THE

C

PROGRAMMING

LANGUAGE

Second Edition

Brian W. Kernighan • Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey

PRENTICE HALL, Englewood Cliffs, New Jersey 07632
Contents

Preface ix
Preface to the First Edition xi

Introduction 1

Chapter 1. A Tutorial Introduction 5
  1.1 Getting Started 5
  1.2 Variables and Arithmetic Expressions 8
  1.3 The For Statement 13
  1.4 Symbolic Constants 14
  1.5 Character Input and Output 15
  1.6 Arrays 22
  1.7 Functions 24
  1.8 Arguments—Call by Value 27
  1.9 Character Arrays 28
  1.10 External Variables and Scope 31

Chapter 2. Types, Operators, and Expressions 35
  2.1 Variable Names 35
  2.2 Data Types and Sizes 36
  2.3 Constants 37
  2.4 Declarations 40
  2.5 Arithmetic Operators 41
  2.6 Relational and Logical Operators 41
  2.7 Type Conversions 42
  2.8 Increment and Decrement Operators 46
  2.9 Bitwise Operators 48
  2.10 Assignment Operators and Expressions 50
  2.11 Conditional Expressions 51
  2.12 Precedence and Order of Evaluation 52

Chapter 3. Control Flow 55
  3.1 Statements and Blocks 55
  3.2 If-Else 55
Chapter 8. The UNIX System Interface
8.1 File Descriptors
8.2 Low Level I/O—Read and Write
8.3 Open, Creat, Close, Unlink
8.4 Random Access—Lseek
8.5 Example—An Implementation of Fopen and Getc
8.6 Example—Listing Directories
8.7 Example—A Storage Allocator

Appendix A. Reference Manual
A1 Introduction
A2 Lexical Conventions
A3 Syntax Notation
A4 Meaning of Identifiers
A5 Objects and Lvalues
A6 Conversions
A7 Expressions
A8 Declarations
A9 Statements
A10 External Declarations
A11 Scope and Linkage
A12 Preprocessing
A13 Grammar

Appendix B. Standard Library
B1 Input and Output: <stdio.h>
B2 Character Class Tests: <ctype.h>
B3 String Functions: <string.h>
B4 Mathematical Functions: <math.h>
B5 Utility Functions: <stdlib.h>
B6 Diagnostics: <assert.h>
B7 Variable Argument Lists: <stdarg.h>
B8 Non-local Jumps: <setjmp.h>
B9 Signals: <signal.h>
B10 Date and Time Functions: <time.h>
B11 Implementation-defined Limits: <limits.h> and <float.h>

Appendix C. Summary of Changes

Index
should be 1 and *argv should point at the pattern. Notice that **argv is a
pointer to an argument string, so (**argv)[0] is its first character. (An
alternate valid form would be **argv.) Because [] binds tighter than *
and ++, the parentheses are necessary; without them the expression would
be taken as **+(argv[0]). In fact, that is what we used in the inner loop,
where the task is to walk along a specific argument string. In the inner loop,
the expression **argv[0] increments the pointer argv[0]!

It is rare that one uses pointer expressions more complicated than these; in
such cases, breaking them into two or three steps will be more intuitive.

Exercise 5-10. Write the program expr, which evaluates a reverse Polish
expression from the command line, where each operator or operand is a separate
argument. For example,

    expr 2 3 4 *

evaluates $2 \times (3+4)$. □

Exercise 5-11. Modify the programs entab and detab (written as exercises in
Chapter 1) to accept a list of tab stops as arguments. Use the default tab set-
tings if there are no arguments. □

Exercise 5-12. Extend entab and detab to accept the shorthand

    entab -m +n

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

Exercise 5-13. Write the program tail, which prints the last $n$ lines of its
input. By default, $n$ is 10, let us say, but it can be changed by an optional
argument, so that

    tail -n

prints the last $n$ lines. The program should behave rationally no matter how
unreasonable the input or the value of $n$. Write the program so it makes the
best use of available storage; lines should be stored as in the sorting program of
Section 5.6, not in a two-dimensional array of fixed size. □

5.11 Pointers to Functions

In C, a function itself is not a variable, but it is possible to define pointers to
functions, which can be assigned, placed in arrays, passed to functions, returned
by functions, and so on. We will illustrate this by modifying the sorting pro-
cedure written earlier in this chapter so that if the optional argument -n is
given, it will sort the input lines numerically instead of lexicographically.

A sort often consists of three parts—a comparison that determines the
ordering of any pair of objects, an exchange that reverses their order, and a
sorting algorithm that makes comparisons and exchanges until the objects are in
order. The sorting algorithm is independent of the comparison and exchange
operations, so by passing different comparison and exchange functions to it, we
can arrange to sort by different criteria. This is the approach taken in our new
sort.

Lexicographic comparison of two lines is done by strcmp, as before; we will
also need a routine numcmp that compares two lines on the basis of numeric
value and returns the same kind of condition indication as strcmp does. These
functions are declared ahead of main and a pointer to the appropriate one is
passed to qsort. We have skimped on error processing for arguments, so as to
concentrate on the main issues.

```c
#include <stdio.h>
#include <string.h>

#define MAXLINES 5000  /* max #lines to be sorted */
char *lineptr[MAXLINES];  /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
          int (*comp)(void *, void *));
int numcmp(char *, char *);

/* sort input lines */
main(int argc, char *argv[])
{
  int nlines;  /* number of input lines read */
  int numeric = 0;  /* 1 if numeric sort */

  if (argc > 1 && strcmp(argv[1], "-n") == 0)
    numeric = 1;
  if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
    qsort((void **) lineptr, 0, nlines-1,
          (int (*)(void *, void *)) (numeric ? numcmp : strcmp));
    writelines(lineptr, nlines);
    return 0;
  } else {
    printf("input too big to sort\n");
    return 1;
  }
}
```

In the call to qsort, strcmp and numcmp are addresses of functions. Since
they are known to be functions, the & operator is not necessary, in the same way
that it is not needed before an array name.

We have written qsort so it can process any data type, not just character
strings. As indicated by the function prototype, qsort expects an array of pointers, two integers, and a function with two pointer arguments. The generic pointer type void * is used for the pointer arguments. Any pointer can be cast to void * and back again without loss of information, so we can call qsort by casting arguments to void *. The elaborate cast of the function argument casts the arguments of the comparison function. These will generally have no effect on actual representation, but assure the compiler that all is well.

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

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

The declarations should be studied with some care. The fourth parameter of qsort is

```c
int (*comp)(void *, void *)
```

which says that comp is a pointer to a function that has two void * arguments and returns an int.

The use of comp in the line

```c
if (((*comp)(v[i], v[left]) < 0)
```

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

```c
(*comp)(v[i], v[left])
```

is the call to it. The parentheses are needed so the components are correctly associated; without them,

```c
int *comp(void *, void *)    /* WRONG */
```

says that comp is a function returning a pointer to an int, which is very different.

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

```c
#include <stdlib.h>

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

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

The swap function, which exchanges two pointers, is identical to what we presented earlier in the chapter, except that the declarations are changed to void *.

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

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

A variety of other options can be added to the sorting program; some make challenging exercises.

Exercise 5-14. Modify the sort program to handle a -r flag, which indicates sorting in reverse (decreasing) order. Be sure that -r works with -n.

Exercise 5-15. Add the option -f to fold upper and lower case together, so that case distinctions are not made during sorting; for example, a and A compare equal.

Exercise 5-16. Add the -d ("directory order") option, which makes comparisons only on letters, numbers and blanks. Make sure it works in conjunction with -f.

Exercise 5-17. Add a field-handling capability, so sorting may be done on fields within lines, each field sorted according to an independent set of options. (The index for this book was sorted with -df for the index category and -n for the page numbers.)
5.12 Complicated Declarations

C is sometimes castigated for the syntax of its declarations, particularly ones that involve pointers to functions. The syntax is an attempt to make the declaration and the use agree; it works well for simple cases, but it can be confusing for the harder ones, because declarations cannot be read left to right, and because parentheses are over-used. The difference between

```c
int *f();    /* f: function returning pointer to int */
```

and

```c
int (*pf)(); /* pf: pointer to function returning int */
```

illustrates the problem: * is a prefix operator and it has lower precedence than ( ), so parentheses are necessary to force the proper association.

Although truly complicated declarations rarely arise in practice, it is important to know how to understand them, and, if necessary, how to create them. One good way to synthesize declarations is in small steps with typedef, which is discussed in Section 6.7. As an alternative, in this section we will present a pair of programs that convert from valid C to a word description and back again. The word description reads left to right.

The first, dc1, is the more complex. It converts a C declaration into a word description, as in these examples:

```c
char **argv
    argv: pointer to pointer to char
int (*daytab)[13]
    daytab: pointer to array[13] of int
int *daytab[13]
    daytab: array[13] of pointer to int
void *comp()
    comp: function returning pointer to void
void (*comp)()
    comp: pointer to function returning void
char (*(*x())[])[5]
    x: function returning pointer to array[] of pointer to function returning char
char (*(*x[3]())[])[5]
```

dc1 is based on the grammar that specifies a declarator, which is spelled out precisely in Appendix A, Section 8.5; this is a simplified form:

```
dcl:    optional *'s direct-dcl
direct-dcl: name
        (dcl)
direct-dcl()
direct-dcl[optional size]
```

In words, a dcl is a direct-dcl, perhaps preceded by *'s. A direct-dcl is a
name, or a parenthesized dcl, or a direct-dcl followed by parentheses, or a
direct-dcl followed by brackets with an optional size.

This grammar can be used to parse declarations. For instance, consider this
declarator:

\[
(*\text{pf}a[])()\
\]

\text{pf}a will be identified as a name and thus as a direct-dcl. Then \text{pf}a[] is also
a direct-dcl. Then \(*\text{pf}a[]\) is a recognized as a dcl, so \(*\text{pf}a[]\) is a direct-
dcl. Then \(*\text{pf}a[]()\) is a direct-dcl and thus a dcl. We can also illustrate
the parse with a parse tree like this (where direct-dcl has been abbreviated to
dir-dcl):

```
(   *   \text{pf}a   [ ]   )   ()
    |     |     |     |     |
   name  dir-dcl  dir-dcl  dcl
g   dir-dcl  dir-dcl
dcl
dir-dcl
```

The heart of the dcl program is a pair of functions, dcl and dirdcl, that
parse a declaration according to this grammar. Because the grammar is recur-
sively defined, the functions call each other recursively as they recognize pieces
of a declaration; the program is called a recursive-descent parser.

```c
/* dcl: parse a declarator */
void dcl(void)
{
   int ns;

   for (ns = 0; gettoken() == '\*' ; ) /* count \*'s */
      ns++;
   dirdcl();
   while (ns-- > 0)
      strcat(out, " pointer to");
}
/* dirdcl: parse a direct declarator */
void dirdcl(void)
{
    int type;

    if (tokentype == '()') { /* ( dcl ) */
        dcl();
        if (tokentype != ')
            printf("error: missing \n");
    } else if (tokentype == NAME) /* variable name */
        strcpy(name, token);
    else
        printf("error: expected name or (dcl)\n");
    while (((type=gettoken()) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, " function returning");
        else {
            strcat(out, " array");
            strcat(out, token);
            strcat(out, " of");
        }
    }
}

Since the programs are intended to be illustrative, not bullet-proof, there are significant restrictions on dcl. It can only handle a simple data type like char or int. It does not handle argument types in functions, or qualifiers like const. Spurious blanks confuse it. It doesn’t do much error recovery, so invalid declarations will also confuse it. These improvements are left as exercises.

Here are the global variables and the main routine:

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);    /* type of last token */
int tokentype;
char token[MAXTOKEN];    /* last token string */
char name[MAXTOKEN];     /* identifier name */
char datatype[MAXTOKEN]; /* data type = char, int, etc. */
char out[1000];          /* output string */
main() /* convert declaration to words */
{
    while (gettoken() != EOF) { /* 1st token on line */
        strcpy(datatype, token); /* is the datatype */
        out[0] = '\0';
        dcl(); /* parse rest of line */
        if (tokentype != '\n')
            printf("syntax error\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}

The function gettoken skips blanks and tabs, then finds the next token in
the input; a "token" is a name, a pair of parentheses, a pair of brackets perhaps
including a number, or any other single character.

int gettoken(void) /* return next token */
{
    int c, getc(void);
    void ungetc(int);
    char *p = token;

    while (((c = getc()) == ' ' || c == '\t')
      
    if (c == '(') {
        if (((c = getc()) == ')')) {
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetc(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getc()) != ']';
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getc());
        *p++ = c;
        *p = '\0';
        ungetc(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}

getc and ungetc were discussed in Chapter 4.

Going in the other direction is easier, especially if we do not worry about
generating redundant parentheses. The program undcl converts a word
description like "x is a function returning a pointer to an array of pointers to functions returning char," which we will express as

\[
x() * [] * () \text{char}
\]

to

\[
\text{char} \ast\{(\ast x())[\]]()\}
\]

The abbreviated input syntax lets us reuse the gettoken function. undcl also uses the same external variables as dcl does.

\[
\begin{align*}
\text{*/ undcl: convert word description to declaration */} \\
\text{main()} \\
\{ \\
\text{int type;} \\
\text{char temp[MAXTOKEN];} \\
\text{while (gettoken() != EOF)} \\
\text{strcpy(out, token);} \\
\text{while ((type = gettoken()) != 'n') } \\
\text{if (type == PARENS || type == BRACKETS)} \\
\text{strcat(out, token);} \\
\text{else if (type == 's')} \\
\text{sprintf(temp, "(*%s" , out);} \\
\text{strcpy(out, temp);} \\
\text{else if (type == NAME)} \\
\text{sprintf(temp, "%s %s", token, out);} \\
\text{strcpy(out, temp);} \\
\text{else} \\
\text{printf("invalid input at %s\n", token);} \\
\text{printf("%s\n", out);} \\
\text{return 0;} \\
\}
\end{align*}
\]

Exercise 5-18. Make dcl recover from input errors.

Exercise 5-19. Modify undcl so that it does not add redundant parentheses to declarations.

Exercise 5-20. Expand dcl to handle declarations with function argument types, qualifiers like const, and so on.