declaration gives `ap` the type ``array to pointers to `int`."

As other examples, the declarations

```
int i, *pi, *const cpi = i
const int ci = 3, *pci;
```

declare an integer `i` and a pointer to an integer `pi`. The value of the constant pointer `cpi` may not be changed; it will always point to the same location, although the value to which it refers may be altered. The integer `ci` is constant, and may not be changed (though it may be initialized, as here.) The type of `pci` is ``pointer to `const int`," and `pci` itself may be changed to point to another place, but the value to which it points may not be altered by assigning through `pci`.

A.8.6.2 Array Declarators

In a declaration `T D` where `D` has the form

`D1 [`*constant-expression*$_{opt}$`]`

and the type of the identifier in the declaration `T D1` is ``*type-modifier* `T`," the type of the identifier of `D` is ``*type-modifier* array of `T`." If the constant expression is present it must have integral type, and value greater than 0. If the constant expression specifying the bound is missing, the array has an incomplete type.

An array may be constructed from an arithmetic type, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array). Any type from which an array is constructed must be complete; it must not be an array of structure of incomplete type. This implies that for a multi-dimensional array, only the first dimension may be missing. The type of an object of incomplete aray type is completed by another, complete, declaration for the object (Par.A.10.2), or by initializing it (Par.A.8.7). For example,

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. Also,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3 `X` 5 `X` 7. In complete detail, `x3d` is an array of three items: each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` may reasonably appear in an expression. The first three have type ``array,", the last has type `int`. More specifically, `x3d[i][j]` is an array of 7 integers, and `x3d[i]` is an array of 5 arrays of 7 integers.

The array subscripting operation is defined so that `E1[E2]` is identical to `*(E1+E2)`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation. Because of the conversion rules that apply to + and to arrays (Pars.A6.6, A.7.1, A.7.7), if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`.

declaration.

For example, the declaration

```
int f(), *fpi(), (*pfi)();
```

declares a function `f` returning an integer, a function `fpi` returning a pointer to an integer, and a pointer `pfi` to a function returning an integer. In none of these are the parameter types specified; they are old-style.

In the new-style declaration

```
int strcpy(char *dest, const char *source), rand(void);
```

`strcpy` is a function returning `int`, with two arguments, the first a character pointer, and the second a pointer to constant characters. The parameter names are effectively comments. The second function `rand` takes no arguments and returns `int`.

> Function declarators with parameter prototypes are, by far, the most important language change introduced by the ANSI standard. They offer an advantage over the ``old-style'' declarators of the first edition by providing error-detection and coercion of arguments across function calls, but at a cost: turmoil and confusion during their introduction, and the necessity of accomodating both forms. Some syntactic ugliness was required for the sake of compatibility, namely `void` as an explicit marker of new-style functions without parameters.
>
> The ellipsis notation ``, ...'' for variadic functions is also new, and, together with the macros in the standard header `<stdarg.h>`, formalizes a mechanism that was officially forbidden but unofficially condoned in the first edition.
>
> These notations were adapted from the C++ language.

A.8.7 Initialization

When an object is declared, its init declarator may specify an initial value for the identifier being declared. The initializer is preceded by `=`, and is either an expression, or a list of initializers nested in braces. A list may end with a comma, a nicety for neat formatting.

initializer:
  assignment-expression
  { *initializer-list* }
  { *initializer-list* , }

initializer-list:
  initializer
  initializer-list , *initializer*

All the expressions in the initializer for a static object or array must be constant expressions as described in Par.A.7.19. The expressions in the initializer for an `auto` or `register` object or array must likewise be constant expressions if the initializer is a brace-enclosed list. However, if the initializer for an automatic object is a single expression, it need not be a constant expression, but must merely have appropriate type for assignment to the object.

> The first edition did not countenance initialization of automatic structures, unions, or arrays. The ANSI standard allows it, but only by constant constructions unless the initializer can be expressed by a simple expression.

A static object not explicitly initialized is initialized as if it (or its members) were assigned the constant 0. The initial value of an automatic object not explicitly intialized is undefined.

> definitions for an externally linked object, throughout all the translation units of the program, are considered together instead of in each translation unit separately. If a definition occurs somewhere in the program, then the tentative definitions become merely declarations, but if no definition appears, then all its tentative definitions become a definition with initializer 0.

A.11 Scope and Linkage

A program need not all be compiled at one time: the source text may be kept in several files containing translation units, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, the *lexical scope* of an identifier which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects and functions with external linkage, which determines the connections between identifiers in separately compiled translation units.

A.11.1 Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These classes are: objects, functions, typedef names, and `enum` constants; labels; tags of structures or unions, and enumerations; and members of each structure or union individually.

> These rules differ in several ways from those described in the first edition of this manual. Labels did not previously have their own name space; tags of structures and unions each had a separate space, and in some implementations enumerations tags did as well; putting different kinds of tags into the same space is a new restriction. The most important departure from the first edition is that each structure or union creates a separate name space for its members, so that the same name may appear in several different structures. This rule has been common practice for several years.

The lexical scope of an object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a label is the whole of the function in which it appears. The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of a translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

A.11.2 Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit. All declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

As discussed in Par.A.10.2, the first external declaration for an identifier gives the identifier internal linkage if the `static` specifier is used, external linkage otherwise. If a declaration for an identifier within a block does not include the `extern` specifier, then the identifier has no linkage and is unique to the function. If it does include `extern`, and an external declaration for is active in the scope surrounding the block, then the identifier has the same linkage as the external declaration, and refers to the same object or function; but if no external declaration is visible, its linkage is external.

```
#define TABSIZE 100
int table[TABSIZE];
```

The definition

```
#define ABSDIFF(a, b)   ((a)>(b) ? (a)-(b) : (b)-(a))
```

defines a macro to return the absolute value of the difference between its arguments. Unlike a function to do the same thing, the arguments and returned value may have any arithmetic type or even be pointers. Also, the arguments, which might have side effects, are evaluated twice, once for the test and once to produce the value.

Given the definition

```
#define tempfile(dir)    #dir "%s"
```

the macro call `tempfile(/usr/tmp)` yields

```
"/usr/tmp" "%s"
```

which will subsequently be catenated into a single string. After

```
#define cat(x, y)        x ## y
```

the call `cat(var, 123)` yields `var123`. However, the call `cat(cat(1,2),3)` is undefined: the presence of `##` prevents the arguments of the outer call from being expanded. Thus it produces the token string

```
cat  (  1  ,  2  )3
```

and `)3` (the catenation of the last token of the first argument with the first token of the second) is not a legal token. If a second level of macro definition is introduced,

```
#define xcat(x, y)       cat(x,y)
```

things work more smoothly; `xcat(xcat(1,2), 3)` does produce `123`, because the expansion of `xcat` itself does not involve the `##` operator.

Likewise, `ABSDIFF(ABSDIFF(a,b),c)` produces the expected, fully-expanded result.

A.12.4 File Inclusion

A control line of the form

```
# include <filename>
```

causes the replacement of that line by the entire contents of the file *filename*. The characters in the name *filename* must not include > or newline, and the effect is undefined if it contains any of `"`, `'`, `\`, or `/*`. The named file is searched for in a sequence of implementation-defined places.

Similarly, a control line of the form

```
# include "filename"
```

searches first in association with the original source file (a deliberately implementation-dependent phrase), and if that search fails, then as in the first form. The effect of using `'`, `\`, or `/*` in the filename remains undefined, but > is permitted.

type-qualifier declaration-specifiers$_{opt}$

storage-class specifier: one of
`auto register static extern typedef`

type specifier: one of
`void char short int long float double signed`
`unsigned` *struct-or-union-specifier enum-specifier typedef-name*

type-qualifier: one of
`const volatile`

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ `{` *struct-declaration-list* `}`
struct-or-union identifier

struct-or-union: one of
`struct union`

struct-declaration-list:
struct declaration
struct-declaration-list struct declaration

init-declarator-list:
init-declarator
init-declarator-list`,` *init-declarator*

init-declarator:
declarator
declarator = *initializer*

struct-declaration:
specifier-qualifier-list struct-declarator-list`;`

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct-declarator
struct-declarator-list `,` *struct-declarator*

struct-declarator:
declarator
declarator$_{opt}$ `:` *constant-expression*

enum-specifier:
`enum` *identifier*$_{opt}$ `{` *enumerator-list* `}`
`enum` *identifier*

enumerator-list:
enumerator
enumerator-list `,` *enumerator*

Appendix B - Standard Library

This appendix is a summary of the library defined by the ANSI standard. The standard library is not part of the C language proper, but an environment that supports standard C will provide the function declarations and type and macro definitions of this library. We have omitted a few functions that are of limited utility or easily synthesized from others; we have omitted multi-byte characters; and we have omitted discussion of locale issues; that is, properties that depend on local language, nationality, or culture.

The functions, types and macros of the standard library are declared in standard *headers*:

```
<assert.h>  <float.h>   <math.h>    <stdarg.h>  <stdlib.h>
<ctype.h>   <limits.h>  <setjmp.h>  <stddef.h>  <string.h>
<errno.h>   <locale.h>  <signal.h>  <stdio.h>   <time.h>
```

A header can be accessed by

```
 "sb.1">B.1 Input and Output: <stdio.h> The input and output functions,
types, and macros defined in <stdio.h> represent nearly one third of the
library.
```

A *stream* is a source or destination of data that may be associated with a disk or other peripheral. The library supports text streams and binary streams, although on some systems, notably UNIX, these are identical. A text stream is a sequence of lines; each line has zero or more characters and is terminated by `'\n'`. An environment may need to convert a text stream to or from some other representation (such as mapping `'\n'` to carriage return and linefeed). A binary stream is a sequence of unprocessed bytes that record internal data, with the property that if it is written, then read back on the same system it will compare equal.

A stream is connected to a file or device by *opening* it; the connection is broken by *closing* the stream. Opening a file returns a pointer to an object of type `FILE`, which records whatever information is necessary to control the stream. We will use ``file pointer'' and ``stream'' interchangeably when there is no ambiguity.

When a program begins execution, the three streams `stdin`, `stdout`, and `stderr` are already open.

B.1.1 File Operations

The following functions deal with operations on files. The type `size_t` is the unsigned integral type produced by the `sizeof` operator.

`FILE *fopen(const char *filename, const char *mode)`
`fopen` opens the named file, and returns a stream, or `NULL` if the attempt fails. Legal values for `mode` include:

`"r"`	open text file for reading
`"w"`	create text file for writing; discard previous contents if any
`"a"`	append; open or create text file for writing at end of file
`"r+"`	open text file for update (i.e., reading and writing)
`"w+"`	create text file for update, discard previous contents if any
`"a+"`	append; open or create text file for update, writing at end

Update mode permits reading and writing the same file; `fflush` or a file-positioning function must be called between a read and a write or vice versa. If the mode includes `b` after the initial letter, as in `"rb"` or `"w+b"`, that indicates a binary file. Filenames are limited to `FILENAME_MAX` characters. At most `FOPEN_MAX` files may be open at once.

`FILE *freopen(const char *filename, const char *mode, FILE *stream)`

  - ♦ `#`, which specifies an alternate output form. For `o`, the first digit will become zero. For `x` or `X`, `0x` or `0X` will be prefixed to a non-zero result. For `e`, `E`, `f`, `g`, and `G`, the output will always have a decimal point; for `g` and `G`, trailing zeros will not be removed.
- A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is `0` if the zero padding flag is present.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for `e`, `E`, or `f` conversions, or the number of significant digits for `g` or `G` conversion, or the number of digits to be printed for an integer (leading `0`s will be added to make up the necessary width).
- A length modifier `h`, `l` (letter ell), or `L`. ``h" indicates that the corresponding argument is to be printed as a `short` or `unsigned short`; ``l" indicates that the argument is a `long` or `unsigned long`, ``L" indicates that the argument is a `long double`.

Width or precision or both may be specified as `*`, in which case the value is computed by converting the next argument(s), which must be `int`.

The conversion characters and their meanings are shown in Table B.1. If the character after the `%` is not a conversion character, the behavior is undefined.

***Table B.1** Printf Conversions*

Character	Argument type; Printed As
`d,i`	`int`; signed decimal notation.
`o`	`int`; unsigned octal notation (without a leading `0`).
`x,X`	`unsigned int`; unsigned hexadecimal notation (without a leading `0x` or `0X`), using `abcdef` for `0x` or `ABCDEF` for `0X`.
`u`	`int`; unsigned decimal notation.
`c`	`int`; single character, after conversion to `unsigned char`.
`s`	`char *`; characters from the string are printed until a `'\0'` is reached or until the number of characters indicated by the precision have been printed.
`f`	`double`; decimal notation of the form `[-]`*mmm.ddd*, where the number of *d*'s is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
`e,E`	`double`; decimal notation of the form `[-]`*m.ddddd*`e+/-`*xx* or `[-]`*m.ddddd*`E+/-`*xx*, where the number of *d*'s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
`g,G`	`double`; `%e` or `%E` is used if the exponent is less than -4 or greater than or equal to the precision; otherwise `%f` is used. Trailing zeros and a trailing decimal point are not printed.
`p`	`void *`; print as a pointer (implementation-dependent representation).
`n`	`int *`; the number of characters written so far by this call to `printf` is *written into* the argument. No argument is converted.
`%`	no argument is converted; print a %


```
int printf(const char *format, ...)
```
`printf(...)` is equivalent to `fprintf(stdout, ...)`.
```
int sprintf(char *s, const char *format, ...)
```
`sprintf` is the same as `printf` except that the output is written into the string `s`, terminated with `'\0'`. `s` must be big enough to hold the result. The return count does not include the `'\0'`.
```
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *stream, const char *format, va_list arg)
```

B.2 Character Class Tests: <ctype.h>

The header `<ctype.h>` declares functions for testing characters. For each function, the argument list is an `int`, whose value must be `EOF` or representable as an `unsigned char`, and the return value is an `int`. The functions return non-zero (true) if the argument `c` satisfies the condition described, and zero if not.

`isalnum(c)`	`isalpha(c)` or `isdigit(c)` is true
`isalpha(c)`	`isupper(c)` or `islower(c)` is true
`iscntrl(c)`	control character
`isdigit(c)`	decimal digit
`isgraph(c)`	printing character except space
`islower(c)`	lower-case letter
`isprint(c)`	printing character including space
`ispunct(c)`	printing character except space or letter or digit
`isspace(c)`	space, formfeed, newline, carriage return, tab, vertical tab
`isupper(c)`	upper-case letter
`isxdigit(c)`	hexadecimal digit

In the seven-bit ASCII character set, the printing characters are `0x20 (' ')` to `0x7E ('~')`; the control characters are 0 `NUL` to `0x1F` (US), and `0x7F` (DEL).

In addition, there are two functions that convert the case of letters:

`int tolower(c)` convert `c` to lower case
`int toupper(c)` convert `c` to upper case

If `c` is an upper-case letter, `tolower(c)` returns the corresponding lower-case letter, `toupper(c)` returns the corresponding upper-case letter; otherwise it returns `c`.

B.3 String Functions: <string.h>

There are two groups of string functions defined in the header `<string.h>`. The first have names beginning with `str`; the second have names beginning with `mem`. Except for `memmove`, the behavior is undefined if copying takes place between overlapping objects. Comparison functions treat arguments as `unsigned char` arrays.

In the following table, variables `s` and `t` are of type `char *`; `cs` and `ct` are of type `const char *`; `n` is of type `size_t`; and `c` is an `int` converted to `char`.

`char *strcpy(s,ct)`	copy string `ct` to string `s`, including `'\0'`; return `s`.
`char *strncpy(s,ct,n)`	copy at most `n` characters of string `ct` to `s`; return `s`. Pad with `'\0'`s if `ct` has fewer than `n` characters.
`char *strcat(s,ct)`	concatenate string `ct` to end of string `s`; return `s`.
`char *strncat(s,ct,n)`	concatenate at most `n` characters of string `ct` to string `s`, terminate `s` with `'\0'`; return `s`.
`int strcmp(cs,ct)`	compare string `cs` to string `ct`, return <0 if `cs<ct`, 0 if `cs==ct`, or >0 if `cs>ct`.
`int strncmp(cs,ct,n)`	compare at most `n` characters of string `cs` to string `ct`; return <0 if `cs<ct`, 0 if `cs==ct`, or >0 if `cs>ct`.
`char *strchr(cs,c)`	return pointer to first occurrence of `c` in `cs` or `NULL` if not present.

DBL_MAX_EXP		maximum *n* such that FLT_RADIX^{n-1} is representable
DBL_MIN	1E-37	minimum normalized double floating-point number
DBL_MIN_EXP		minimum *n* such that 10^n is a normalized number

Back to Appendix A -- Index -- Appendix C

Compiled by tmdcjsl(skidrow8123@hotmail.com)

Back to Appendix B -- Index