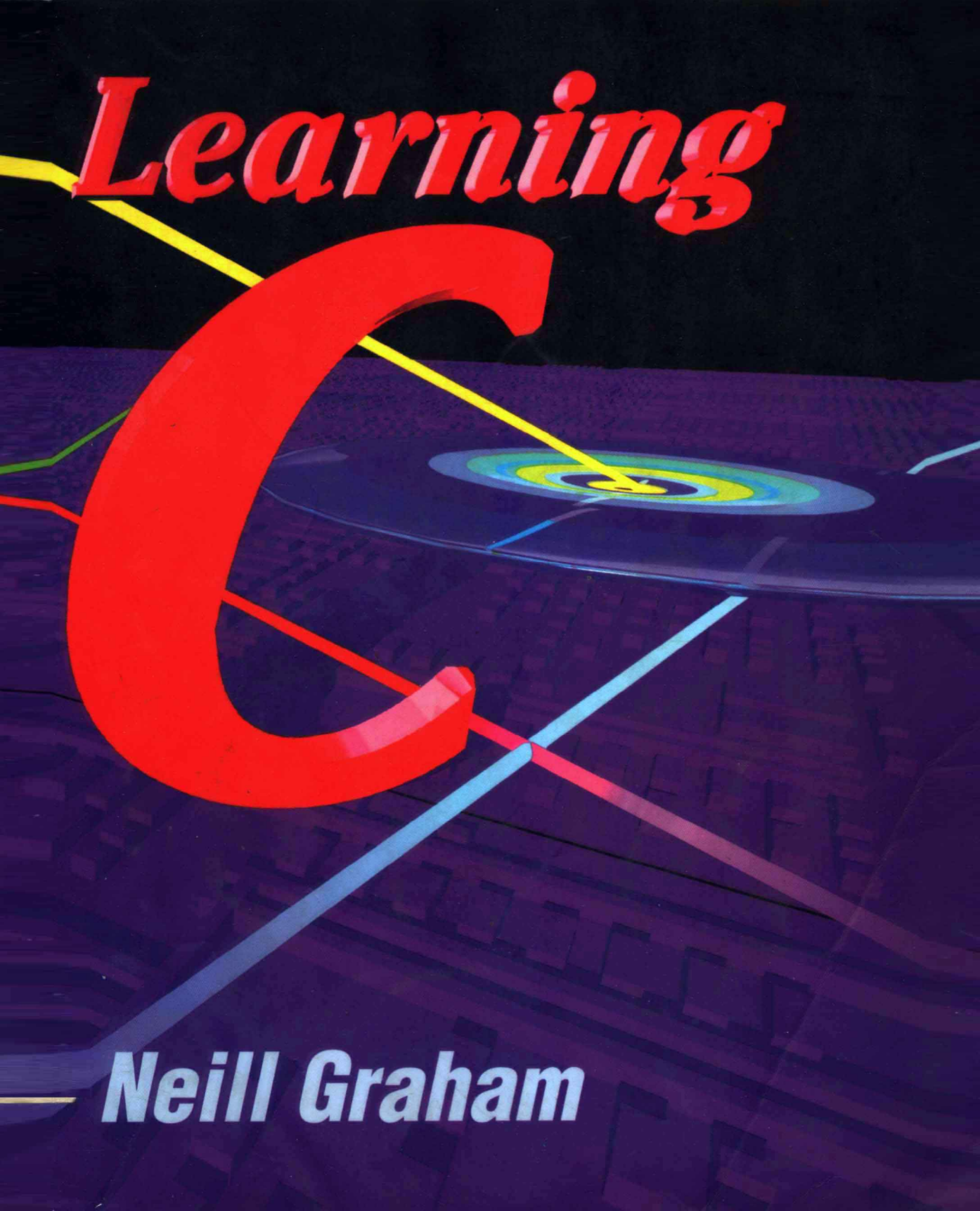
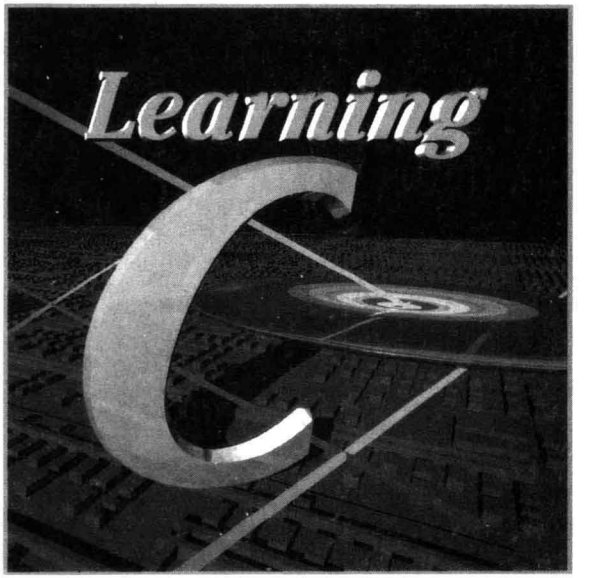
Learning C



Neill Graham


Learning
C

Preface

This book is an introduction to C for readers who are already acquainted with at least one programming language (which will probably be Pascal). The only reason for such a prerequisite is that this book focuses more on the details of C than on the elements of programming. No background is assumed in any other area of computing, such as computer science or system programming. Important computing concepts, such as variables and pointers, are reviewed briefly before being used, a practice for which we ask the indulgence of readers with more experience.

About C

C is now the most popular language for professional software development on minicomputers and microcomputers. Many programmers who formerly used assembly language or Pascal have switched to C, and some existing software products have actually been translated into C from Pascal or assembly language.

Although C provides most of the features one expects of a higher-level language, it also offers the low-level access to hardware and software that is characteristic of assembly language (and that is required for all system programming and much application programming). Some languages that offer this low-level access are so hardware specific that they can be used on only one kind of computer (assembly language is, of course, the prime example). C seems to have hit the right level of generality in that it offers the low-level access that programmers need yet can be (and has been) implemented on a wide variety of main frames, minicomputers, and microcomputers.

A recently developed ANSI (American National Standards Institute) standard for C will make it even easier to port (move) C programs from one computer to another. The computer industry is moving toward the ANSI version of C as fast as possible, considering the amount of existing software that is written in older versions. To avoid confusion, this book covers only ANSI C and the style of programming that is recommeded for it. Readers who have mastered ANSI C should have little difficulty understanding the programming styles used in older versions of the language.

Because C does not put the programmer in a straitjacket, programmers have more opportunities for making errors, a fact for which the language is often criticized. Yet powerful tools always require more care in their use than less capable ones. A high-performance automobile or airplane is far more demanding of the driver or pilot than a lower-performance machine. The solution is to make sure that driver, pilot, and programmer are properly trained in the use of their respective tools, including any necessary safety precautions. Like a conscientious driving or flying instructor, this book points out a number of areas in which the careless student is likely to get into trouble.

Newcomers to the language are sometimes intimidated by C programs, which look so different from programs in Pascal-style languages. Yet these differences are often superficial, reflecting merely different notations for the same underlying concepts. This book frequently points out the similarity of C constructions to those found in other languages (without, however, demanding knowledge of any particular other language).

About This Book

To get the student writing nontrivial C programs as quickly as possible, Chapters 1 and 2 focus on those features of C that have close counterparts in other languages. Only a few of C's idiosyncrasies, including the increment, decrement, and compound assignment operators, are introduced in these chapters. Only two data types, `int` and `double`, are introduced. To heighten the sense of familiarity, some of the examples and exercises were deliberately chosen from among such old standbys as Fibonacci's rabbit problem and finding the number of grains of wheat paid to the inventor of chess.

After Chapters 1 and 2 have brought the reader up to speed in C programming, Chapter 3 takes some time to explore some details of the language. Following a brief introduction to hardware memory

organization (which influences so much of C), the remaining arithmetic data types are introduced. Conversions between arithmetic types are also introduced, as are the additional conversion specifiers that enable `scanf()` and `printf()` to handle the types introduced in this chapter. In addition to the topics mentioned in the chapter titles, most of the remaining chapters have sections devoted to filling in background details about some aspects of the language.

Once we get beyond the elements of C, our attention invariably shifts to pointers and arrays, which are central to all advanced applications of the language. Chapter 4 introduces these two closely related concepts, but focuses mainly on array processing. Chapter 5 introduces pointer arithmetic and string processing, with some of the string library functions providing examples of both. Although the main topic of Chapter 6 is structures, pointers and arrays are never far from our thoughts.

So far, we have only used the standard input, output, and error streams, although file redirection has been suggested for systems that have that capability. Chapter 7 explores the C model of files and streams and shows the student how to open files in various modes, obtain file names from command-line parameters, read and write binary data, and use positioning functions for direct access. Functions with varying numbers of arguments and conditional preprocessor directives are the two "extra" topics discussed in this chapter.

Four appendices provide additional information. Appendix 1 lists C keywords and Appendix 2 gives the precedence, associativity, and arity (number of operands) of C operators. Appendix 3 describes a typical integrated development environment, in the hopes that newcomers will prefer this modern tool to the old-fashioned command-line compilers so beloved by many current C programmers. Appendix 4 introduces the *memory models* used with MS-DOS, one of two most popular operating systems for C programming (the other is UNIX, the operating system under which C was developed).

For Further Reading describes some sources of additional information, including the all-important ANSI standard.

Acknowledgments

I wish to thank the following for their helpful comments on the manuscript: Timothy J. McGuire, Texas A&M University; Greg M. Perry, Tulsa Junior College; Paul W. Ross, Millersville University, Millersville, Penna.; Vincent F. Russo, Purdue University; and Phillip C-Y. Sheu, Rutgers University.

Contents

3
Types and Conversions

4
Arrays and Pointers

5
Pointers and Strings

6
Structures, Unions, Bitfields, and Bitwise Operators

7
Streams and Files

1 Getting Started

WE BEGIN WITH the basic elements found in most programming languages: identifiers, types, values, variables, functions, expressions, and provisions for input and output. The C versions of these are largely similar to their counterparts in other languages. The greatest differences are in the input-output facilities, which vary considerably from language to language.

FUNCTIONS

A C program is a collection of *functions*, which play the same roles that functions, procedures, and subroutines do in other languages. When we run a program, the system first calls the function `main()`, which every C program must define. Function `main()`, in turn, can call other functions, which can call still other functions, and so on as needed to accomplish the purpose of the program. Thus function `main()` plays the same role in C as the "main program" does in some other languages.

As can be seen in the previous paragraph, this book uses a distinctive typeface for elements of a C program, such as `main()`. Also, the name of a function is followed by a pair of parentheses. The latter convention lets us see at a glance that names such as `main()`, `printf()`, and `scanf()` represent functions rather than other elements of a C program.

In some other languages, such as Pascal, Basic, and Fortran, the term *function* is reserved for subprograms that return a value. Other terms, such as *procedure* and *subroutine*, are used for subprograms that do not return a value. In C, however, all such subprograms are

known as functions. The definition of each individual function determines whether or not it returns a value.

To save us from having to "reinvent the wheel" every time we write a program, all C implementations come with a large *library* of predefined functions. Additional libraries can be created by the programmer or purchased from third-party developers.

THE HELLO-WORLD PROGRAM

It is traditional to begin the study of C with a program that prints the message "Hello, world!" Listing 1-1 shows our version of the hello-world program. Working through this listing line-by-line will acquaint us with the structure of a C program.

Comments

The first two lines of Listing 1-1 are *comments*, which explain the program to human readers but are ignored by the compiler. In C, comments are enclosed by the symbols `/*` and `*/`:

```
/* This is a comment */
```

A comment can extend over several lines:

```
/* Now is the time
   for all good programmers
   to use more and better comments */
```

Nested comments—comments within comments—are *not* allowed. The following will produce one or more error messages:

```
/* This is an /* invalid */ comment */
```

Our hello-world program (Listing 1-1) begins with two comments:

```
/* File hello.c */
/* Say hello to user */
```

The first comment gives the name of the *source file*—the disk file in which the program is stored (Listing 1-1 is just a printout of this disk file). Although the reader seldom actually needs to know these file names, mentioning them in comments emphasizes the correspondence between listings and disk files. Including such comments in your own programs will help you keep track of which listings go with which files.

The second comment describes briefly what the program does. As is often done, the comment is phrased as the command we might give the computer if it could understand English.

Listing 1-1

```
/* File hello.c */
/* Say hello to user */

#include <stdio.h>

int main(void)
{
   printf("Hello, world!\n");
   return 0;
}
```

Header Files, Preprocessing, and `#include` Directives

Every function must be declared before it is called; the *declaration* contains information that the compiler may need to call the function properly. This requirement applies to library functions as well as those defined by the programmer.

To help us declare library functions, the C implementation provides *header files* containing the necessary declarations for various parts of the library. For example, the header file `math.h` contains declarations for mathematical functions, the header file `stdio.h` contains declarations for standard input and output functions, and so on. We sometimes use the names of the header files to refer to different parts of the library. We speak of the functions declared in `math.h` as the *math library*, the functions declared in `stdio.h` as the *stdio library*, and so on.

The C compiler provides a simple way of effectively inserting header files into source files. The first phase of compilation is *preprocessing*, which manipulates the text read from the source file before sending it on to the remaining phases of compilation. Some of these manipulations can be controlled with *preprocessing directives*, all of which begin with the symbol #.

An `#include` directive designates a file whose contents are to be inserted into the program text. When the preprocessor encounters an `#include` directive, it replaces the directive by the contents of the designated file. This replacement takes place only in the text that is being passed on to the rest of the compiler. The original source file and the file that was included both remain unchanged.

The third line of the hello-world program includes the header file `stdio.h`:

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

The file `stdio.h` contains the declaration for `printf()`, the library function that the program calls to print the hello-world message. Note that the file name is enclosed by a less-than sign and a greater-than sign, which serve as angle brackets.

When C programmers say that one file *includes* another, they always mean that the one file contains an `#include` directive that names the other file. They do *not* mean that the one file contains a copy of the contents of the other file.

Defining `main()`

The remainder of the hello-world program defines the function `main()`. A function definition has the following form:

```
heading
{
    declarations (if any)
    statements
}
```

The *heading* provides the information needed to call the function: the name of the function, the types of arguments (if any) that it requires, and the type of value (if any) that it returns.

The heading is followed by the *body* of the function, which specifies the action the function will take when it is called. The body of the function is a *block*, which is a series of declarations and statements delimited by the braces { and }. The braces play the same role in C as do the words **begin** and **end** in many other languages.

The hello-world program declares `main()` as follows:

```
int main(void)
{
   printf("Hello, world!\n");
   return 0;
}
```

The heading

```
int main(void)
```

gives the name of the function as `main`. The `void` in parentheses means that the function takes no arguments. The `int` indicates that the function returns an integer value (`int` is the most frequently used of C's numerous integer data types).

The body of the function is a block containing two statements:

```
{
 printf("Hello, world!\n");
 return 0;
}
```

Each statement ends with a semicolon. The first statement, which calls the library function `printf()`, will be discussed in more detail shortly. The second statement causes `main()` to return a value of 0 to the system. The value returned by `main()` is a *return code* that indicates whether the program ran successfully. Customarily, a return code of 0 indicates a successful run.

Because of the special role of `main()`, its definition can be abbreviated somewhat:

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

The full heading `int main(void)` can be shortened to `main();` the return type defaults to `int`, and no information is provided about possible arguments. If the return code is not used by the system, which is often the case, the `return` statement also can be omitted. The abbreviated form, which was standard in pre-ANSI versions of C, is so widely used that the reader needs to be able to recognize it. However, we will continue to use the unabbreviated form recommended for ANSI C, and we suggest that the reader do likewise.

Strings, Escape Sequences, and calling `printf()`

Pieces of text, such as `Hello, world!`, are stored as *strings*. We can represent a string in a C program by a *string literal*, which consists of the characters of the string enclosed in quotation marks. For example, the message `Hello, world!` can be represented by the string literal

```
"Hello, world!"
```

Some characters that we may wish to include in string literals do not have conventional graphics such as `a`, `b`, and `c`. Such a character can be represented by an *escape sequence*, which consists of a backslash, `\` , followed by a letter or number representing the character. A case in point is the *newline character*, which causes the output device

to start a new line. The newline character is represented by the escape sequence `\n`. If the string represented by

```
"Hello, world!\n"
```

is sent to an output device, the device will print `Hello, world!` on the current line, then go to the beginning of the following line.

We call a function by writing the name of the function followed by a pair of parentheses. The paired parentheses are known as the *function-call operator.* Inside the parentheses are listed any argument values that are to be passed to the function.

The library function `printf()` is a powerful function for formatting and printing output. It can be called with different numbers of arguments depending on how it is being used. The simplest use of this function is to call it with a single argument, which is a string to be printed. For example, the function call

```
printf("Hello, world!\n")
```

calls `printf()` and passes it the string represented by the string literal `"Hello, world!\n"`. When called, `printf()` writes the message `Hello, world!` and starts a new line.

The functions in `stdio.h` access input and output devices via *streams.* Each stream is connected to a source or destination for data, such as a keyboard, a display, a printer, or a disk file. Data read from or written to a stream comes from or goes to the device or file to which the stream is connected. The function `printf()` writes to the standard output stream, `stdout`, which is normally connected to the user's display. By changing this connection (we'll see how later), we could send the output from our program to a printer, a disk file, or even a communcations link rather than to the user's display.

The preceding function call is a C *expression,* which could conceivably be part of a larger expression. If we wish to use an expression as a statement, rather than as part of a larger expression, we convert it to an *expression statement* by following it with a semicolon. For example, the hello-world program uses the expression statement

```
printf("Hello, world!\n");
```

to print the desired message. An expression statement *always* ends with a semicolon, regardless of where it may occur in the program (such as embedded within another statement).

Note that `printf()` does not automatically cause the printer to start a new line after a string has been printed. For example, the statements

```
printf("abc");
printf("def");
printf("ghi");
```

print

```
abcdefghi
```

If we wish the strings printed on separate lines, we must include newline characters in the string literals:

```
printf("abc\n");
printf("def\n");
printf("ghi\n");
```

These statements produce the output

```
abc
def
ghi
```

and cause the output device to start a new line after the third line is printed.

Newline characters can be embedded within a string. For example, the preceding output can also be produced by the single statement

```
printf("abc\ndef\nghi\n");
```

EDITING, COMPILING, AND LINKING

The text of a C program is stored in one or more source files. For simple programs, such as most of the examples in this book, a single source file will suffice. For large programming projects, however, multiple source files are the rule. Different source files contain logically distinct parts of the program and may have been written by different programmers. Source files are created and revised with the aid of a *text editor*.

We use a C *compiler* to translate each source file into a machine-coded *object file*. Each source file is compiled separately. If we need to make changes in a source file, then only that source file needs to be recompiled. This is one reason for using multiple source files: it is generally much faster to recompile one source file than to recompile all the program text for a large project.

The *linker* combines the object files with one another and with the code for any library functions called by the program. The output

from the linker is an *executable file*, which contains all the machine code for the program and is ready to be run on the computer. Even for a program with only one source file, linking is still needed to combine the code in the object file with the code for whatever library functions the program calls.

Unfortunately, the detailed commands for editing, compiling, and linking vary too much between implementations for us to consider them here. However, it is worth mentioning two different kinds of implementation: command-line implementations and integrated development environments.

A *command-line implementation* is so called because each software tool, such as a text editor, compiler, or linker, must be invoked by typing an operating system command line. The programmer must invoke a text editor and use it to create or modify each source file. Next, the compiler must be invoked for each source file to translate it into an object file. The linker is then run to produce the executable file. Finally, the C program can be run on the computer to see if it executes properly. If it does not, or if error messages were encountered in any of the preceding steps, the programmer must return to the text editor to locate the errors and make necessary corrections.

Many operating systems provide *scripting* facilities (UNIX scripts, MS-DOS batch files) that allow new operating system commands to be defined in terms of existing commands. Command-line implementations often use these facilities to provide simplified commands for common situations. For example, there may be a single command for compiling, linking, and executing a program that has only one source file. Also, many command-line implementations provide a *make* utility, which will automatically compile and link all the source files in a programming project. The make utility recompiles only those source files in which changes have been made.

An *integrated development environment (IDE)* is a single software tool that can be used for editing, compiling, linking, executing, and debugging C programs. (The term *integrated programming environment (IPE)* is also used.) Normally, the IDE serves as a text editor, allowing us to create and modify source files. When we are ready to compile, link, or execute, however, we can do so with commands to the IDE rather than with operating system command lines. Generally, an IDE will do whatever work is necessary to carry out our commands. For example, if we tell it to execute a program, it will first do any compiling and linking necessary to produce an executable file. Like the make utility, the IDE will recompile only those source files in which changes have been made, and will relink only if one or more source files had to be recom-

piled. An online help facility provides easy access to information about IDE commands, compiler and linker error messages, and the C library.

IDEs also provide help for debugging. When the compiler discovers errors, the erroneous statements are highlighted on the screen. During execution, the programmer can monitor the values of selected variables and can trace the flow of control—the order in which program statements are executed.

Some old hands at C programming love their command-line compilers and would rather fight than switch. Newcomers, however, are urged to explore the power and convenience of integrated development environments. A typical microcomputer IDE is described in Appendix 3.

IDENTIFIERS

As in most other programming languages, C programmers must devise names for such program elements as variables and functions. These names, or *identifiers*, must be formed according to certain rules:

- An identifier can contain only letters, digits, and the underscore character, _. Thus `amount` and `hit_count` are valid identifiers but `$_amount` and `employee_#` are not.
- An identifier must begin with a letter of the alphabet or an underscore character. Thus `_dos_call` is a valid identifier but `1st_round` is not.
- Identifiers that begin with an underscore are reserved for the implementation and should not be defined by the programmer. Thus you should not define an identifier `_dos_call`, but you *can* use it if it is already defined by the implementation.
- An identifier must not be the same as one of the *keywords* listed in Appendix 1. Thus `int` and `void` are not valid identifiers, because they are keywords.
- C distinguishes between uppercase and lowercase letters, so that `amount`, `Amount`, and `AMOUNT` are three different identifiers.
- Only the first 31 characters of an identifier are significant—that is, are used in distinguishing one identifier from another. Thus

  ```
  very_very_very_long_identifier_1
  ```