SECOND EDITION

THE C PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Preface 6
Preface to the first edition 8
Chapter 1 - A Tutorial Introduction 9
1.1 Getting Started 9
1.2 Variables and Arithmetic Expressions 11
1.3 The for statement 16
1.4 Symbolic Constants 17
1.5 Character Input and Output 18
1.5.1 File Copying 18
1.5.2 Character Counting 20
1.5.3 Line Counting 21
1.5.4 Word Counting 22
1.6 Arrays 23
1.7 Functions 25
1.8 Arguments - Call by Value 28
1.9 Character Arrays 29
1.10 External Variables and Scope 31
Chapter 2 - Types, Operators and Expressions 35
2.1 Variable Names 35
2.2 Data Types and Sizes 35
2.3 Constants 36
2.4 Declarations 39
2.5 Arithmetic Operators 40
2.6 Relational and Logical Operators 40
2.7 Type Conversions 41
2.8 Increment and Decrement Operators 44
2.9 Bitwise Operators 46
2.10 Assignment Operators and Expressions 47
2.11 Conditional Expressions 49
2.12 Precedence and Order of Evaluation 49
Chapter 3 - Control Flow 52
3.1 Statements and Blocks 52
3.2 If-Else 52
3.3 Else-If 53
3.4 Switch 54
3.5 Loops - While and For 56
3.6 Loops - Do-While 58
3.7 Break and Continue 59
3.8 Goto and labels 60
Chapter 4 - Functions and Program Structure 62
4.1 Basics of Functions 62
4.2 Functions Returning Non-integers 65
4.3 External Variables 67
4.4 Scope Rules 72
4.5 Header Files 73
4.6 Static Variables 75
4.7 Register Variables 75
4.8 Block Structure 76
4.9 Initialization 76
4.10 Recursion 78
4.11 The C Preprocessor 79

4.11.1 File Inclusion 79
4.11.2 Macro Substitution 80
4.11.3 Conditional Inclusion 82
Chapter 5 - Pointers and Arrays 83
5.1 Pointers and Addresses 83
5.2 Pointers and Function Arguments 84
5.3 Pointers and Arrays 87
5.4 Address Arithmetic 90
5.5 Character Pointers and Functions 93
5.6 Pointer Arrays; Pointers to Pointers 96
5.7 Multi-dimensional Arrays 99
5.8 Initialization of Pointer Arrays 101
5.9 Pointers vs. Multi-dimensional Arrays 101
5.10 Command-line Arguments 102
5.11 Pointers to Functions 106
5.12 Complicated Declarations 108
Chapter 6 - Structures 114
6.1 Basics of Structures 114
6.2 Structures and Functions 116
6.3 Arrays of Structures 118
6.4 Pointers to Structures 122
6.5 Self-referential Structures 124
6.6 Table Lookup 127
6.7 Typedef 129
6.8 Unions 131
6.9 Bit-fields 132
Chapter 7 - Input and Output 135
7.1 Standard Input and Output 135
7.2 Formatted Output - printf 137
7.3 Variable-length Argument Lists 138
7.4 Formatted Input - Scanf 140
7.5 File Access 142
7.6 Error Handling - Stderr and Exit 145
7.7 Line Input and Output 146
7.8 Miscellaneous Functions 147
7.8.1 String Operations 147
7.8.2 Character Class Testing and Conversion 148
7.8.3 Ungetc 148
7.8.4 Command Execution 148
7.8.5 Storage Management 148
7.8.6 Mathematical Functions 149
7.8.7 Random Number generation 149
Chapter 8 - The UNIX System Interface 151
8.1 File Descriptors 151
8.2 Low Level I/O - Read and Write 152
8.3 Open, Creat, Close, Unlink 153
8.4 Random Access - Lseek 155
8.5 Example - An implementation of Fopen and Getc 156
8.6 Example - Listing Directories 159
8.7 Example - A Storage Allocator 163
Appendix A - Reference Manual 168
A.1 Introduction 168

A.2 Lexical Conventions 168
A.2.1 Tokens 168
A.2.2 Comments 168
A.2.3 Identifiers 168
A.2.4 Keywords 169
A.2.5 Constants 169
A.2.6 String Literals 171
A.3 Syntax Notation 171
A.4 Meaning of Identifiers 171
A.4.1 Storage Class 171
A.4.2 Basic Types 172
A.4.3 Derived types 173
A.4.4 Type Qualifiers 173
A.5 Objects and Lvalues 173
A.6 Conversions 173
A.6.1 Integral Promotion 174
A.6.2 Integral Conversions 174
A.6.3 Integer and Floating 174
A.6.4 Floating Types 174
A.6.5 Arithmetic Conversions 174
A.6.6 Pointers and Integers 175
A.6.7 Void 176
A.6.8 Pointers to Void 176
A.7 Expressions 176
A.7.1 Pointer Conversion 177
A.7.2 Primary Expressions 177
A.7.3 Postfix Expressions 177
A.7.4 Unary Operators 179
A.7.5 Casts 181
A.7.6 Multiplicative Operators 181
A.7.7 Additive Operators 182
A.7.8 Shift Operators 182
A.7.9 Relational Operators 183
A.7.10 Equality Operators 183
A.7.11 Bitwise AND Operator 183
A.7.12 Bitwise Exclusive OR Operator 184
A.7.13 Bitwise Inclusive OR Operator 184
A.7.14 Logical AND Operator 184
A.7.15 Logical OR Operator 184
A.7.16 Conditional Operator 184
A.7.17 Assignment Expressions 185
A.7.18 Comma Operator 185
A.7.19 Constant Expressions 186
A.8 Declarations 186
A.8.1 Storage Class Specifiers 187
A.8.2 Type Specifiers 188
A.8.3 Structure and Union Declarations 188
A.8.4 Enumerations 191
A.8.5 Declarators 192
A.8.6 Meaning of Declarators 193
A.8.7 Initialization 196
A.8.8 Type names 198

A.8.9 Typedef 199
A.8.10 Type Equivalence 199
A.9 Statements 199
A.9.1 Labeled Statements 200
A.9.2 Expression Statement 200
A.9.3 Compound Statement 200
A.9.4 Selection Statements 201
A.9.5 Iteration Statements 201
A.9.6 Jump statements 202
A.10 External Declarations 203
A.10.1 Function Definitions 203
A.10.2 External Declarations 204
A.11 Scope and Linkage 205
A.11.1 Lexical Scope 205
A.11.2 Linkage 206
A.12 Preprocessing 206
A.12.1 Trigraph Sequences 207
A.12.2 Line Splicing 207
A.12.3 Macro Definition and Expansion 207
A.12.4 File Inclusion 209
A.12.5 Conditional Compilation 210
A.12.6 Line Control 211
A.12.7 Error Generation 211
A.12.8 Pragmas 212
A.12.9 Null directive 212
A.12.10 Predefined names 212
A.13 Grammar 212
Appendix B - Standard Library 220
B.1 Input and Output: <stdio.h> 220
B.1.1 File Operations 220
B.1.2 Formatted Output 222
B.1.3 Formatted Input 223
B.1.4 Character Input and Output Functions 225
B.1.5 Direct Input and Output Functions 225
B.1.6 File Positioning Functions 226
B.1.7 Error Functions 226
B.2 Character Class Tests: <ctype.h> 226
B.3 String Functions: <string.h> 227
B.4 Mathematical Functions: <math.h> 228
B.5 Utility Functions: <stdlib.h> 229
B.6 Diagnostics: <assert.h> 231
B.7 Variable Argument Lists: <stdarg.h> 231
B.8 Non-local Jumps: <setjmp.h> 232
B.9 Signals: <signal.h> 232
B.10 Date and Time Functions: <time.h> 233
B.11 Implementation-defined Limits: <limits.h> and <float.h> 234
Appendix C - Summary of Changes 236

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

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 °C=(5/9)(°F-32) to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

[illegible], 2025, Bartwiladiger

```
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 `int`s, which lie between -32768 and

+32767, are common, as are 32-bit `int`s. A `float` number is typically a 32-bit quantity, with at least six significant digits and magnitude generally between about 10^{-38} and 10^{38}.

C provides several other 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, 0°F is actually about -17.8°C, 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.

2025 - Bartwodig

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);
}
```

Bartwürdiger & Bartvogt
2025, N.Schlüter

`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, $statement_1$ is executed; if not, $statement_2$ 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>

/* 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);
}
```

2025 BW

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, `char`s are just small integers, so `char` variables and constants are identical to `int`s 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;
}
```

Karimehy

2025 - Oberst-Bartvogt / Würdiger der Vereinigung der Diplomfeier

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.

```
#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.

```
#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;
}
```

Andreas [illegible]
FTDZa-ZH Barbündiger

```
/* 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:

`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

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 ints are at least 16 bits, `long`s are at least 32 bits, and `short` is no longer than `int`, which is no longer than `long`.

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

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

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

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

2.3 Constants

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

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

The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading `0` (zero) on an integer constant means octal; a leading `0x` or `0X` means hexadecimal. For example, decimal 31 can be written as `037` in octal and `0x1f` or `0x1F` in hex. Octal and

Bartwürdiger
2025, D.

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 `#define`s. 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, conceptionally 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 `char`s 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;
```

2025 ♡
Illuminierte Ehrenwürdige

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

$expr_1$? $expr_2$: $expr_3$

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.

Dr Kernighan
2025 - Bartwürdiger

Chapter 3 - Control Flow

The control-flow of a language specify the order in which computations are performed. We have already met the most common control-flow constructions in earlier examples; here we will complete the set, and be more precise about the ones discussed before.

3.1 Statements and Blocks

An expression such as `x = 0` or `i++` or `printf(...)` becomes a *statement* when it is followed by a semicolon, as in

```
x = 0;
i++;
printf(...);
```

In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal.

Braces `{` and `}` are used to group declarations and statements together into a *compound statement*, or *block*, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an `if`, `else`, `while`, or `for` are another. (Variables can be declared inside *any* block; we will talk about this in Chapter 4.) There is no semicolon after the right brace that ends a block.

3.2 If-Else

The `if-else` statement is used to express decisions. Formally the syntax is

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

where the `else` part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), $statement_1$ is executed. If it is false (*expression* is zero) and if there is an `else` part, $statement_2$ is executed instead.

Since an `if` tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
if (expression)
```

instead of

```
if (expression != 0)
```

Sometimes this is natural and clear; at other times it can be cryptic.

Because the `else` part of an `if-else` is optional,there is an ambiguity when an else if omitted from a nested `if` sequence. This is resolved by associating the `else` with the closest previous `else`-less `if`. For example, in

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

the `else` goes to the inner `if`, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
    if (a > b)
         z = a;
}
else
    z = b;
```

The ambiguity is especially pernicious in situations like this:

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else        /* WRONG */
    printf("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the `else` with the inner `if`. This kind of bug can be hard to find; it's a good idea to use braces when there are nested `if`s.

By the way, notice that there is a semicolon after `z = a` in

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

This is because grammatically, a *statement* follows the `if`, and an expression statement like ``z = a;" is always terminated by a semicolon.

3.3 Else-If

The construction

```
if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else
    statement
```

occurs so often that it is worth a brief separate discussion. This sequence of `if` statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if an *expression* is true, the *statement* associated with it is executed, and this terminates the whole chain. As always, the code for each *statement* is either a single statement, or a group of them in braces.

The last `else` part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```
else
    statement
```

can be omitted, or it may be used for error checking to catch an ``impossible'' condition.

To illustrate a three-way decision, here is a binary search function that decides if a particular value `x` occurs in the sorted array `v`. The elements of `v` must be in increasing order. The function returns the position (a number between 0 and `n-1`) if `x` occurs in `v`, and -1 if not.

Binary search first compares the input value `x` to the middle element of the array `v`. If `x` is less than the middle value, searching focuses on the lower half of the table, otherwise on the upper half. In either case, the next step is to compare `x` to the middle element of the selected half. This process of dividing the range in two continues until the value is found or the range is empty.

```
/* binsearch:  find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high)/2;
        if (x < v[mid])
            high = mid + 1;
        else if (x  > v[mid])
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;    /* no match */
}
```

The fundamental decision is whether `x` is less than, greater than, or equal to the middle element `v[mid]` at each step; this is a natural for `else-if`.

Exercise 3-1. Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside.) Write a version with only one test inside the loop and measure the difference in run-time.

3.4 Switch

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```
switch (expression) {
    case const-expr: statements
    case const-expr: statements
    default: statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled `default` is executed if none of the other cases are satisfied. A `default` is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

In Chapter 1 we wrote a program to count the occurrences of each digit, white space, and all other characters, using a sequence of `if ... else if ... else`. Here is the same program with a `switch`:

```
#include <stdio.h>

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

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c-'0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("digits =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
        nwhite, nother);
    return 0;
}
```

The `break` statement causes an immediate exit from the `switch`. Because cases serve just as labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. `break` and `return` are the most common ways to leave a `switch`. A `break` statement can also be used to force an immediate exit from `while`, `for`, and `do` loops, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a `break` to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented.

As a matter of good form, put a `break` after the last case (the `default` here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

Exercise 3-2. Write a function `escape(s,t)` that converts characters like newline and tab into visible escape sequences like `\n` and `\t` as it copies the string `t` to `s`. Use a `switch`. Write a function for the other direction as well, converting escape sequences into the real characters.

3.5 Loops - While and For

We have already encountered the `while` and `for` loops. In

```
while (expression)
    statement
```

the *expression* is evaluated. If it is non-zero, *statement* is executed and *expression* is re-evaluated. This cycle continues until *expression* becomes zero, at which point execution resumes after *statement*.

The `for` statement

```
for (expr1; expr2; expr3)
    statement
```

is equivalent to

```
expr1;
while (expr2) {
    statement
    expr3;
}
```

except for the behaviour of `continue`, which is described in Section 3.7.

Grammatically, the three components of a `for` loop are expressions. Most commonly, $expr_1$ and $expr_3$ are assignments or function calls and $expr_2$ is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If $expr_1$ or $expr_3$ is omitted, it is simply dropped from the expansion. If the test, $expr_2$, is not present, it is taken as permanently true, so

```
for (;;) {
    ...
}
```

is an ``infinite" loop, presumably to be broken by other means, such as a `break` or `return`.

Whether to use `while` or `for` is largely a matter of personal preference. For example, in

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
    ;   /* skip white space characters */
```

there is no initialization or re-initialization, so the `while` is most natural.

The `for` is preferable when there is a simple initialization and increment since it keeps the loop control statements close together and visible at the top of the loop. This is most obvious in

```
for (i = 0; i < n; i++)
    ...
```

which is the C idiom for processing the first `n` elements of an array, the analog of the Fortran DO loop or the Pascal `for`. The analogy is not perfect, however, since the index variable `i` retains its value when the loop terminates for any reason. Because the components of the `for` are arbitrary expressions, `for` loops are not restricted to arithmetic progressions. Nonetheless, it is bad style to force unrelated computations into the initialization and increment of a `for`, which are better reserved for loop control operations.

As a larger example, here is another version of `atoi` for converting a string to its numeric equivalent. This one is slightly more general than the one in Chapter 2; it copes with optional leading white space and an optional `+` or `-` sign. (Chapter 4 shows `atof`, which does the same conversion for floating-point numbers.)

The structure of the program reflects the form of the input:

skip white space, if any
get sign, if any
get integer part and convert it

Each step does its part, and leaves things in a clean state for the next. The whole process terminates on the first character that could not be part of a number.

```
#include <ctype.h>

/* atoi:  convert s to integer; version 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++)  /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')  /* skip sign */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

The standard library provides a more elaborate function `strtol` for conversion of strings to long integers; see Section 5 of Appendix B.

The advantages of keeping loop control centralized are even more obvious when there are several nested loops. The following function is a Shell sort for sorting an array of integers. The basic idea of this sorting algorithm, which was invented in 1959 by D. L. Shell, is that in early stages, far-apart elements are compared, rather than adjacent ones as in simpler interchange sorts. This tends to eliminate large amounts of disorder quickly, so later stages have less work to do. The interval between compared elements is gradually decreased to one, at which point the sort effectively becomes an adjacent interchange method.

```
/* shellsort:  sort v[0]...v[n-1] into increasing order */
void shellsort(int v[], int n)
{
```

```
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

There are three nested loops. The outermost controls the gap between compared elements, shrinking it from `n/2` by a factor of two each pass until it becomes zero. The middle loop steps along the elements. The innermost loop compares each pair of elements that is separated by `gap` and reverses any that are out of order. Since `gap` is eventually reduced to one, all elements are eventually ordered correctly. Notice how the generality of the `for` makes the outer loop fit in the same form as the others, even though it is not an arithmetic progression.

One final C operator is the comma ``,'', which most often finds use in the `for` statement. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand. Thus in a for statement, it is possible to place multiple expressions in the various parts, for example to process two indices in parallel. This is illustrated in the function `reverse(s)`, which reverses the string `s` in place.

```
#include <string.h>

/* reverse:  reverse string s in place */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

The commas that separate function arguments, variables in declarations, etc., are *not* comma operators, and do not guarantee left to right evaluation.

Comma operators should be used sparingly. The most suitable uses are for constructs strongly related to each other, as in the `for` loop in `reverse`, and in macros where a multistep computation has to be a single expression. A comma expression might also be appropriate for the exchange of elements in `reverse`, where the exchange can be thought of a single operation:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
        c = s[i], s[i] = s[j], s[j] = c;
```

Exercise 3-3. Write a function `expand(s1,s2)` that expands shorthand notations like `a-z` in the string `s1` into the equivalent complete list `abc...xyz` in `s2`. Allow for letters of either case and digits, and be prepared to handle cases like `a-b-c` and `a-z0-9` and `-a-z`. Arrange that a leading or trailing `-` is taken literally.

3.6 Loops - Do-While

As we discussed in Chapter 1, the `while` and `for` loops test the termination condition at the top. By contrast, the third loop in C, the `do-while`, tests at the bottom *after* making each pass through the loop body; the body is always executed at least once.

The syntax of the `do` is

```
do
    statement
while (expression);
```

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

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

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

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

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

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

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

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

3.7 Break and Continue

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

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

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

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

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

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

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

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

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

3.8 Goto and labels

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

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

2025 – IT21ts_WIN – Kunschwander, Cyril

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

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

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

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

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto found;
    /* didn't find any common element */
    ...
found:
    /* got one: a[i] == b[j] */
    ...
```

Code involving a `goto` can always be written without one, though perhaps at the price of some repeated tests or an extra variable. For example, the array search becomes

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* got one: a[i-1] == b[j-1] */
    ...
else
    /* didn't find any common element */
    ...
```

With a few exceptions like those cited here, code that relies on `goto` statements is generally harder to understand and to maintain than code without `goto`s. Although we are not dogmatic about the matter, it does seem that `goto` statements should be used rarely, if at all.

Chapter 4 - Functions and Program Structure

Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes.

C has been designed to make functions efficient and easy to use; C programs generally consist of many small functions rather than a few big ones. A program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. We will not go into that process here, however, since the details vary from system to system.

Function declaration and definition is the area where the ANSI standard has made the most changes to C. As we saw first in Chapter 1, it is now possible to declare the type of arguments when a function is declared. The syntax of function declaration also changes, so that declarations and definitions match. This makes it possible for a compiler to detect many more errors than it could before. Furthermore, when arguments are properly declared, appropriate type coercions are performed automatically.

The standard clarifies the rules on the scope of names; in particular, it requires that there be only one definition of each external object. Initialization is more general: automatic arrays and structures may now be initialized.

The C preprocessor has also been enhanced. New preprocessor facilities include a more complete set of conditional compilation directives, a way to create quoted strings from macro arguments, and better control over the macro expansion process.

4.1 Basics of Functions

To begin with, let us design and write a program to print each line of its input that contains a particular ``pattern" or string of characters. (This is a special case of the UNIX program `grep`.) For example, searching for the pattern of letters ``ould" in the set of lines

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

will produce the output

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

The job falls neatly into three pieces:

```
while (there's another line)
    if (the line contains the pattern)
        print it
```

Although it's certainly possible to put the code for all of this in `main`, a better way is to use the structure to advantage by making each part a separate function. Three small pieces are better to deal with than one big one, because irrelevant details can be buried in the functions, and the chance of unwanted interactions is minimized. And the pieces may even be useful in other programs.

``While there's another line" is `getline`, a function that we wrote in Chapter 1, and ``print it" is `printf`, which someone has already provided for us. This means we need only write a routine to decide whether the line contains an occurrence of the pattern.

We can solve that problem by writing a function `strindex(s,t)` that returns the position or index in the string `s` where the string `t` begins, or `-1` if `s` does not contain `t`. Because C arrays begin at position zero, indexes will be zero or positive, and so a negative value like `-1` is convenient for signaling failure. When we later need more sophisticated pattern matching, we only have to replace `strindex`; the rest of the code can remain the same. (The standard library provides a function `strstr` that is similar to `strindex`, except that it returns a pointer instead of an index.)

Given this much design, filling in the details of the program is straightforward. Here is the whole thing, so you can see how the pieces fit together. For now, the pattern to be searched for is a literal string, which is not the most general of mechanisms. We will return shortly to a discussion of how to initialize character arrays, and in Chapter 5 will show how to make the pattern a parameter that is set when the program is run. There is also a slightly different version of `getline`; you might find it instructive to compare it to the one in Chapter 1.

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

int getline(char line[], int max)
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";    /* pattern to search for */

/* find all lines matching pattern */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

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

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
```

```
    s[i] = '\0';
    return i;
}

/* strindex:  return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Each function definition has the form

```
return-type function-name(argument declarations)
{
    declarations and statements
}
```

Various parts may be absent; a minimal function is

```
dummy() {}
```

which does nothing and returns nothing. A do-nothing function like this is sometimes useful as a place holder during program development. If the return type is omitted, `int` is assumed.

A program is just a set of definitions of variables and functions. Communication between the functions is by arguments and values returned by the functions, and through external variables. The functions can occur in any order in the source file, and the source program can be split into multiple files, so long as no function is split.

The `return` statement is the mechanism for returning a value from the called function to its caller. Any expression can follow `return`:

```
return expression;
```

The *expression* will be converted to the return type of the function if necessary. Parentheses are often used around the *expression*, but they are optional.

The calling function is free to ignore the returned value. Furthermore, there need to be no expression after `return`; in that case, no value is returned to the caller. Control also returns to the caller with no value when execution ``falls off the end" of the function by reaching the closing right brace. It is not illegal, but probably a sign of trouble, if a function returns a value from one place and no value from another. In any case, if a function fails to return a value, its ``value" is certain to be garbage.

The pattern-searching program returns a status from `main`, the number of matches found. This value is available for use by the environment that called the program

The mechanics of how to compile and load a C program that resides on multiple source files vary from one system to the next. On the UNIX system, for example, the `cc` command mentioned in Chapter 1 does the job. Suppose that the three functions are stored in three files called `main.c`, `getline.c`, and `strindex.c`. Then the command

2025 Bartmñrdiger


```
cc main.c getline.c strindex.c
```

compiles the three files, placing the resulting object code in files `main.o`, `getline.o`, and `strindex.o`, then loads them all into an executable file called `a.out`. If there is an error, say in `main.c`, the file can be recompiled by itself and the result loaded with the previous object files, with the command

```
cc main.c getline.o strindex.o
```

The `cc` command uses the ``.c" versus ``.o" naming convention to distinguish source files from object files.

Exercise 4-1. Write the function `strindex(s,t)` which returns the position of the *rightmost* occurrence of `t` in `s`, or `-1` if there is none.

4.2 Functions Returning Non-integers

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

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

```
#include <ctype.h>

/* atof:  convert string s to double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++)  /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }
    return sign * val / power;
}
```

Second, and just as important, the calling routine must know that `atof` returns a non-int value. One way to ensure this is to declare `atof` explicitly in the calling routine. The declaration is shown in this primitive calculator (barely adequate for check-book balancing), which reads

one number per line, optionally preceded with a sign, and adds them up, printing the running sum after each input:

```
#include <stdio.h>

#define MAXLINE 100

/* rudimentary calculator */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

The declaration

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

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

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

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

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

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

```
double atof();
```

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

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

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

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

Notice the structure of the declarations and the return statement. The value of the expression in

```
return expression;
```

is converted to the type of the function before the return is taken. Therefore, the value of `atof`, a `double`, is converted automatically to `int` when it appears in this `return`, since the function `atoi` returns an `int`. This operation does potentionally discard information, however, so some compilers warn of it. The cast states explicitly that the operation is intended, and suppresses any warning.

Exercise 4-2. Extend `atof` to handle scientific notation of the form

```
123.45e-6
```

where a floating-point number may be followed by `e` or `E` and an optionally signed exponent.

4.3 External Variables

A C program consists of a set of external objects, which are either variables or functions. The adjective ``external" is used in contrast to ``internal", which describes the arguments and variables defined inside functions. External variables are defined outside of any function, and are thus potentionally available to many functions. Functions themselves are always external, because C does not allow functions to be defined inside other functions. By default, external variables and functions have the property that all references to them by the same name, even from functions compiled separately, are references to the same thing. (The standard calls this property *external linkage*.) In this sense, external variables are analogous to Fortran COMMON blocks or variables in the outermost block in Pascal. We will see later how to define external variables and functions that are visible only within a single source file. Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name, if the name has been declared somehow.

If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists. As pointed out in Chapter 1, however, this reasoning should be applied with some caution, for it can have a bad effect on program structure, and lead to programs with too many data connections between functions.

External variables are also useful because of their greater scope and lifetime. Automatic variables are internal to a function; they come into existence when the function is entered, and disappear when it is left. External variables, on the other hand, are permanent, so they can retain values from one function invocation to the next. Thus if two functions must share some data, yet neither calls the other, it is often most convenient if the shared data is kept in external variables rather than being passed in and out via arguments.

Let us examine this issue with a larger example. The problem is to write a calculator program that provides the operators `+`, `-`, `*` and `/`. Because it is easier to implement, the calculator will use reverse Polish notation instead of infix. (Reverse Polish notation is used by some pocket calculators, and in languages like Forth and Postscript.)

In reverse Polish notation, each operator follows its operands; an infix expression like

```
(1 - 2) * (4 + 5)
```

is entered as

```
1 2 - 4 5 + *
```

Parentheses are not needed; the notation is unambiguous as long as we know how many operands each operator expects.

The implementation is simple. Each operand is pushed onto a stack; when an operator arrives, the proper number of operands (two for binary operators) is popped, the operator is applied to them, and the result is pushed back onto the stack. In the example above, for instance, 1 and 2 are pushed, then replaced by their difference, -1. Next, 4 and 5 are pushed and then replaced by their sum, 9. The product of -1 and 9, which is -9, replaces them on the stack. The value on the top of the stack is popped and printed when the end of the input line is encountered.

The structure of the program is thus a loop that performs the proper operation on each operator and operand as it appears:

```
while (next operator or operand is not end-of-file indicator)
    if (number)
        push it
    else if (operator)
        pop operands
        do operation
        push result
    else if (newline)
        pop and print top of stack
    else
        error
```

The operation of pushing and popping a stack are trivial, but by the time error detection and recovery are added, they are long enough that it is better to put each in a separate function than to repeat the code throughout the whole program. And there should be a separate function for fetching the next input operator or operand.

The main design decision that has not yet been discussed is where the stack is, that is, which routines access it directly. On possibility is to keep it in `main`, and pass the stack and the current stack position to the routines that push and pop it. But `main` doesn't need to know about the variables that control the stack; it only does push and pop operations. So we have decided to store the stack and its associated information in external variables accessible to the `push` and `pop` functions but not to `main`.

Translating this outline into code is easy enough. If for now we think of the program as existing in one source file, it will look like this:

```
#includes
#defines

function declarations for main

main() { ... }

external variables for push and pop
```

```
void push( double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }
```

routines called by `getop`

Later we will discuss how this might be split into two or more source files.

The function `main` is a loop containing a big `switch` on the type of operator or operand; this is a more typical use of `switch` than the one shown in Section 3.4.

```
#include <stdio.h>
#include <stdlib.h>  /* for  atof() */

#define MAXOP   100  /* max size of operand or operator */
#define NUMBER  '0'  /* signal that a number was found */

int getop(char []);
void push(double);
double pop(void);

/* reverse Polish calculator */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
        }
    }
    return 0;
}
```

Because + and * are commutative operators, the order in which the popped operands are combined is irrelevant, but for - and / the left and right operand must be distinguished. In

```
push(pop() - pop());    /* WRONG */
```

the order in which the two calls of pop are evaluated is not defined. To guarantee the right order, it is necessary to pop the first value into a temporary variable as we did in main.

```
#define MAXVAL  100  /* maximum depth of val stack */

int sp = 0;          /* next free stack position */
double val[MAXVAL];  /* value stack */

/* push:  push f onto value stack */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}

/* pop:  pop and return top value from stack */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

A variable is external if it is defined outside of any function. Thus the stack and stack index that must be shared by push and pop are defined outside these functions. But main itself does not refer to the stack or stack position - the representation can be hidden.

Let us now turn to the implementation of getop, the function that fetches the next operator or operand. The task is easy. Skip blanks and tabs. If the next character is not a digit or a hexadecimal point, return it. Otherwise, collect a string of digits (which might include a decimal point), and return NUMBER, the signal that a number has been collected.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop:  get next character or numeric operand */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;       /* not a number */
    i = 0;
    if (isdigit(c))     /* collect integer part */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')       /* collect fraction part */
```

```
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

What are `getch` and `ungetch`? It is often the case that a program cannot determine that it has read enough input until it has read too much. One instance is collecting characters that make up a number: until the first non-digit is seen, the number is not complete. But then the program has read one character too far, a character that it is not prepared for.

The problem would be solved if it were possible to ``un-read'' the unwanted character. Then, every time the program reads one character too many, it could push it back on the input, so the rest of the code could behave as if it had never been read. Fortunately, it's easy to simulate un-getting a character, by writing a pair of cooperating functions. `getch` delivers the next input character to be considered; `ungetch` will return them before reading new input.

How they work together is simple. `ungetch` puts the pushed-back characters into a shared buffer -- a character array. `getch` reads from the buffer if there is anything else, and calls `getchar` if the buffer is empty. There must also be an index variable that records the position of the current character in the buffer.

Since the buffer and the index are shared by `getch` and `ungetch` and must retain their values between calls, they must be external to both routines. Thus we can write `getch`, `ungetch`, and their shared variables as:

```
#define BUFSIZE 100

char buf[BUFSIZE];      /* buffer for ungetch */
int bufp = 0;           /* next free position in buf */

int getch(void)   /* get a (possibly pushed-back) character */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)    /* push character back on input */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

The standard library includes a function `ungetch` that provides one character of pushback; we will discuss it in Chapter 7. We have used an array for the pushback, rather than a single character, to illustrate a more general approach.

Exercise 4-3. Given the basic framework, it's straightforward to extend the calculator. Add the modulus (%) operator and provisions for negative numbers.

Exercise 4-4. Add the commands to print the top elements of the stack without popping, to duplicate it, and to swap the top two elements. Add a command to clear the stack.

Exercise 4-5. Add access to library functions like `sin`, `exp`, and `pow`. See <math.h> in Appendix B, Section 4.

Exercise 4-6. Add commands for handling variables. (It's easy to provide twenty-six variables with single-letter names.) Add a variable for the most recently printed value.

Exercise 4-7. Write a routine `ungets(s)` that will push back an entire string onto the input. Should `ungets` know about `buf` and `bufp`, or should it just use `ungetch`?

Exercise 4-8. Suppose that there will never be more than one character of pushback. Modify `getch` and `ungetch` accordingly.

Exercise 4-9. Our `getch` and `ungetch` do not handle a pushed-back `EOF` correctly. Decide what their properties ought to be if an `EOF` is pushed back, then implement your design.

Exercise 4-10. An alternate organization uses `getline` to read an entire input line; this makes `getch` and `ungetch` unnecessary. Revise the calculator to use this approach.

4.4 Scope Rules

The functions and external variables that make up a C program need not all be compiled at the same time; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. Among the questions of interest are

- How are declarations written so that variables are properly declared during compilation?
- How are declarations arranged so that all the pieces will be properly connected when the program is loaded?
- How are declarations organized so there is only one copy?
- How are external variables initialized?

Let us discuss these topics by reorganizing the calculator program into several files. As a practical matter, the calculator is too small to be worth splitting, but it is a fine illustration of the issues that arise in larger programs.

The *scope* of a name is the part of the program within which the name can be used. For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared. Local variables of the same name in different functions are unrelated. The same is true of the parameters of the function, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled. For example, if `main`, `sp`, `val`, `push`, and `pop` are defined in one file, in the order shown above, that is,

```
main() { ... }

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

~~Etwas~~ Bartwürdig sofern als Frau möglich!
2025 Informatik

then the variables `sp` and `val` may be used in `push` and `pop` simply by naming them; no further declarations are needed. But these names are not visible in `main`, nor are `push` and `pop` themselves.

On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an `extern` declaration is mandatory.

It is important to distinguish between the *declaration* of an external variable and its *definition*. A declaration announces the properties of a variable (primarily its type); a definition also causes storage to be set aside. If the lines

```
int sp;
double val[MAXVAL];
```

appear outside of any function, they *define* the external variables `sp` and `val`, cause storage to be set aside, and also serve as the declarations for the rest of that source file. On the other hand, the lines

```
extern int sp;
extern double val[];
```

declare for the rest of the source file that `sp` is an `int` and that `val` is a `double` array (whose size is determined elsewhere), but they do not create the variables or reserve storage for them.

There must be only one *definition* of an external variable among all the files that make up the source program; other files may contain `extern` declarations to access it. (There may also be `extern` declarations in the file containing the definition.) Array sizes must be specified with the definition, but are optional with an `extern` declaration.

Initialization of an external variable goes only with the definition.

Although it is not a likely organization for this program, the functions `push` and `pop` could be defined in one file, and the variables `val` and `sp` defined and initialized in another. Then these definitions and declarations would be necessary to tie them together:

in file1:

```
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

in file2:

```
int sp = 0;
double val[MAXVAL];
```

Because the `extern` declarations in *file1* lie ahead of and outside the function definitions, they apply to all functions; one set of declarations suffices for all of *file1*. This same organization would also bee needed if the definition of `sp` and `val` followed their use in one file.

4.5 Header Files

Let is now consider dividing the calculator program into several source files, as it might be is each of the components were substantially bigger. The `main` function would go in one file, which we will call `main.c`; `push`, `pop`, and their variables go into a second file, `stack.c`; `getop` goes into a third, `getop.c`. Finally, `getch` and `ungetch` go into a fourth file, `getch.c`; we separate them from the others because they would come from a separately-compiled library in a realistic program.

There is one more thing to worry about - the definitions and declarations shared among files. As much as possible, we want to centralize this, so that there is only one copy to get and keep right as the program evolves. Accordingly, we will place this common material in a *header file*, `calc.h`, which will be included as necessary. (The `#include` line is described in Section 4.11.) The resulting program then looks like this:

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

There is a tradeoff between the desire that each file have access only to the information it needs for its job and the practical reality that it is harder to maintain more header files. Up to some moderate program size, it is probably best to have one header file that contains

everything that is to be shared between any two parts of the program; that is the decision we made here. For a much larger program, more organization and more headers would be needed.

4.6 Static Variables

The variables `sp` and `val` in `stack.c`, and `buf` and `bufp` in `getch.c`, are for the private use of the functions in their respective source files, and are not meant to be accessed by anything else. The `static` declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled. External `static` thus provides a way to hide names like `buf` and `bufp` in the `getch-ungetch` combination, which must be external so they can be shared, yet which should not be visible to users of `getch` and `ungetch`.

Static storage is specified by prefixing the normal declaration with the word `static`. If the two routines and the two variables are compiled in one file, as in

```
static char buf[BUFSIZE];  /* buffer for ungetch */
static int bufp = 0;       /* next free position in buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

then no other routine will be able to access `buf` and `bufp`, and those names will not conflict with the same names in other files of the same program. In the same way, the variables that `push` and `pop` use for stack manipulation can be hidden, by declaring `sp` and `val` to be `static`.

The external `static` declaration is most often used for variables, but it can be applied to functions as well. Normally, function names are global, visible to any part of the entire program. If a function is declared `static`, however, its name is invisible outside of the file in which it is declared.

The `static` declaration can also be applied to internal variables. Internal `static` variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal `static` variables provide private, permanent storage within a single function.

Exercise 4-11. Modify `getop` so that it doesn't need to use `ungetch`. Hint: use an internal `static` variable.

4.7 Register Variables

A `register` declaration advises the compiler that the variable in question will be heavily used. The idea is that `register` variables are to be placed in machine registers, which may result in smaller and faster programs. But compilers are free to ignore the advice.

The `register` declaration looks like

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

and so on. The `register` declaration can only be applied to automatic variables and to the formal parameters of a function. In this later case, it looks like

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

In practice, there are restrictions on register variables, reflecting the realities of underlying hardware. Only a few variables in each function may be kept in registers, and only certain types are allowed. Excess register declarations are harmless, however, since the word `register` is ignored for excess or disallowed declarations. And it is not possible to take the address of a register variable (a topic covered in Chapter 5), regardless of whether the variable is actually placed in a register. The specific restrictions on number and types of register variables vary from machine to machine.

4.8 Block Structure

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

```
if (n > 0) {
    int i;  /* declare a new i */

    for (i = 0; i < n; i++)
        ...
}
```

the scope of the variable `i` is the ``true" branch of the `if`; this `i` is unrelated to any `i` outside the block. An automatic variable declared and initialized in a block is initialized each time the block is entered.

Automatic variables, including formal parameters, also hide external variables and functions of the same name. Given the declarations

```
int x;
int y;

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

then within the function `f`, occurrences of `x` refer to the parameter, which is a `double`; outside `f`, they refer to the external `int`. The same is true of the variable `y`.

As a matter of style, it's best to avoid variable names that conceal names in an outer scope; the potential for confusion and error is too great.

4.9 Initialization

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

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

Scalar variables may be initialized when they are defined, by following the name with an equals sign and an expression:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

For external and static variables, the initializer must be a constant expression; the initialization is done once, conceptionally before the program begins execution. For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls. For example, the initialization of the binary search program in Section 3.3 could be written as

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

instead of

```
    int low, high, mid;

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

In effect, initialization of automatic variables are just shorthand for assignment statements. Which form to prefer is largely a matter of taste. We have generally used explicit assignments, because initializers in declarations are harder to see and further away from the point of use.

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas. For example, to initialize an array `days` with the number of days in each month:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case.

If there are fewer initializers for an array than the specified size, the others will be zero for external, static and automatic variables. It is an error to have too many initializers. There is no way to specify repetition of an initializer, nor to initialize an element in the middle of an array without supplying all the preceding values as well.

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

```
char pattern = "ould";
```

is a shorthand for the longer but equivalent

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

In this case, the array size is five (four characters plus the terminating `'\0'`).

4.10 Recursion

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

There are two solutions to this problem. On is to store the digits in an array as they are generated, then print them in the reverse order, as we did with `itoa` in section 3.6. The alternative is a recursive solution, in which `printd` first calls itself to cope with any leading digits, then prints the trailing digit. Again, this version can fail on the largest negative number.

```
#include <stdio.h>

/* printd:  print n in decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set. This in `printd(123)` the first `printd` receives the argument `n = 123`. It passes `12` to a second `printd`, which in turn passes `1` to a third. The third-level `printd` prints `1`, then returns to the second level. That `printd` prints `2`, then returns to the first level. That one prints `3` and terminates.

Another good example of recursion is quicksort, a sorting algorithm developed by C.A.R. Hoare in 1962. Given an array, one element is chosen and the others partitioned in two subsets - those less than the partition element and those greater than or equal to it. The same process is then applied recursively to the two subsets. When a subset has fewer than two elements, it doesn't need any sorting; this stops the recursion.

Our version of quicksort is not the fastest possible, but it's one of the simplest. We use the middle element of each subarray for partitioning.

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

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

We moved the swapping operation into a separate function `swap` because it occurs three times in `qsort`.

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

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

The standard library includes a version of `qsort` that can sort objects of any type.

Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent. Recursion is especially convenient for recursively defined data structures like trees, we will see a nice example in Section 6.6.

Exercise 4-12. Adapt the ideas of `printd` to write a recursive version of `itoa`; that is, convert an integer into a string by calling a recursive routine.

Exercise 4-13. Write a recursive version of the function `reverse(s)`, which reverses the string `s` in place.

4.11 The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptionally a separate first step in compilation. The two most frequently used features are `#include`, to include the contents of a file during compilation, and `#define`, to replace a token by an arbitrary sequence of characters. Other features described in this section include conditional compilation and macros with arguments.

4.11.1 File Inclusion

File inclusion makes it easy to handle collections of `#define`s and declarations (among other things). Any source line of the form

```
#include "filename"
```

or

```
#include <filename>
```

is replaced by the contents of the file *filename*. If the *filename* is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is