The C programming Language

By Brian W. Kernighan and Dennis M. Ritchie.

Published by Prentice-Hall in 1988

ISBN 0-13-110362-8 (paperback)
ISBN 0-13-110370-9

Contents

Preface

The computing world has undergone a revolution since the publication of *The C Programming Language* in 1978. Big computers are much bigger, and personal computers have capabilities that rival mainframes of a decade ago. During this time, C has changed too, although only modestly, and it has spread far beyond its origins as the language of the UNIX operating system.

The growing popularity of C, the changes in the language over the years, and the creation of compilers by groups not involved in its design, combined to demonstrate a need for a more precise and more contemporary definition of the language than the first edition of this book provided. In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce ``an unambiguous and machine-independent definition of the language C'', while still retaining its spirit. The result is the ANSI standard for C.

The standard formalizes constructions that were hinted but not described in the first edition, particularly structure assignment and enumerations. It provides a new form of function declaration that permits cross-checking of definition with use. It specifies a standard library, with an extensive set of functions for performing input and output, memory management, string manipulation, and similar tasks. It makes precise the behavior of features that were not spelled out in the original definition, and at the same time states explicitly which aspects of the language remain machine-dependent.

This Second Edition of *The C Programming Language* describes C as defined by the ANSI standard. Although we have noted the places where the language has evolved, we have chosen to write exclusively in the new form. For the most part, this makes no significant difference; the most visible change is the new form of function declaration and definition. Modern compilers already support most features of the standard.

We have tried to retain the brevity of the first edition. C is not a big language, and it is not well served by a big book. We have improved the exposition of critical features, such as pointers, that are central to C programming. We have refined the original examples, and have added new examples in several chapters. For instance, the treatment of complicated declarations is augmented by programs that convert declarations into words and vice versa. As before, all examples have been tested directly from the text, which is in machine-readable form.

Appendix A, the reference manual, is not the standard, but our attempt to convey the essentials of the

standard in a smaller space. It is meant for easy comprehension by programmers, but not as a definition for compiler writers -- that role properly belongs to the standard itself. Appendix B is a summary of the facilities of the standard library. It too is meant for reference by programmers, not implementers. Appendix C is a concise summary of the changes from the original version.

As we said in the preface to the first edition, C ``wears well as one's experience with it grows''. With a decade more experience, we still feel that way. We hope that this book will help you learn C and use it well.

We are deeply indebted to friends who helped us to produce this second edition. Jon Bently, Doug Gwyn, Doug McIlroy, Peter Nelson, and Rob Pike gave us perceptive comments on almost every page of draft manuscripts. We are grateful for careful reading by Al Aho, Dennis Allison, Joe Campbell, G.R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford, and Chris van Wyk. We also received helpful suggestions from Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo, and Peter Weinberger. Dave Prosser answered many detailed questions about the ANSI standard. We used Bjarne Stroustrup's C++ translator extensively for local testing of our programs, and Dave Kristol provided us with an ANSI C compiler for final testing. Rich Drechsler helped greatly with typesetting.

Our sincere thanks to all.

Brian W. Kernighan
Dennis M. Ritchie

Preface to the first edition

C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C is not a ``very high level'' language, nor a ``big'' one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. Production compilers also exist for several other machines, including the IBM System/370, the Honeywell 6000, and the Interdata 8/32. C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.

This book is meant to help the reader learn how to program in C. It contains a tutorial introduction to get new users started as soon as possible, separate chapters on each major feature, and a reference manual. Most of the treatment is based on reading, writing and revising examples, rather than on mere statements of rules. For the most part, the examples are complete, real programs rather than isolated fragments. All examples have been tested directly from the text, which is in machine-readable form. Besides showing how to make effective use of the language, we have also tried where possible to illustrate useful algorithms and principles of good style and sound design.

The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. Nonetheless, a novice programmer should be able to read along and pick up the language, although access to more knowledgeable colleague will help.

In our experience, C has proven to be a pleasant, expressive and versatile language for a wide variety of programs. It is easy to learn, and it wears well as on's experience with it grows. We hope that this book will help you to use it well.

The thoughtful criticisms and suggestions of many friends and colleagues have added greatly to this book and to our pleasure in writing it. In particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy Bill Roome, Bob Rosin and Larry Rosler all read multiple volumes with care. We are also indebted to Al

Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson, and Peter Weinberger for helpful comments at various stages, and to Mile Lesk and Joe Ossanna for invaluable assistance with typesetting.

Brian W. Kernighan
Dennis M. Ritchie

Introduction

C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a ``system programming language'' because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the DEC PDP-7.

BCPL and B are ``typeless'' languages. By contrast, C provides a variety of data types. The fundamental types are characters, and integers and floating point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (`if-else`), selecting one of a set of possible values (`switch`), looping with the termination test at the top (`while, for`) or at the bottom (`do`), and early loop exit (`break`).

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Local variables are typically ``automatic'', or created anew with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.

C is a relatively ``low-level'' language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions; there is no heap or garbage collection. Finally, C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization, or coroutines.

Although the absence of some of these features may seem like a grave deficiency, (``You mean I have to call a function to compare two character strings?''), keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

For many years, the definition of C was the reference manual in the first edition of *The C Programming Language*. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or ``ANSI C'', was completed in late 1988. Most of the features of the standard are already supported by modern compilers.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce warnings of new behavior.

For most programmers, the most important change is the new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a very useful addition to the language.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are now officially part of the language. Floating-point computations may now be done in single precision. The properties of arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate. Most of these changes will have only minor effects on most programmers.

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers

provides uniform access to declarations of functions in data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the ``standard I/O library'' of the UNIX system. This library was described in the first edition, and has been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to ben an extremely effective and expressive language for a wide variety of programming applications.

The book is organized as follows. Chapter 1 is a tutorial on the central part of C. The purpose is to get the reader started as quickly as possible, since we believe strongly that the way to learn a new language is to write programs in it. The tutorial does assume a working knowledge of the basic elements of programming; there is no explanation of computers, of compilation, nor of the meaning of an expression like `n=n+1`. Although we have tried where possible to show useful programming techniques, the book is not intended to be a reference work on data structures and algorithms; when forced to make a choice, we have concentrated on the language.

Chapters 2 through 6 discuss various aspects of C in more detail, and rather more formally, than does Chapter 1, although the emphasis is still on examples of complete programs, rather than isolated fragments. Chapter 2 deals with the basic data types, operators and expressions. Chapter 3 threats control flow: `if-else`, `switch`, `while`, `for`, etc. Chapter 4 covers functions and program structure - external variables, scope rules, multiple source files, and so on - and also touches on the preprocessor. Chapter 5 discusses pointers and address arithmetic. Chapter 6 covers structures and unions.

Chapter 7 describes the standard library, which provides a common interface to the operating system. This library is defined by the ANSI standard and is meant to be supported on all machines that support C, so programs that use it for input, output, and other operating system access can be moved from one system to another without change.

Chapter 8 describes an interface between C programs and the UNIX operating system, concentrating on input/output, the file system, and storage allocation. Although some of this chapter is specific to UNIX systems, programmers who use other systems should still find useful material here, including some insight into how one version of the standard library is implemented, and suggestions on portability.

Appendix A contains a language reference manual. The official statement of the syntax and semantics of the C language is the ANSI standard itself. That document, however, is intended foremost for compiler writers. The reference manual here conveys the definition of the language more concisely and without the same legalistic style. Appendix B is a summary of the standard library, again for users rather than implementers. Appendix C is a short summary of changes from the original language. In cases of doubt, however, the standard and one's own compiler remain the final authorities on the language.

Chapter 1 - A Tutorial Introduction

Let us begin with a quick introduction in C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, rules, and exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). We want to get you as quickly as possible to the point where you can write useful programs, and to do that we have to concentrate on the basics: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output. We are intentionally leaving out of this chapter features of C that are important for writing bigger programs. These include pointers, structures, most of C's rich set of operators, several control-flow statements, and the standard library.

This approach and its drawbacks. Most notable is that the complete story on any particular feature is not found here, and the tutorial, by being brief, may also be misleading. And because the examples do not use the full power of C, they are not as concise and elegant as they might be. We have tried to minimize these effects, but be warned. Another drawback is that later chapters will necessarily repeat some of this chapter. We hope that the repetition will help you more than it annoys.

In any case, experienced programmers should be able to extrapolate from the material in this chapter to their own programming needs. Beginners should supplement it by writing small, similar programs of their own. Both groups can use it as a framework on which to hang the more detailed descriptions that begin in [Chapter 2](#).

1.1 Getting Started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

Print the words
```
hello, world
```

This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print ``hello, world'' is

```
    #include <stdio.h>

    main()
    {
      printf("hello, world\n");
```

```
    }
```

Just how to run this program depends on the system you are using. As a specific example, on the UNIX operating system you must create the program in a file whose name ends in ``.c'', such as `hello.c`, then compile it with the command

```
    cc hello.c
```

If you haven't botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called `a.out`. If you run `a.out` by typing the command

```
    a.out
```

it will print

```
    hello, world
```

On other systems, the rules will be different; check with a local expert.

Now, for some explanations about the program itself. A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions in Fortran or the procedures and functions of Pascal. Our example is a function named `main`. Normally you are at liberty to give functions whatever names you like, but ``main'' is special - your program begins executing at the beginning of main. This means that every program must have a `main` somewhere.

`main` will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program,

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

tells the compiler to include information about the standard input/output library; the line appears at the beginning of many C source files. The standard library is described in [Chapter 7](#) and [Appendix B](#).

One method of communicating data between functions is for the calling function to provide a list of values, called *arguments*, to the function it calls. The parentheses after the function name surround the argument list. In this example, `main` is defined to be a function that expects no arguments, which is indicated by the empty list ( ).

```
#include <stdio.h>               include information about standard library
main()                                    define a function called main
                                       that received no argument values
{                                   statements of main are enclosed in braces
    printf("hello, world\n");            main calls library function printf
```

to print this sequence of characters
}   *\n represents the newline character*

The first C program

The statements of a function are enclosed in braces { }. The function `main` contains only one statement,

```
printf("hello, world\n");
```

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function `printf` with the argument `"hello, world\n"`. `printf` is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like `"hello, world\n"`, is called a *character string* or *string constant*. For the moment our only use of character strings will be as arguments for `printf` and other functions.

The sequence \n in the string is C notation for the *newline character*, which when printed advances the output to the left margin on the next line. If you leave out the \n (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use \n to include a newline character in the `printf` argument; if you try something like

```
printf("hello, world
");
```

the C compiler will produce an error message.

`printf` never supplies a newline character automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written

```
#include <stdio.h>

main()
{
  printf("hello, ");
  printf("world");
  printf("\n");
}
```

to produce identical output.

Notice that \n represents only a single character. An *escape sequence* like \n provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the others that C provides are \t for tab, \b for backspace, \" for the double quote and \\ for the backslash itself. There is a complete list in [Section 2.3](#).

Exercise 1-1. Run the ``hello, world'' program on your system. Experiment with leaving out parts of the program, to see what error messages you get.

Exercise 1-2. Experiment to find out what happens when `prints`'s argument string contains *c*, where *c* is some character not listed above.

1.2 Variables and Arithmetic Expressions

The next program uses the formula $^oC=(5/9)(^oF-32)$ to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

```
1     -17
20    -6
40    4
60    15
80    26
100   37
120   48
140   60
160   71
180   82
200   93
220   104
240   115
260   126
280   137
300   148
```

The program itself still consists of the definition of a single function named `main`. It is longer than the one that printed ``hello, world'', but not complicated. It introduces several new ideas, including comments, declarations, variables, arithmetic expressions, loops , and formatted output.

```
#include <stdio.h>

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

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

  fahr = lower;
```

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

The two lines

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

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

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

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

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

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

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

The size of these objects is also machine-dependent. There are also *arrays*, *structures* and *unions* of these basic types, *pointers* to them, and *functions* that return them, all of which we will meet in due course.

Computation in the temperature conversion program begins with the *assignment statements*

```
    lower = 0;
    upper = 300;
    step = 20;
```

which set the variables to their initial values. Individual statements are terminated by semicolons.

Each line of the table is computed the same way, so we use a loop that repeats once per output line; this is the purpose of the `while` loop

```
    while (fahr <= upper) {
        ...
    }
```

The `while` loop operates as follows: The condition in parentheses is tested. If it is true (`fahr` is less than or equal to `upper`), the body of the loop (the three statements enclosed in braces) is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false (`fahr` exceeds `upper`) the loop ends, and execution continues at the statement that follows the loop. There are no further statements in this program, so it terminates.

The body of a `while` can be one or more statements enclosed in braces, as in the temperature converter, or a single statement without braces, as in

```
  while (i < j)
      i = 2 * i;
```

In either case, we will always indent the statements controlled by the `while` by one tab stop (which we have shown as four spaces) so you can see at a glance which statements are inside the loop. The indentation emphasizes the logical structure of the program. Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read. We recommend writing only one statement per line, and using blanks around operators to clarify grouping. The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently.

Most of the work gets done in the body of the loop. The Celsius temperature is computed and assigned to the variable `celsius` by the statement

```
        celsius = 5 * (fahr-32) / 9;
```

The reason for multiplying by 5 and dividing by 9 instead of just multiplying by `5/9` is that in C, as in many other languages, integer division *truncates*: any fractional part is discarded. Since 5 and 9 are integers. `5/9` would be truncated to zero and so all the Celsius temperatures would be reported as zero.

This example also shows a bit more of how `printf` works. `printf` is a general-purpose output formatting function, which we will describe in detail in [Chapter 7](). Its first argument is a string of characters to be printed, with each `%` indicating where one of the other (second, third, ...) arguments is to be substituted, and in what form it is to be printed. For instance, `%d` specifies an integer argument, so the statement

```
        printf("%d\t%d\n", fahr, celsius);
```

causes the values of the two integers `fahr` and `celsius` to be printed, with a tab (`\t`) between them.

Each `%` construction in the first argument of `printf` is paired with the corresponding second argument, third argument, etc.; they must match up properly by number and type, or you will get wrong answers.

By the way, `printf` is not part of the C language; there is no input or output defined in C itself. `printf` is just a useful function from the standard library of functions that are normally accessible to C programs. The behaviour of `printf` is defined in the ANSI standard, however, so its properties should be the same with any compiler and library that conforms to the standard.

In order to concentrate on C itself, we don't talk much about input and output until [chapter 7](). In particular, we will defer formatted input until then. If you have to input numbers, read the discussion of the function `scanf` in [Section 7.4](). `scanf` is like `printf`, except that it reads input instead of writing output.

There are a couple of problems with the temperature conversion program. The simpler one is that the output isn't very pretty because the numbers are not right-justified. That's easy to fix; if we augment each `%d` in the `printf` statement with a width, the numbers printed will be right-justified in their fields. For instance, we might say

```
printf("%3d %6d\n", fahr, celsius);
```

to print the first number of each line in a field three digits wide, and the second in a field six digits wide, like this:

```
  0       -17
 20        -6
 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:

```
     0     -17.8
    20      -6.7
    40       4.4
   ...
```

Width and precision may be omitted from a specification: `%6f` says that the number is to be at least six characters wide; `%.2f` specifies two characters after the decimal point, but the width is not constrained; and `%f` merely says to print the number as floating point.

`%d`	print as decimal integer
`%6d`	print as decimal integer, at least 6 characters wide
`%f`	print as floating point
`%6f`	print as floating point, at least 6 characters wide
`%.2f`	print as floating point, 2 characters after decimal point
`%6.2f`	print as floating point, at least 6 wide and 2 after decimal point

Among others, `printf` also recognizes `%o` for octal, `%x` for hexadecimal, `%c` for character, `%s` for character string and `%%` for itself.

Exercise 1-3. Modify the temperature conversion program to print a heading above the table.

Exercise 1-4. Write a program to print the corresponding Celsius to Fahrenheit table.

1.3 The for statement

There are plenty of different ways to write a program for a particular task. Let's try a variation on the temperature converter.

```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

This produces the same answers, but it certainly looks different. One major change is the elimination of most of the variables; only `fahr` remains, and we have made it an `int`. The lower and upper limits and the step size appear only as constants in the `for` statement, itself a new construction, and the expression that computes the Celsius temperature now appears as the third argument of `printf` instead of a separate assignment statement.

This last change is an instance of a general rule - in any context where it is permissible to use the value of some type, you can use a more complicated expression of that type. Since the third argument of `printf` must be a floating-point value to match the `%6.1f`, any floating-point expression can occur here.

The `for` statement is a loop, a generalization of the `while`. If you compare it to the earlier `while`, its operation should be clear. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialization

```
    fahr = 0
```

is done once, before the loop proper is entered. The second part is the test or condition that controls the loop:

```
    fahr <= 300
```

This condition is evaluated; if it is true, the body of the loop (here a single `ptintf`) is executed. Then the increment step

```
    fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become false. As with the `while`, the body of the loop can be a single statement or a group of statements enclosed in braces. The initialization, condition and increment can be any expressions.

The choice between `while` and `for` is arbitrary, based on which seems clearer. The `for` is usually appropriate for loops in which the initialization and increment are single statements and logically related, since it is more compact than `while` and it keeps the loop control statements together in one place.

Exercise 1-5. Modify the temperature conversion program to print the table in reverse order, that is, from 300 degrees to 0.

1.4 Symbolic Constants

A final observation before we leave temperature conversion forever. It's bad practice to bury ``magic numbers'' like 300 and 20 in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way. One way to deal with magic numbers is to give them meaningful names. A `#define` line defines a *symbolic name or symbolic constant* to be a particular string of characters:

```
 #define name replacement list
```

Thereafter, any occurrence of *name* (not in quotes and not part of another name) will be replaced by the corresponding *replacement text*. The *name* has the same form as a variable name: a sequence of letters and digits that begins with a letter. The *replacement text* can be any sequence of characters; it is not limited to numbers.

```
    #include <stdio.h>

    #define LOWER  0     /* lower limit of table */
    #define UPPER  300   /* upper limit */
    #define STEP   20    /* step size */

    /* print Fahrenheit-Celsius table */
    main()
    {
        int fahr;
```

```
    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

The quantities LOWER, UPPER and STEP are symbolic constants, not variables, so they do not appear in declarations. Symbolic constant names are conventionally written in upper case so they can ber readily distinguished from lower case variable names. Notice that there is no semicolon at the end of a #define line.

1.5 Character Input and Output

We are going to consider a family of related programs for processing character data. You will find that many programs are just expanded versions of the prototypes that we discuss here.

The model of input and output supported by the standard library is very simple. Text input or output, regardless of where it originates or where it goes to, is dealt with as streams of characters. A *text stream* is a sequence of characters divided into lines; each line consists of zero or more characters followed by a newline character. It is the responsibility of the library to make each input or output stream confirm this model; the C programmer using the library need not worry about how lines are represented outside the program.

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:

```
    #include <stdio.h>

    /* copy input to output; 1st version  */
    main()
    {
        int c;

        c = getchar();
        while (c != EOF) {
            putchar(c);
            c = getchar();
        }
    }
```

The relational operator != means ``not equal to''.

What appears to be a character on the keyboard or screen is of course, like everything else, stored internally just as a bit pattern. The type char is specifically meant for storing such character data, but any integer type can be used. We used int for a subtle but important reason.

The problem is distinguishing the end of input from valid data. The solution is that getchar returns a distinctive value when there is no more input, a value that cannot be confused with any real character. This value is called EOF, for ``end of file''. We must declare c to be a type big enough to hold any value that getchar returns. We can't use char since c must be big enough to hold EOF in addition to any possible char. Therefore we use int.

EOF is an integer defined in <stdio.h>, but the specific numeric value doesn't matter as long as it is not the same as any char value. By using the symbolic constant, we are assured that nothing in the program depends on the specific numeric value.

The program for copying would be written more concisely by experienced C programmers. In C, any assignment, such as

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

is an expression and has a value, which is the value of the left hand side after the assignment. This means that a assignment can appear as part of a larger expression. If the assignment of a character to c is put inside the test part of a while loop, the copy program can be written this way:

```
    #include <stdio.h>

    /* copy input to output; 2nd version  */
```

```
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

The `while` gets a character, assigns it to `c`, and then tests whether the character was the end-of-file signal. If it was not, the body of the `while` is executed, printing the character. The `while` then repeats. When the end of the input is finally reached, the `while` terminates and so does `main`.

This version centralizes the input - there is now only one reference to `getchar` - and shrinks the program. The resulting program is more compact, and, once the idiom is mastered, easier to read. You'll see this style often. (It's possible to get carried away and create impenetrable code, however, a tendency that we will try to curb.)

The parentheses around the assignment, within the condition are necessary. The *precedence* of `!=` is higher than that of `=`, which means that in the absence of parentheses the relational test `!=` would be done before the assignment `=`. So the statement

```
c = getchar() != EOF
```

is equivalent to

```
c = (getchar() != EOF)
```

This has the undesired effect of setting `c` to 0 or 1, depending on whether or not the call of `getchar` returned end of file. (More on this in 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.

```
  #include <stdio.h>

/* count characters in input; 2nd version */
main()
{
    double nc;

    for (nc = 0; gechar() != EOF; ++nc)
        ;
    printf("%.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 legal expression, so you will get no warning.

A character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a *character constant*, although it is just another way to write a small integer. So, for example, `'A'` is a character constant; in the ASCII character set its value is 65, the internal representation of the character A. Of course, `'A'` is to be preferred over `65`: its meaning is obvious, and it is independent of a particular character set.

The escape sequences used in string constants are also legal in character constants, so `'\n'` stands for the value of the newline character, which is 10 in ASCII. You should note carefully that `'\n'` is a single character, and in expressions is just an integer; on the other hand, `'\n'` is a string constant that happens to contain only one character. The topic of strings versus characters is discussed further in [Chapter 2]().

Exercise 1-8. Write a program to count blanks, tabs, and newlines.

Exercise 1-9. Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank.

Exercise 1-10. Write a program to copy its input to its output, replacing each tab by \t, each backspace by \b, and each backslash by \\. This makes tabs and backspaces visible in an unambiguous way.

1.5.4 Word Counting

The fourth in our series of useful programs counts lines, words, and characters, with the loose definition that a word is any sequence of characters that does not contain a blank, tab or newline. This is a bare-bones version of the UNIX program wc.

```
#include <stdio.h>

#define IN   1  /* inside a word */
#define OUT  0  /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c = '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Every time the program encounters the first character of a word, it counts one more word. The variable state records whether the program is currently in a word or not; initially it is ``not in a word'', which is assigned the value OUT. We prefer the symbolic constants IN and OUT to the literal values 1 and 0 because they make the program more readable. In a program as tiny as this, it makes little difference, but in larger programs, the increase in clarity is well worth the modest extra effort to write it this way from the beginning. You'll also find that it's easier to make extensive changes in programs where magic numbers appear only as symbolic constants.

The line

```
nl = nw = nc = 0;
```

sets all three variables to zero. This is not a special case, but a consequence of the fact that an assignment is an expression with the value and assignments associated from right to left. It's as if we had written

```
nl = (nw = (nc = 0));
```

The operator || means OR, so the line

```
if (c == ' ' || c == '\n' || c = '\t')
```

says ``if c is a blank *or* c is a newline *or* c is a tab''. (Recall that the escape sequence \t is a visible representation of the tab character.) There is a corresponding operator && for AND; its precedence is just higher than ||. Expressions connected by && or || are evaluated left to right, and it is guaranteed that evaluation will stop as soon as the truth or falsehood is known. If c is a blank, there is no need to test whether it is a newline or tab, so these tests are not made. This isn't particularly important here, but is significant in more complicated situations, as we will soon see.

The example also shows an else, which specifies an alternative action if the condition part of an if statement is false. The general form is

```
if (expression)
    statement₁
else
    statement₂
```

One and only one of the two statements associated with an if-else is performed. If the *expression* is true, *statement₁* is executed; if not, *statement₂* is executed. Each *statement* can be a single statement or several in braces. In the word count program, the one after the else is an if that controls two statements in braces.

Exercise 1-11. How would you test the word count program? What kinds of input are most likely to uncover bugs if there are any?

Exercise 1-12. Write a program that prints its input one word per line.

1.6 Arrays

Let is write a program to count the number of occurrences of each digit, of white space characters (blank, tab, newline), and of all other characters. This is artificial, but it permits us to illustrate several aspects of C in one program.

There are twelve categories of input, so it is convenient to use an array to hold the number of occurrences of each digit, rather than ten individual variables. Here is one version of the program:

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

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

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

        while ((c = getchar()) != EOF)
            if (c >= '0' && c <= '9')
                ++ndigit[c-'0'];
            else if (c == ' ' || c == '\n' || c == '\t')
                ++nwhite;
            else
                ++nother;

        printf("digits =");
        for (i = 0; i < 10; ++i)
            printf(" %d", ndigit[i]);
        printf(", white space = %d, other = %d\n",
            nwhite, nother);
    }
```

The output of this program on itself is

```
    digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

The declaration

```
    int ndigit[10];
```

declares `ndigit` to be an array of 10 integers. Array subscripts always start at zero in C, so the elements are `ndigit[0]`, `ndigit[1]`, `...`, `ndigit[9]`. This is reflected in the `for` loops that initialize and print the array.

A subscript can be any integer expression, which includes integer variables like `i`, and integer constants.

This particular program relies on the properties of the character representation of the digits. For example, the test

```
    if (c >= '0' && c <= '9')
```

determines whether the character in `c` is a digit. If it is, the numeric value of that digit is

```
    c - '0'
```

This works only if `'0'`, `'1'`, `...`, `'9'` have consecutive increasing values. Fortunately, this is true for all character sets.

By definition, `chars` are just small integers, so `char` variables and constants are identical to `ints` in arithmetic expressions. This is natural and convenient; for example `c-'0'` is an integer expression with a value between 0 and 9 corresponding to the character `'0'` to `'9'` stored in `c`, and thus a valid subscript for the array `ndigit`.

The decision as to whether a character is a digit, white space, or something else is made with the sequence

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

The pattern

```
if (condition₁)
    statement₁
else if (condition₂)
    statement₂
    ...
    ...
else
    statementₙ
```

occurs frequently in programs as a way to express a multi-way decision. The *conditions* are evaluated in order from the top until some *condition* is satisfied; at that point the corresponding *statement* part is executed, and the entire construction is finished. (Any *statement* can be several statements enclosed in braces.) If none of the conditions is satisfied, the *statement* after the final `else` is executed if it is present. If the final `else` and *statement* are omitted, as in the word count program, no action takes place. There can be any number of

```
else if(condition)
   statement
```

groups between the initial `if` and the final `else`.

As a matter of style, it is advisable to format this construction as we have shown; if each `if` were indented past the previous `else`, a long sequence of decisions would march off the right side of the page.

The `switch` statement, to be discussed in [Chapter 4](), provides another way to write a multi-way branch that is particulary suitable when the condition is whether some integer or character expression matches one of a set of constants. For contrast, we will present a `switch` version of this program in [Section 3.4]().

Exercise 1-13. Write a program to print a histogram of the lengths of words in its input. It is easy to draw the histogram with the bars horizontal; a vertical orientation is more challenging.

Exercise 1-14. Write a program to print a histogram of the frequencies of different characters in its input.

1.7 Functions

In C, a function is equivalent to a subroutine or function in Fortran, or a procedure or function in Pascal. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore *how* a job is done; knowing *what* is done is sufficient. C makes the sue of functions easy, convinient and efficient; you will often see a short function defined and called only once, just because it clarifies some piece of code.

So far we have used only functions like `printf`, `getchar` and `putchar` that have been provided for us; now it's time to write a few of our own. Since C has no exponentiation operator like the `**` of Fortran, let us illustrate the mechanics of function definition by writing a function `power(m,n)` to raise an integer `m` to a positive integer power `n`. That is, the value of `power(2,5)` is 32. This function is not a practical exponentiation routine, since it handles only positive powers of small integers, but it's good enough for illustration.(The standard library contains a function `pow(x,y)` that computes x^y.)

Here is the function `power` and a main program to exercise it, so you can see the whole structure at once.

```
#include <stdio.h>

int power(int m, int n);

/* test power function */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power:  raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int i,  p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

A function definition has this form:

```
return-type function-name(parameter declarations, if any)
{
    declarations
    statements
}
```

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. If the source program appears in several files, you may have to say more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute. For the moment, we will assume that both functions are in the same file, so whatever you have learned about running C programs will still work.

The function `power` is called twice by `main`, in the line

```
    printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Each call passes two arguments to `power`, which each time returns an integer to be formatted and printed. In an expression, `power(2,i)` is an integer just as `2` and `i` are. (Not all functions produce an integer value; we will take this up in [Chapter 4](#).)

The first line of `power` itself,

```
    int power(int base, int n)
```

declares the parameter types and names, and the type of the result that the function returns. The names used by `power` for its parameters are local to `power`, and are not visible to any other function: other routines can use the same names without conflict. This is also true of the variables `i` and `p`: the `i` in `power` is unrelated to the `i` in `main`.

We will generally use *parameter* for a variable named in the parenthesized list in a function. The terms *formal argument* and *actual argument* are sometimes used for the same distinction.

The value that `power` computes is returned to `main` by the `return`: statement. Any expression may follow return:

```
    return expression;
```

A function need not return a value; a return statement with no expression causes control, but no useful value, to be returned to the caller, as does ``falling off the end'' of a function by reaching the terminating right brace. And the calling function can ignore a value returned by a function.

You may have noticed that there is a `return` statement at the end of `main`. Since `main` is a function like any

other, it may return a value to its caller, which is in effect the environment in which the program was executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions. In the interests of simplicity, we have omitted `return` statements from our `main` functions up to this point, but we will include them hereafter, as a reminder that programs should return status to their environment.

The declaration

```
int power(int base, int n);
```

just before `main` says that `power` is a function that expects two `int` arguments and returns an `int`. This declaration, which is called a *function prototype*, has to agree with the definition and uses of `power`. It is an error if the definition of a function or any uses of it do not agree with its prototype.

parameter names need not agree. Indeed, parameter names are optional in a function prototype, so for the prototype we could have written

```
int power(int, int);
```

Well-chosen names are good documentation however, so we will often use them.

A note of history: the biggest change between ANSI C and earlier versions is how functions are declared and defined. In the original definition of C, the `power` function would have been written like this:

```
/* power:  raise base to n-th power; n >= 0 */
/*          (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

The parameters are named between the parentheses, and their types are declared before opening the left brace; undeclared parameters are taken as `int`. (The body of the function is the same as before.)

The declaration of `power` at the beginning of the program would have looked like this:

```
int power();
```

No parameter list was permitted, so the compiler could not readily check that `power` was being called correctly. Indeed, since by default `power` would have been assumed to return an `int`, the entire declaration might well have

been omitted.

The new syntax of function prototypes makes it much easier for a compiler to detect errors in the number of arguments or their types. The old style of declaration and definition still works in ANSI C, at least for a transition period, but we strongly recommend that you use the new form when you have a compiler that supports it.

Exercise 1.15. Rewrite the temperature conversion program of [Section 1.2](#) to use a function for conversion.

1.8 Arguments - Call by Value

One aspect of C functions may be unfamiliar to programmers who are used to some other languages, particulary Fortran. In C, all function arguments are passed ``by value.'' This means that the called function is given the values of its arguments in temporary variables rather than the originals. This leads to some different properties than are seen with ``call by reference'' languages like Fortran or with `var` parameters in Pascal, in which the called routine has access to the original argument, not a local copy.

Call by value is an asset, however, not a liability. It usually leads to more compact programs with fewer extraneous variables, because parameters can be treated as conveniently initialized local variables in the called routine. For example, here is a version of `power` that makes use of this property.

```
/* power:  raise base to n-th power; n >= 0; version 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

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 nicely, 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.

```c
#include <stdio.h>
#define MAXLINE 1000   /* maximum input line length */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print the longest input line */
main()
{
    int len;              /* current line length */
    int max;              /* maximum length seen so far */
    char line[MAXLINE];   /* current input line */
    char longest[MAXLINE]; /* longest line saved here */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
```

```
    if (max > 0)  /* there was a line */
        printf("%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;
    }
    s[i] = '\0';
    return i;
}

/* copy:  copy 'from' into 'to'; assume to is big enough */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

The functions getline and copy are declared at the beginning of the program, which we assume is contained in one file.

main and getline communicate through a pair of arguments and a returned value. In getline, the arguments are declared by the line

```
    int getline(char s[], int lim);
```

which specifies that the first argument, s, is an array, and the second, lim, is an integer. The purpose of supplying the size of an array in a declaration is to set aside storage. The length of an array s is not necessary in getline since its size is set in main. getline uses return to send a value back to the caller, just as the function power did. This line also declares that getline returns an int; since int is the default return type, it could be omitted.

Some functions return a useful value; others, like copy, are used only for their effect and return no value. The return type of copy is void, which states explicitly that no value is returned.

getline puts the character '\0' (the *null character*, whose value is zero) at the end of the array it is creating, to

mark the end of the string of characters. This conversion is also used by the C language: when a string constant like

```
    "hello\n"
```

appears in a C program, it is stored as an array of characters containing the characters in the string and terminated with a '\0' to mark the end.

h	e	l	l	o	\n	\0

The `%s` format specification in `printf` expects the corresponding argument to be a string represented in this form. `copy` also relies on the fact that its input argument is terminated with a '\0', and copies this character into the output.

It is worth mentioning in passing that even a program as small as this one presents some sticky design problems. For example, what should `main` do if it encounters a line which is bigger than its limit? `getline` works safely, in that it stops collecting when the array is full, even if no newline has been seen. By testing the length and the last character returned, `main` can determine whether the line was too long, and then cope as it wishes. In the interests of brevity, we have ignored this issue.

There is no way for a user of `getline` to know in advance how long an input line might be, so `getline` checks for overflow. On the other hand, the user of `copy` already knows (or can find out) how big the strings 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.

```c
#include <stdio.h>

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

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

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

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

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

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

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

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

        i = 0;
        while ((longest[i] = line[i]) != '\0')
            ++i;
    }
```

The external variables in `main`, `getline` and `copy` are defined by the first lines of the example above, which state their type and cause storage to be allocated for them. Syntactically, external definitions are just like definitions of local variables, but since they occur outside of functions, the variables are external. Before a function can use an external variable, the name of the variable must be made known to the function; the declaration is the same as before except for the added keyword `extern`.

In certain circumstances, the `extern` declaration can be omitted. If the definition of the external variable occurs in the source file before its use in a particular function, then there is no need for an `extern` declaration in the function. The `extern` declarations in `main`, `getline` and `copy` are thus redundant. In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all extern declarations.

If the program is in several source files, and a variable is defined in *file1* and used in *file2* and *file3*, then `extern` declarations are needed in *file2* and *file3* to connect the occurrences of the variable. The usual practice is to collect `extern` declarations of variables and functions in a separate file, historically called a *header*, that is included by `#include` at the front of each source file. The suffix `.h` is conventional for header names. The functions of the standard library, for example, are declared in headers like `<stdio.h>`. This topic is discussed at length in Chapter 4, and the library itself in Chapter 7 and Appendix B.

Since the specialized versions of `getline` and `copy` have no arguments, logic would suggest that their prototypes at the beginning of the file should be `getline()` and `copy()`. But for compatibility with older C programs the standard takes an empty list as an old-style declaration, and turns off all argument list checking; the word `void` must be used for an explicitly empty list. We will discuss this further in [Chapter 4](#).

You should note that we are using the words *definition* and *declaration* carefully when we refer to external variables in this section.``Definition'' refers to the place where the variable is created or assigned storage; ``declaration'' refers to places where the nature of the variable is stated but no storage is allocated.

By the way, there is a tendency to make everything in sight an `extern` variable because it appears to simplify communications - argument lists are short and variables are always there when you want them. But external variables are always there even when you don't want them. Relying too heavily on external variables is fraught with peril since it leads to programs whose data connections are not all obvious - variables can be changed in unexpected and even inadvertent ways, and the program is hard to modify. The second version of the longest-line program is inferior to the first, partly for these reasons, and partly because it destroys the generality of two useful functions by writing into them the names of the variables they manipulate.

At this point we have covered what might be called the conventional core of C. With this handful of building blocks, it's possible to write useful programs of considerable size, and it would probably be a good idea if you paused long enough to do so. These exercises suggest programs of somewhat greater complexity than the ones earlier in this chapter.

Exercise 1-20. Write a program `detab` that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every *n* columns. Should *n* be a variable or a symbolic parameter?

Exercise 1-21. Write a program `entab` that replaces strings of blanks by the minimum number of tabs and blanks to achieve the same spacing. Use the same tab stops as for `detab`. When either a tab or a single blank would suffice to reach a tab stop, which should be given preference?

Exercise 1-22. Write a program to ``fold'' long input lines into two or more shorter lines after the last non-blank character that occurs before the *n*-th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.

Exercise 1-23. Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. C comments don't nest.

Exercise 1-24. Write a program to check a C program for rudimentary syntax errors like unmatched parentheses, brackets and braces. Don't forget about quotes, both single and double, escape sequences, and comments. (This program is hard if you do it in full generality.)

Chapter 2 - Types, Operators and Expressions

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it. These building blocks are the topics of this chapter.

The ANSI standard has made many small changes and additions to basic types and expressions. There are now `signed` and `unsigned` forms of all integer types, and notations for unsigned constants and hexadecimal character constants. Floating-point operations may be done in single precision; there is also a `long` double type for extended precision. String constants may be concatenated at compile time. Enumerations have become part of the language, formalizing a feature of long standing. Objects may be declared `const`, which prevents them from being changed. The rules for automatic coercions among arithmetic types have been augmented to handle the richer set of types.

2.1 Variable Names

Although we didn't say so in [Chapter 1](#), there are some restrictions on the names of variables and symbolic constants. Names are made up of letters and digits; the first character must be a letter. The underscore ``_'' counts as a letter; it is sometimes useful for improving the readability of long variable names. Don't begin variable names with underscore, however, since library routines often use such names. Upper and lower case letters are distinct, so `x` and `X` are two different names. Traditional C practice is to use lower case for variable names, and all upper case for symbolic constants.

At least the first 31 characters of an internal name are significant. For function names and external variables, the number may be less than 31, because external names may be used by assemblers and loaders over which the language has no control. For external names, the standard guarantees uniqueness only for 6 characters and a single case. Keywords like `if`, `else`, `int`, `float`, etc., are reserved: you can't use them as variable names. They must be in lower case.

It's wise to choose variable names that are related to the purpose of the variable, and that are unlikely to

get mixed up typographically. We tend to use short names for local variables, especially loop indices, and longer names for external variables.

2.2 Data Types and Sizes

There are only a few basic data types in C:

<blockquote>

`char`   a single byte, capable of holding one character in the local character set

`int`   an integer, typically reflecting the natural size of integers on the host machine

`float` single-precision floating point

`double` double-precision floating point

</blockquote>

In addition, there are a number of qualifiers that can be applied to these basic types. `short` and `long` apply to integers:

```
short int sh;
long int counter;
```

The word `int` can be omitted in such declarations, and typically it is.

The intent is that `short` and `long` should provide different lengths of integers where practical; `int` will 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 `int`s 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 chars` 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'

where *hh* is one or more hexadecimal digits (0...9, a...f, A...F). So we might write

```
#define VTAB '\013'    /* ASCII vertical tab */
#define BELL '\007'    /* ASCII bell character */
```

or, in hexadecimal,

```
#define VTAB '\xb'    /* ASCII vertical tab */
#define BELL '\x7'    /* ASCII bell character */
```

The complete set of escape sequences is

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	\ooo	octal number
\t	horizontal tab	\xhh	hexadecimal number
\v	vertical tab		

The character constant '\0' represents the character with value zero, the null character. '\0' is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

A *constant expression* is an expression that involves only constants. Such expressions may be evaluated at during compilation rather than run-time, and accordingly may be used in any place that a constant can occur, as in

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

or

```
#define LEAP 1 /* in leap years */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

A *string constant*, or *string literal*, is a sequence of zero or more characters surrounded by double quotes, as in

```
"I am a string"
```

or

```
"" /* the empty string */
```

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; \" represents the double-quote character. String constants can be concatenated at compile time:

```
"hello, " "world"
```

is equivalent to

```
"hello, world"
```

This is useful for splitting up long strings across several source lines.

Technically, a string constant is an array of characters. The internal representation of a string has a null character '\0' at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be, but programs must scan a string completely to determine its length. The standard library function strlen(s) returns the length of its character string argument s, excluding the terminal '\0'. Here is our version:

```
/* strlen:  return length of s */
int strlen(char s[])
{
    int i;

    while (s[i] != '\0')
        ++i;
    return i;
}
```

strlen and other string functions are declared in the standard header <string.h>.

Be careful to distinguish between a character constant and a string that contains a single character: 'x' is not the same as "x". The former is an integer, used to produce the numeric value of the letter x in the machine's character set. The latter is an array of characters that contains one character (the letter x) and a '\0'.

There is one other kind of constant, the *enumeration constant*. An enumeration is a list of constant integer values, as in

```
enum boolean { NO, YES };
```

The first name in an `enum` has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
                       /* FEB = 2, MAR = 3, etc. */
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to `#define` with the advantage that the values can be generated for you. Although variables of `enum` types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, enumeration variables offer the chance of checking and so are often better than `#defines`. In addition, a debugger may be able to print values of enumeration variables in their symbolic form.

2.4 Declarations

All variables must be declared before use, although certain declarations can be made implicitly by content. A declaration specifies a type, and contains a list of one or more variables of that type, as in

```
int  lower, upper, step;
char c, line[1000];
```

Variables can be distributed among declarations in any fashion; the lists above could well be written as

```
int  lower;
int  upper;
int  step;
char c;
char line[1000];
```

The latter form takes more space, but is convenient for adding a comment to each declaration for subsequent modifications.

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer, as in

```
char   esc = '\\';
int    i = 0;
int    limit = MAXLINE+1;
float eps = 1.0e-5;
```

If the variable in question is not automatic, the initialization is done once only, conceptually before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression. External and static variables are initialized to zero by default. Automatic variables for which is no explicit initializer have undefined (i.e., garbage) values.

The qualifier `const` can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the `const` qualifier says that the elements will not be altered.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

The `const` declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

The result is implementation-defined if an attempt is made to change a `const`.

2.5 Arithmetic Operators

The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression

```
x % y
```

produces the remainder when x is divided by y, and thus is zero when y divides x exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 *are* leap years. Therefore

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d is a leap year\n", year);
else
    printf("%d is not a leap year\n", year);
```

The % operator cannot be applied to a `float` or `double`. The direction of truncation for / and the sign

of the result for `%` are machine-dependent for negative operands, as is the action taken on overflow or underflow.

The binary + and - operators have the same precedence, which is lower than the precedence of `*`, `/` and `%`, which is in turn lower than unary + and -. Arithmetic operators associate left to right.

Table 2.1 at the end of this chapter summarizes precedence and associativity for all operators.

2.6 Relational and Logical Operators

The relational operators are

```
>    >=    <    <=
```

They all have the same precedence. Just below them in precedence are the equality operators:

```
==    !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like `i < lim-1` is taken as `i < (lim-1)`, as would be expected.

More interesting are the logical operators `&&` and `||`. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. Most C programs rely on these properties. For example, here is a loop from the input function `getline` that we wrote in [Chapter 1](#):

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

Before reading a new character it is necessary to check that there is room to store it in the array `s`, so the test `i < lim-1` *must* be made first. Moreover, if this test fails, we must not go on and read another character.

Similarly, it would be unfortunate if `c` were tested against `EOF` before `getchar` is called; therefore the call and assignment must occur before the character in `c` is tested.

The precedence of `&&` is higher than that of `||`, and both are lower than relational and equality operators, so expressions like

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

need no extra parentheses. But since the precedence of != is higher than assignment, parentheses are needed in

```
(c=getchar()) != '\n'
```

to achieve the desired result of assignment to c and then comparison with '\n'.

By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false.

The unary negation operator ! converts a non-zero operand into 0, and a zero operand in 1. A common use of ! is in constructions like

```
if (!valid)
```

rather than

```
if (valid == 0)
```

It's hard to generalize about which form is better. Constructions like !valid read nicely (``if not valid''), but more complicated ones can be hard to understand.

Exercise 2-2. Write a loop equivalent to the for loop above without using && or ||.

2.7 Type Conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a ``narrower'' operand into a ``wider'' one without losing information, such as converting an integer into floating point in an expression like f + i. Expressions that don't make sense, like using a float as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

A char is just a small integer, so chars may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function atoi, which converts a string of digits into its numeric equivalent.

```
/* atoi:  convert s to integer */
int atoi(char s[])
{
    int i, n;
```

```
        n = 0;
        for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
            n = 10 * n + (s[i] - '0');
        return n;
    }
```

As we discussed in [Chapter 1](), the expression

```
    s[i] - '0'
```

gives the numeric value of the character stored in `s[i]`, because the values of `'0'`, `'1'`, etc., form a contiguous increasing sequence.

Another example of `char` to `int` conversion is the function `lower`, which maps a single character to lower case *for the ASCII character set*. If the character is not an upper case letter, `lower` returns it unchanged.

```
    /* lower:  convert c to lower case; ASCII only */
    int lower(int c)
    {
        if (c >= 'A' && c <= 'Z')
            return c + 'a' - 'A';
        else
            return c;
    }
```

This works for ASCII because corresponding upper case and lower case letters are a fixed distance apart as numeric values and each alphabet is contiguous -- there is nothing but letters between `A` and `Z`. This latter observation is not true of the EBCDIC character set, however, so this code would convert more than just letters in EBCDIC.

The standard header `<ctype.h>`, described in [Appendix B](), defines a family of functions that provide tests and conversions that are independent of character set. For example, the function `tolower` is a portable replacement for the function `lower` shown above. Similarly, the test

```
    c >= '0' && c <= '9'
```

can be replaced by

```
    isdigit(c)
```

We will use the `<ctype.h>` functions from now on.

There is one subtle point about the conversion of characters to integers. The language does not specify whether variables of type `char` are signed or unsigned quantities. When a `char` is converted to an `int`, can it ever produce a negative integer? The answer varies from machine to machine, reflecting differences in architecture. On some machines a `char` whose leftmost bit is 1 will be converted to a negative integer (``sign extension''). On others, a `char` is promoted to an int by adding zeros at the left end, and thus is always positive.

The definition of C guarantees that any character in the machine's standard printing character set will never be negative, so these characters will always be positive quantities in expressions. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others. For portability, specify `signed` or `unsigned` if non-character data is to be stored in `char` variables.

Relational expressions like `i > j` and logical expressions connected by `&&` and `||` are defined to have value 1 if true, and 0 if false. Thus the assignment

```
d = c >= '0' && c <= '9'
```

sets `d` to 1 if `c` is a digit, and 0 if not. However, functions like `isdigit` may return any non-zero value for true. In the test part of `if`, `while`, `for`, etc., ``true'' just means ``non-zero'', so this makes no difference.

Implicit arithmetic conversions work much as expected. In general, if an operator like + or * that takes two operands (a binary operator) has operands of different types, the ``lower'' type is *promoted* to the ``higher'' type before the operation proceeds. The result is of the integer type. Section 6 of Appendix A states the conversion rules precisely. If there are no `unsigned` operands, however, the following informal set of rules will suffice:

- If either operand is `long double`, convert the other to `long double`.
- Otherwise, if either operand is `double`, convert the other to `double`.
- Otherwise, if either operand is `float`, convert the other to `float`.
- Otherwise, convert `char` and `short` to `int`.
- Then, if either operand is `long`, convert the other to `long`.

Notice that `floats` in an expression are not automatically converted to `double`; this is a change from the original definition. In general, mathematical functions like those in `<math.h>` will use double precision. The main reason for using `float` is to save storage in large arrays, or, less often, to save time on machines where double-precision arithmetic is particularly expensive.

Conversion rules are more complicated when `unsigned` operands are involved. The problem is that comparisons between signed and unsigned values are machine-dependent, because they depend on the

sizes of the various integer types. For example, suppose that `int` is 16 bits and `long` is 32 bits. Then `-1L < 1U`, because `1U`, which is an `unsigned int`, is promoted to a `signed long`. But `-1L > 1UL` because `-1L` is promoted to `unsigned long` and thus appears to be a large positive number.

Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result.

A character is converted to an integer, either by sign extension or not, as described above.

Longer integers are converted to shorter ones or to `chars` by dropping the excess high-order bits. Thus in

```
int  i;
char c;

i = c;
c = i;
```

the value of `c` is unchanged. This is true whether or not sign extension is involved. Reversing the order of assignments might lose information, however.

If `x` is `float` and `i` is `int`, then `x = i` and `i = x` both cause conversions; `float` to `int` causes truncation of any fractional part. When a `double` is converted to `float`, whether the value is rounded or truncated is implementation dependent.

Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions. In the absence of a function prototype, `char` and `short` become int, and `float` becomes `double`. This is why we have declared function arguments to be `int` and `double` even when the function is called with `char` and `float`.

Finally, explicit type conversions can be forced (``coerced'') in any expression, with a unary operator called a `cast`. In the construction

 (*type name*) *expression*

the *expression* is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the *expression* were assigned to a variable of the specified type, which is then used in place of the whole construction. For example, the library routine `sqrt` expects a `double` argument, and will produce nonsense if inadvertently handled something else. (`sqrt` is declared in `<math.h>`.) So if `n` is an integer, we can use

```
sqrt((double) n)
```

to convert the value of n to double before passing it to sqrt. Note that the cast produces the *value* of n in the proper type; n itself is not altered. The cast operator has the same high precedence as other unary operators, as summarized in the table at the end of this chapter.

If arguments are declared by a function prototype, as the normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for sqrt:

```
double sqrt(double)
```

the call

```
root2 = sqrt(2)
```

coerces the integer 2 into the double value 2.0 without any need for a cast.

The standard library includes a portable implementation of a pseudo-random number generator and a function for initializing the seed; the former illustrates a cast:

```
unsigned long int next = 1;

/* rand:  return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand:  set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Exercise 2-3. Write a function htoi(s), which converts a string of hexadecimal digits (including an optional 0x or 0X) into its equivalent integer value. The allowable digits are 0 through 9, a through f, and A through F.

2.8 Increment and Decrement Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1. We have frequently used ++ to

increment variables, as in

```
if (c == '\n')
    ++nl;
```

The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++n), or postfix operators (after the variable: n++). In both cases, the effect is to increment n. But the expression ++n increments n *before* its value is used, while n++ increments n *after* its value has been used. This means that in a context where the value is being used, not just the effect, ++n and n++ are different. If n is 5, then

```
x = n++;
```

sets x to 5, but

```
x = ++n;
```

sets x to 6. In both cases, n becomes 6. The increment and decrement operators can only be applied to variables; an expression like (i+j)++ is illegal.

In a context where no value is wanted, just the incrementing effect, as in

```
if (c == '\n')
    nl++;
```

prefix and postfix are the same. But there are situations where one or the other is specifically called for. For instance, consider the function squeeze(s,c), which removes all occurrences of the character c from the string s.

```
/* squeeze:  delete all c from s */
void squeeze(char s[], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Each time a non-c occurs, it is copied into the current j position, and only then is j incremented to be ready for the next character. This is exactly equivalent to

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

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 bitwise OR operator `|` is used to turn bits on:

```
x = x | SET_ON;
```

sets to one in `x` the bits that are set to one in `SET_ON`.

The bitwise exclusive OR operator `^` sets a one in each bit position where its operands have different bits, and zero where they are the same.

One must distinguish the bitwise operators `&` and `|` from the logical operators `&&` and `||`, which imply left-to-right evaluation of a truth value. For example, if `x` is 1 and `y` is 2, then `x & y` is zero while `x && y` is one.

The shift operators `<<` and `>>` perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be non-negative. Thus `x << 2` shifts the value of `x` by two positions, filling vacated bits with zero; this is equivalent to multiplication by 4. Right shifting an `unsigned` quantity always fits the vacated bits with zero. Right shifting a signed quantity will fill with

bit signs (``arithmetic shift'') on some machines and with 0-bits (``logical shift'') on others.

The unary operator ~ yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa. For example

```
x = x & ~077
```

sets the last six bits of x to zero. Note that x & ~077 is independent of word length, and is thus preferable to, for example, x & 0177700, which assumes that x is a 16-bit quantity. The portable form involves no extra cost, since ~077 is a constant expression that can be evaluated at compile time.

As an illustration of some of the bit operators, consider the function getbits(x,p,n) that returns the (right adjusted) n-bit field of x that begins at position p. We assume that bit position 0 is at the right end and that n and p are sensible positive values. For example, getbits(x,4,3) returns the three bits in positions 4, 3 and 2, right-adjusted.

```
/* getbits:  get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

The expression x >> (p+1-n) moves the desired field to the right end of the word. ~0 is all 1-bits; shifting it left n positions with ~0<<n places zeros in the rightmost n bits; complementing that with ~ makes a mask with ones in the rightmost n bits.

Exercise 2-6. Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of y, leaving the other bits unchanged.

Exercise 2-7. Write a function invert(x,p,n) that returns x with the n bits that begin at position p inverted (i.e., 1 changed into 0 and vice versa), leaving the others unchanged.

Exercise 2-8. Write a function rightrot(x,n) that returns the value of the integer x rotated to the right by n positions.

2.10 Assignment Operators and Expressions

An expression such as

```
i = i + 2
```

in which the variable on the left side is repeated immediately on the right, can be written in the compressed form

```
i += 2
```

The operator += is called an *assignment operator.*

Most binary operators (operators like + that have a left and right operand) have a corresponding assignment operator *op=*, where *op* is one of

```
+    -    *    /    %    <<    >>    &    ^    |
```

If $expr_1$ and $expr_2$ are expressions, then

```
expr₁ op= expr₂
```

is equivalent to

```
expr₁ = (expr₁) op (expr₂)
```

except that $expr_1$ is computed only once. Notice the parentheses around $expr_2$:

```
x *= y + 1
```

means

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

rather than

```
x = x * y + 1
```

As an example, the function `bitcount` counts the number of 1-bits in its integer argument.

```
/* bitcount:  count 1 bits in x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
```

```
            b++;
    return b;
}
```

Declaring the argument `x` to be an `unsigned` ensures that when it is right-shifted, vacated bits will be filled with zeros, not sign bits, regardless of the machine the program is run on.

Quite apart from conciseness, assignment operators have the advantage that they correspond better to the way people think. We say ``add 2 to `i`'' or ``increment `i` by 2'', not ``take `i`, add 2, then put the result back in `i`''. Thus the expression `i += 2` is preferable to `i = i+2`. In addition, for a complicated expression like

```
yyval[yypv[p3+p4] + yypv[p1]] += 2
```

the assignment operator makes the code easier to understand, since the reader doesn't have to check painstakingly that two long expressions are indeed the same, or to wonder why they're not. And an assignment operator may even help a compiler to produce efficient code.

We have already seen that the assignment statement has a value and can occur in expressions; the most common example is

```
while ((c = getchar()) != EOF)
    ...
```

The other assignment operators (`+=`, `-=`, etc.) can also occur in expressions, although this is less frequent.

In all such expressions, the type of an assignment expression is the type of its left operand, and the value is the value after the assignment.

Exercise 2-9. In a two's complement number system, `x &= (x-1)` deletes the rightmost 1-bit in `x`. Explain why. Use this observation to write a faster version of `bitcount`.

2.11 Conditional Expressions

The statements

```
if (a > b)
    z = a;
else
    z = b;
```

compute in z the maximum of a and b. The *conditional expression*, written with the ternary operator ``?:'', provides an alternate way to write this and similar constructions. In the expression

```
    expr₁ ? expr₂ : expr₃
```

the expression $expr_1$ is evaluated first. If it is non-zero (true), then the expression $expr_2$ is evaluated, and that is the value of the conditional expression. Otherwise $expr_3$ is evaluated, and that is the value. Only one of $expr_2$ and $expr_3$ is evaluated. Thus to set z to the maximum of a and b,

```
    z = (a > b) ? a : b;     /* z = max(a, b) */
```

It should be noted that the conditional expression is indeed an expression, and it can be used wherever any other expression can be. If $expr_2$ and $expr_3$ are of different types, the type of the result is determined by the conversion rules discussed earlier in this chapter. For example, if f is a float and n an int, then the expression

```
    (n > 0) ? f : n
```

is of type float regardless of whether n is positive.

Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ?: is very low, just above assignment. They are advisable anyway, however, since they make the condition part of the expression easier to see.

The conditional expression often leads to succinct code. For example, this loop prints n elements of an array, 10 per line, with each column separated by one blank, and with each line (including the last) terminated by a newline.

```
    for (i = 0; i < n; i++)
        printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

A newline is printed after every tenth element, and after the n-th. All other elements are followed by one blank. This might look tricky, but it's more compact than the equivalent if-else. Another good example is

```
    printf("You have %d items%s.\n", n, n==1 ? "" : "s");
```

Exercise 2-10. Rewrite the function lower, which converts upper case letters to lower case, with a conditional expression instead of if-else.

2.12 Precedence and Order of Evaluation

Table 2.1 summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence, which is higher than that of binary `+` and `-`. The ``operator'' `()` refers to function call. The operators `->` and `.` are used to access members of structures; they will be covered in Chapter 6, along with `sizeof` (size of an object). Chapter 5 discusses `*` (indirection through a pointer) and `&` (address of an object), and Chapter 3 discusses the comma operator.

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - *` *(type)* `sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Unary `&` `+`, `-`, and `*` have higher precedence than the binary forms.

Table 2.1: Precedence and Associativity of Operators

Note that the precedence of the bitwise operators `&`, `^`, and `|` falls below `==` and `!=`. This implies that bit-testing expressions like

```
if ((x & MASK) == 0) ...
```

must be fully parenthesized to give proper results.

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are `&&`, `||`, `?:`, and `` `,'``.) For example, in a statement like

```
x = f() + g();
```

`f` may be evaluated before `g` or vice versa; thus if either `f` or `g` alters a variable on which the other depends, `x` can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n));    /* WRONG */
```

can produce different results with different compilers, depending on whether `n` is incremented before `power` is called. The solution, of course, is to write

```
++n;
printf("%d %d\n", n, power(2, n));
```

Function calls, nested assignment statements, and increment and decrement operators cause ``side effects'' - some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated. One unhappy situation is typified by the statement

```
a[i] = i++;
```

The question is whether the subscript is the old value of `i` or the new. Compilers can interpret this in different ways, and generate different answers depending on their interpretation. The standard intentionally leaves most such matters unspecified. When side effects (assignment to variables) take place within an expression is left to the discretion of the compiler, since the best order depends strongly on machine architecture. (The standard does specify that all side effects on arguments take effect before a function is called, but that would not help in the call to `printf` above.)

The moral is that writing code that depends on order of evaluation is a bad programming practice in any language. Naturally, it is necessary to know what things to avoid, but if you don't know *how* they are done on various machines, you won't be tempted to take advantage of a particular implementation.