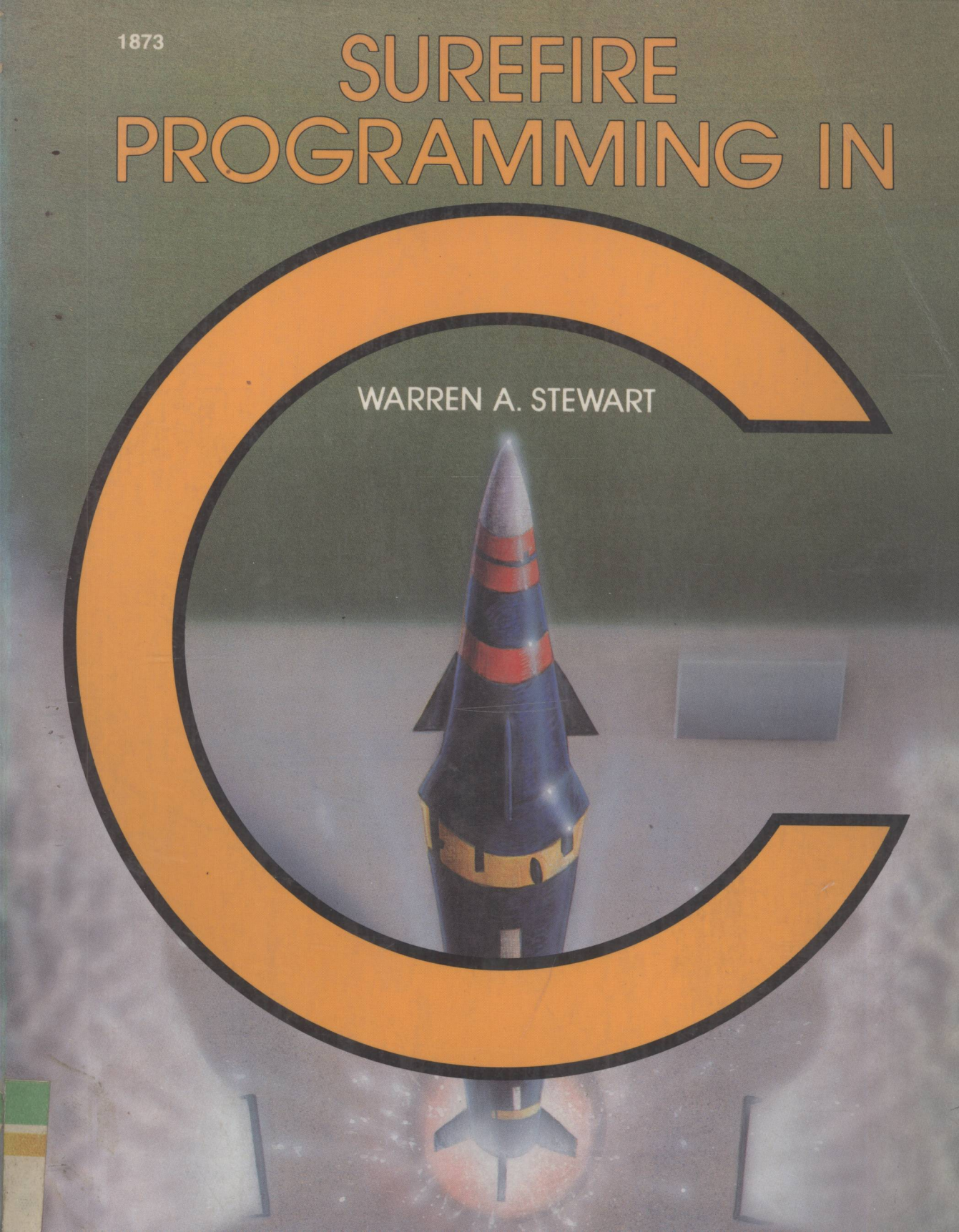
1873

SUREFIRE PROGRAMMING IN C

WARREN A. STEWART



SUREFIRE PROGRAMMING IN C

WARREN A. STEWART

A hands-on introduction to using C on CP/M, MS-DOS, *and* Unix-based microcomputers!

TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

FIRST EDITION

FIRST PRINTING

Library of Congress Cataloging in Publication Data

Stewart, Warren A.
Surefire programming in C.

Bibliography: p.
Includes index.

1. C (Computer program language) I. Title.
QA76.73.C15S74 1985 001.64'24 84-26898
ISBN 0-8306-0873-7
ISBN 0-8306-1873-2 (pbk.)

Cover illustration by Larry Selman

Preface

In the media surrounding computer technology, UNIX and C have become familiar terms. Industry analysts assess the strategic importance of UNIX for AT&T in its battle with IBM for the computer marketplace. They monitor the reactions to the announcements of a growing number of UNIX clones—operating systems such as Coherent, Zeus, Idris, Xenix, Cromix, Unos, Pcix, and Ultrix, just to name a few.

The success of UNIX in the world of minicomputers is legendary. Briefly, UNIX began with Ken Thompson's work on a PDP-7 minicomputer in 1969. In a 1978 article, McIlroy reports that there was one overriding objective of the UNIX development—to create a computing environment in which programming research could comfortably and effectively take place (*Bell System Technical Journal*, July-August, 1978). From these beginnings, UNIX became a full-blown operating system for the Digital Equipment Corporation PDP/11 series of minicomputers.

The original UNIX was written in assembly language. The C programming language was designed by Dennis Ritchie of Bell Laboratories as a language in which to rewrite UNIX. Now the vast majority of the UNIX system and its application programs are written in C. For that matter, C is the primary programming language of the UNIX environment.

Acknowledgments

Several people have made generous contributions toward my writing this book, and to all of them I am grateful. My thanks go to Walt Bilofsky, Susan Hayes, Leor Zolmon, Fredrick Richter, Rick Rump, Harvey Nero, and Mark Byrd for contributing C compilers and for helping solve various disk formatting problems encountered along the way.

Manuscript review comments by Denis Conrady and James Hunt significantly improved the book. Larry Freyou and Art Norton tested many of the programs on their computers. And Carol Lindsey's painstaking review of the manuscript and numerous editorial suggestions greatly improved the book.

Without my wife Ellen, this book would not have been possible. In addition to her editing skills and word processing prowess, her encouragement, understanding, and love sustained me throughout the project. To Ellen, my thanks and my love.

Introduction

One writes a computer program to instruct a machine to perform a task. Because machines cannot communicate in natural languages like English, it is necessary to write the instructions in a language the computer can understand.

The computer's native language is composed of ones and zeros. A valid instruction in the computer's language might be 1011101111000001. These *words* are part of a binary language, a language in which only the symbols 0 and 1 are used. This binary word may tell the computer to perform an addition operation in one of its arithmetic registers. Clearly, writing thousands of these binary instructions yields a computer program difficult for humans to understand. Higher level programming languages were developed to serve as a middle ground. The C programming language is one such language.

This book is intended for those who want to learn to write in C. It is assumed that you have access to a C compiler and a computer system on which you can experiment with the programs shown in this book. C compilers are available from a variety of manufacturers for many different computer systems. There *is* some variation in the way the language has been implemented by different manufacturers. In addition to providing detailed information on the common features these implementations share, this book presents material and exercises that

will help you discover the nonstandard features your compiler possesses. Further, guidelines are given as to how the programs in this book can be modified to properly execute with a nonstandard C compiler.

This book is a tutorial on writing programs in C; you need not be an expert programmer to use it. In some cases BASIC language programs are used for comparative purposes. Though little is lost if you are not familiar with BASIC, some familiarity with the terminology associated with programming languages is helpful.

Chapter 1 presents examples of elementary C programs and discusses the environment in which C programs are developed. The C preprocessor and some fundamental input/output operations are also discussed. Chapter 2 introduces variables, data types, constants, and arrays and indicates how each of the data types is printed.

In Chapter 3, the overall organization of a typical C program is discussed, the distinction between local variables and global variables is presented, and the concept of a C function is developed. Chapter 4 presents arithmetic and relational operators, while Chapter 5 introduces control statements.

Pointers and arrays are the topics of Chapter 6, structures are introduced in Chapter 7, and Chapter 8 is devoted to input and output (I/O) operations. Examples of programs that perform disk I/O for standard UNIX C and nonstandard C compilers are provided. Finally, Chapter 9 presents several advanced topics, including recursion, bit level operations, typedef statements, and unions.

Two appendices are also proyided. Appendix A is for readers who are not familiar with the octal and hexadecimal number systems, or scientific notation. The appendix provides an overview of these topics. Appendix B is a series of tables that have been placed at the end of the book for easy reference.

NOTATION USED IN THIS BOOK

The ellipses(...) are used to mean "and so on." For example,

$$1,2,3,\ldots,99$$

represents the first 99 positive integers.

References to other literature are cited by a bracketed indicator such as [KERN78], which refers to Kernighan and

Ritchie's definitive work on the C programming language. The list of references is found at the end of the book.

Some sections are marked with an asterisk (*). These sections can safely be skimmed or skipped on the first reading. You can return to these sections at a later point after gaining more familiarity with C. All other notation used is introduced in the text.

Contents

Chapter 1

Getting Started with C

To begin learning C, let's examine our first C program:

```
main() {
      /* A program to print welcome */

      printf("welcome");
}
```

If you are familiar with BASIC, you'll find this C program is just like the following BASIC program:

```
10 REM A PROGRAM TO PRINT WELCOME
20 PRINT "welcome"
```

Or, if you have programmed in Pascal, it is like the following Pascal program:

```
PROGRAM WELCOMEPROG (OUTPUT);
(* A PROGRAM TO PRINT WELCOME *)
BEGIN
      WRITELN('welcome')
END.
```

Because this is our first C program, let's examine it in some detail. *Main* is the name of a C *function*. Every C program has a

function called main. The parentheses following the word *main* indicate that it is a function. Functions are like black boxes of computation and logic. Data can be passed to them and they act on the data according to the instructions they contain. The main function of the welcome program is not passed any data; that is why there is nothing between the parentheses following its name.

The left brace ({) introduces the statements that define the function main. This version of main begins with a comment. All text between the symbols /* and */ is commentary for the benefit of the programmer or other human readers. Following the comment, there is one statement:

```
printf("welcome");
```

Statements are written in a free format; they are not required to begin in a particular column, and more than one statement can be written on the same line.

Printf, like main, is a function. You can tell by the parentheses that follow its name. Between printf's parentheses are the characters "welcome". This string of seven characters (the double quotation marks are not included) is data that is passed to the function printf. Printf is a function that writes the data passed to it on the terminal screen.

The semicolon following printf("welcome") is the C statement terminator. Every C statement is terminated by a semicolon. Finally, the closing right brace (}) indicates the end of the statements in the function main.

You have just examined your first C program. But, before you leave it, you should be aware that the first C program behaves a little differently than a first BASIC or Pascal program (as they are written here). The second C program points out the difference:

```
main() {
      /* This is your
            second C program */

      printf("hi");
      printf("there");
}
```

It may appear that this program will print two lines of text, but actually it prints only one line. The printf function does not

automatically move down a line each time it is used. If you want a new line, you must explicitly say so. Thus, the first program prints:

welcome

while the second program prints:

hithere

If the second program is intended to print two lines, the program would look like this:

```
main() {
        printf("hi\n");
        printf("there");
}
```

The \n is a special "escape sequence" used in C to represent a new line. C provides escape sequences for several invisible characters. A discussion of this and other C escape sequences is found in Chapter 2. Note that the comment in the second program extends across two lines. All text between /* and */ is a comment, even when the text spans two or more lines.

These programs are simple indeed; but to make them actually work on your system you must enter the program text into the system, and then *compile* and *link* the program. If you have used compilers before and are familiar with the process of compilation and linkage, skip forward to the section on the "C preprocessor" later in this chapter. If you are not familiar with the process, you should read the description that follows. In the next section, you will examine the type of environment that supports C programming.

THE C PROGRAMMING ENVIRONMENT

In the previous section you examined a couple of elementary C programs. Now, you will examine the process which transforms C language programs into executable machine code programs. The process has several distinct phases: program creation using a text editor, compilation, linkage editing, and debugging. The following sections highlight these phases of program development.

Text Editing and Compilation

The first step in writing a C program is to type the C statements into a file. A text editor is used to accomplish this task. The use of a text editor will not be discussed in this book; if

you are not already accustomed to the text editor on your system, take a little time to familiarize yourself with its use.

The file containing C statements is called a *source file* or *source code*. Source files are translated into the machine code instructions that execute in the computer. The translation from source code to machine code is performed by a series of computer programs. The first translation program is the C compiler. A C compiler's job is to translate (or compile) C language statements into assembly language statements. Assembly language is a human readable form of machine code. The assembly language statements are then processed by the assembler. The assembler translates assembly language into object modules or object code. Figure 1-1 depicts the process of C program development discussed so far.

With the production of the object code, the translation from source code to machine code is nearly complete. However, several object files are required to make one executable program. For example, if your program uses the printf function, the object code for that function must become a part of your program. Various object files must be linked together to form one executable program. Thus, the last step in the translation process is to link object files into the final machine code. This task is performed by a program called a *linkage-editor* or *linker*.

The Linker

The linker allows previously compiled functions to be used in programs. This is an important feature of the C environment. It saves time; often-used functions do not have to be repeatedly

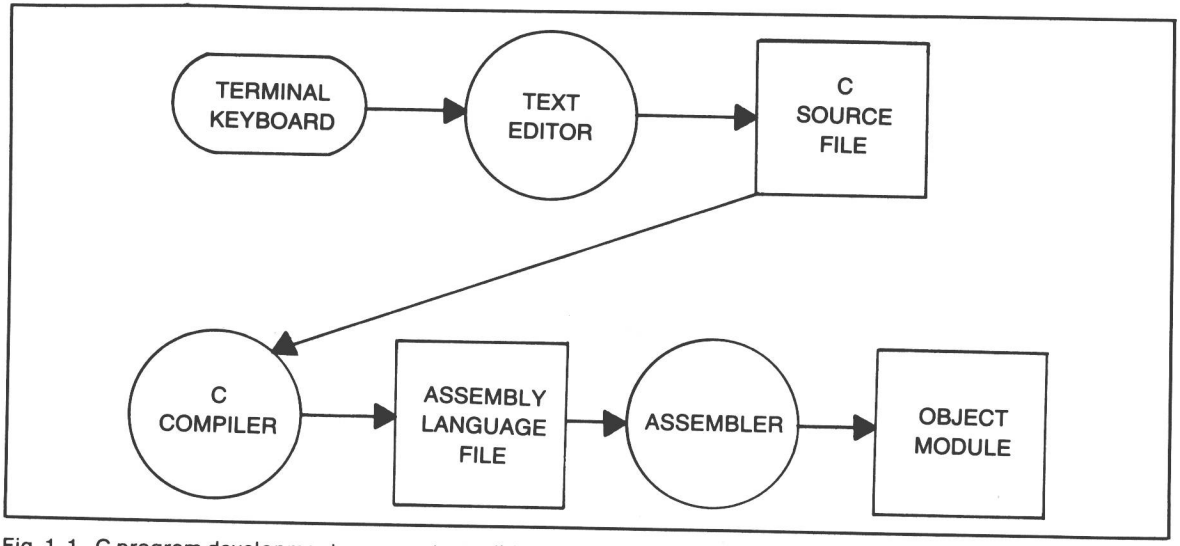


Fig. 1-1. C program development process: text editing and compilation.

recompiled. Further, compiler vendors will supply libraries of functions you can link into your programs. For example, functions such as printf, getchar, and putchar are supplied in input/output libraries and are readily available for your use. Such libraries let you create useful programs much more rapidly than if you had to write all the functions yourself.

In addition to linking vendor-supplied object code into your programs, you can use the linker to link functions you have developed into your programs. For instance, suppose you have written a C function to sort names alphabetically. It would be convenient if you could use this function in several programs without having to compile it on every occasion. You could generate the object code once, and use that code in several programs. Environments that support C generally allow this feature. This feature is called *separate compilation*. The linker is used to link separately compiled functions into one executable program.

Figure 1-2 updates Fig. 1-1 by adding the action of the linker and showing the resulting picture of the program development process.

Compile and Link Commands

The commands you issue on your system to perform compilation and linkage depend on the compiler you are using. As examples of how the command sequence appears, suppose you want to create an executable program from C source statements stored in a file called prog.c. Using the BDS C compiler, the commands would be:

```
cc prog.c
clink prog
```

The first command invokes the compiler. After the compilation is complete, the second command invokes the linker.

To perform the same process with the DeSmet C compiler on an IBM PC, the commands would be:

```
c88 prog.c
bind prog
```

Again, the compiler is invoked by the first command, and the linker is invoked by the second.

As you can see, the actual commands required to perform the compilation and linkage process vary from system to system. Refer to the user guide supplied with your compiler to determine the commands required on your system.

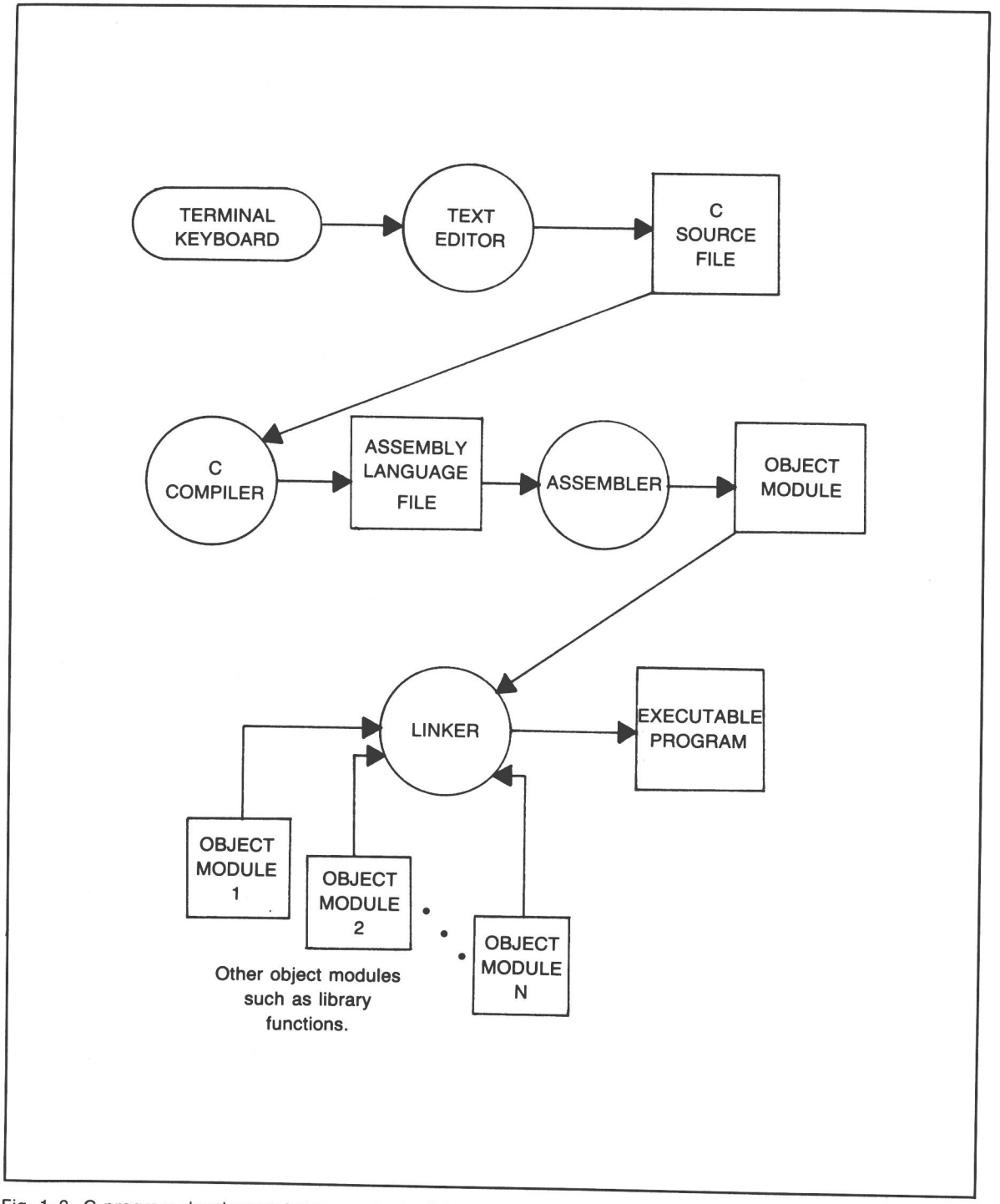


Fig. 1-2. C program development process: text editing, compilation and linkage.