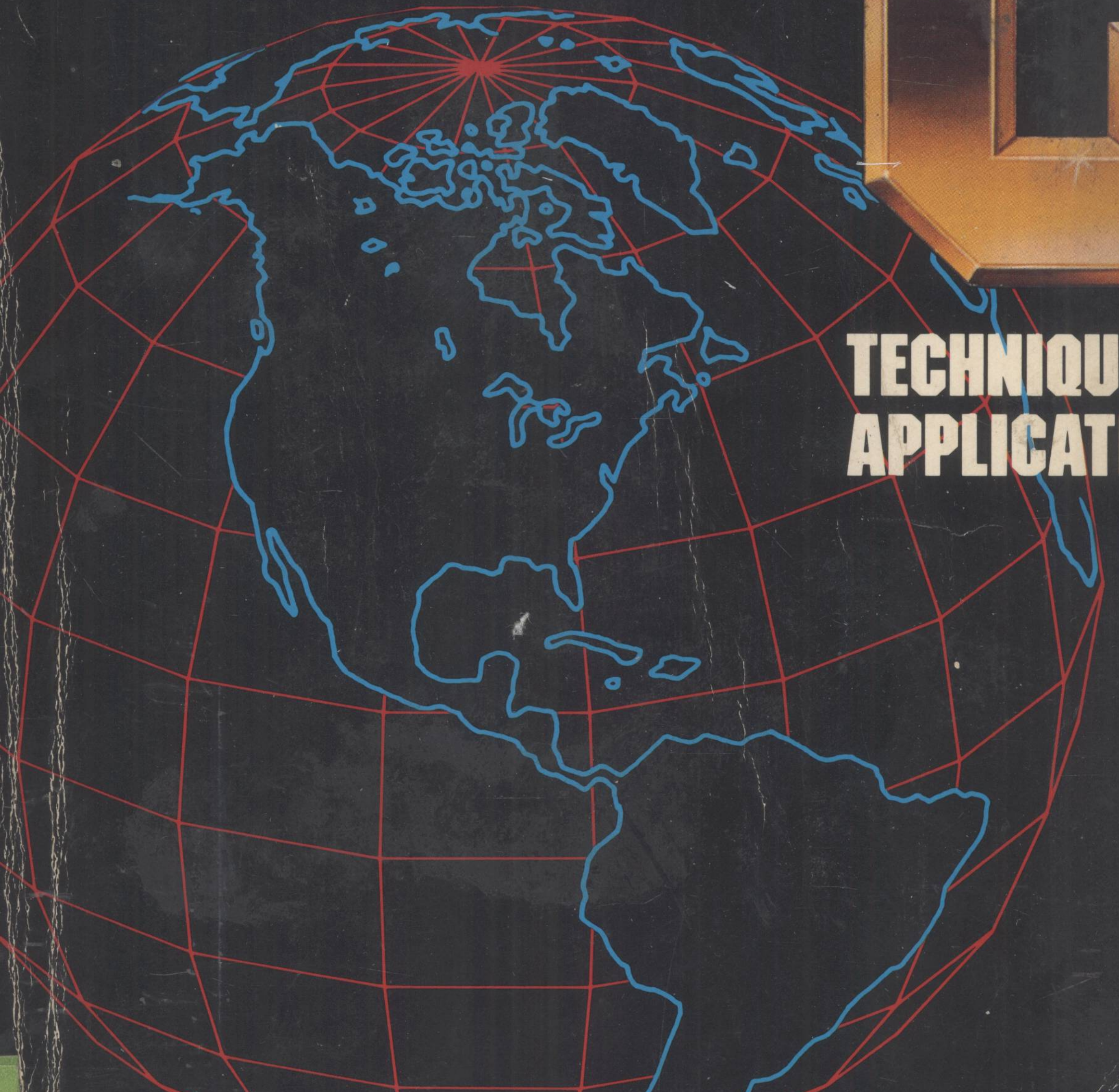
ADVANCED C

TECHNIQUES & APPLICATIONS



que™

TP312
S677

Advanced C: Techniques and Applications

Gerald E. Sobelman

David E. Krekelberg

Que Corporation
Indianapolis, Indiana

Library of Congress Catalog No.: 85-60694
ISBN 0-88022-162-3

Editorial Director

David F. Noble, Ph.D.

Editors

Jeannine Freudenberger, M.A.
Katherine S. Ewing
Kathie-Jo Arnoff

Technical Editor

Chris DeVoney

88 87 86 85   8 7 6 5 4 3 2 1

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit number, the number of the book's printing. For example, a printing code of 85-4 shows that the fourth printing of the book occurred in 1985.

Dedication

To Joanne
—G.E.S.

To my parents
—D.E.K.

About the Authors

Gerald E. Sobelman

Gerald E. Sobelman is a senior technical consultant at Control Data Corporation. He received a Ph.D. in physics from Harvard University and has been involved in the development of integrated circuits for five years. Dr. Sobelman has had more than 10 years of computer programming experience. He has published nine technical papers and has given presentations at several major technical conferences.

David E. Krekelberg

David E. Krekelberg is a computer-aided design research engineer at Control Data Corporation. He received a B.S.E.E. degree from the University of Minnesota Institute of Technology in 1982. At age 24, Mr. Krekelberg has more than 10 years of programming experience and has held software design positions at Sperry Corporation and Control Data Corporation. Mr. Krekelberg's work on CAD tool design and development has been published in the proceedings of several major technical conferences.

Preface

This book is an excursion into the world of advanced programming in the C language. The book is written for several groups of programmers: first, those who have a basic knowledge of C and wish to extend their understanding of the language; second, programmers of C or of other languages who want to enhance their programming skills; and finally, programmers who are interested in building state-of-the-art applications, particularly applications involving graphics or advanced user interfaces.

The book is divided into three sections. The first part (chapters 1-4) deals with several advanced aspects of the C language itself, including techniques for designing large programs, the use of dynamic (updated) memory allocation, construction of user-defined data types, and the use of functions. Top-down design, information hiding, and the use of multiple source files are described. Pointers are emphasized throughout the book, especially in the descriptions of dynamic memory allocation and complex data types. In the chapter about functions, the stress is on two advanced topics: recursion and pointers to functions.

The second part of the book (chapters 5-6) examines in detail several techniques for constructing linked lists and trees. These techniques are presented in general formats so that the programmer can store and manipulate different types of elements. Various ways of extending the basic data structures are also discussed.

In the third part of the book (Chapters 7-9 and Appendixes A, B, and C), these ideas and techniques are illustrated through the construction of two related function libraries. The first function library (Appendix A) provides a set of routines by which applications programmers can generate and control complex graphics operations. Chapter 7 explains the basic concepts of the code presented in Appendix A. The second library (Appendix B) builds on the first. With the second library, programmers can implement an advanced user interface including multiple menus and windows. Chapter 9 explains the basic concepts of the code presented in Appendix B. Screen images derived from the code in Appendix B are in Appendix C.

The code that appears in Appendix A and B of this book is available from the Que Corporation on disk. To order, use the form in the back of the book.

We would like to thank Pegg Kennedy and Chris DeVoney of Que for their encouragement and assistance throughout the course of this project.

Gerald E. Sobelman
David E. Krekelberg

Minneapolis, Minnesota
1985

Trademark Acknowledgements

Table of Contents

1 Programming Style

Programming languages are tools for solving problems. The steps for solving a complex programming problem can be expressed in general terms, but to implement the solution on a computer, a specific programming language is required. Every language has special characteristics that dictate the style of coding necessary to achieve good results. As the size and complexity of the program increases, stylistic considerations become more important to the success of a programming project.

This chapter presents several methods for solving complex programming problems in the C language. In particular, the text introduces techniques for dividing large tasks into sets of functions and smaller files so that the programs run efficiently and are easily portable among different C environments.

Program Structure

C is an ideal language for the design and implementation of complex programming projects. This capability is due largely to the great flexibility provided by the proper use of functions. The language lends itself naturally to a top-down method of design, in which large and complex problems are broken down into successively smaller pieces. To use this technique, you begin by identifying the overall flow of control and the principal tasks. Then you can easily write a C pseudocode (a kind of structured English) description of the high-level design. In writing your descriptions, you assume that a function exists to perform each major task. Next, you examine the individual high-level functions and divide each function into a set of simpler ones. This refinement process can be continued until you reach a stage where you can write straight C code to perform each required action.

Block Structure

An important difference between C and other modern programming languages (such as Pascal) is seen in C's concept of block structure. In other languages, a complex function consists of a package of smaller functions bundled together to form a single larger unit. C treats all functions as separate entities. The set of functions assembled to solve a particular programming problem can be thought of as a collection of "tools" used to perform the job at hand. With C you actually have a "toolbox" of functions that are applicable to many programming projects.

This concept is best illustrated by an example. Suppose you want to write a program that plays a board game with a user. Alternate moves are made by each "player" until one side wins. At the highest level, the computer and the human player exchange a series of moves; so you may start by writing pseudocode that describes the main stages of the program. The statements of the pseudocode can be easily translated into function calls. In this case, the main function can be sketched as follows:

```
initialize game flags
initialize display
while the game is still on
    {
    if it's the computer's move
        {
        determine the next move
        update the display
        set flags according to result
        }
    else
        {
        wait for move to be entered
        check legality
        update the display
        set flags according to result
        }
    }
```

You should note several important points about the pseudocode. First, for every statement dictating some sort of action, you must create a function to perform that task. For this reason, the statements in the preceding pseudocode *initialize game flags, initialize display, determine the next move,* and so on, are candidates for high-level functions. Second, the control structures *while, if,* and *else* map directly into the corresponding C constructs. Finally, status phrases such as *the game is still on* and *it's the computer's move* translate into integer variables used to hold information about the current state of the game. These variables are used by the control structures to make decisions.

Most of these high-level functions are not simple enough to translate directly into C code, and the functions themselves must be broken down into smaller functions. For instance, consider the move generation function, which may be pseudocoded as follows:

```
find all legal moves
for all such moves
    {
    evaluate position
    if best move so far
        update selected move
    {
return selected move
```

This function follows the same pattern as the first one. Clearly, *find all legal moves* and *evaluate position* are themselves complex functions that can be further broken down. On the other hand, *update selected move* is probably simple enough to be coded directly in C. By following this procedure through to its conclusion, you can develop a full working program.

Whether you are writing pseudocode or actual C code, you must consider indentation and the location of braces in order to develop readable programs. C programs can easily be logically correct and completely unreadable. Several styles of arranging the text on the page are available. These styles ensure that the resulting code is easily understandable. The coding style used in this book is one that looks natural and clean. The important point, though, is for you to choose and stick to a style that suits you.

Variable Names in Block Structure

The concept of block structure in C is evident in the scope of variable names. *Block* means the code between any matching pair of left and right braces. A block may be an entire function or a set of actions following a control structure like `while`, `if`, or `else`. You can define variables that are local to any specific block. For example, you frequently need a temporary variable to count some range of values. Access to this variable is probably not needed in other parts of the program. Therefore, defining the variable so that it is known only within the set of statements in which it is active makes sense. In the following code fragment, the variables `items` and `min` are defined outside the block, but the variable `i`, which is needed only within the block, is defined inside the block. The local version of the variable `i` does not interfere with another variable named `i` that may occur in the enclosing block or in a separate block.

```
int items, min;

/* set items and min to some values */
items = 3;
min   = 1;
               . . .

/* the block */
if (items > min)
    {
    /* define i to be local to this block */
    int i;

    /* begin loop that uses the local i as a counter */
    for (i = Ø; i < items; ++i)
        {
        /* statements that will  be executed "items" times */

                         . . .

        }
    }
```

The variables that have been discussed are called *automatic variables*. For these variables, storage is allocated at the entry to a block and deallocated at the exit from the block. No memory of previous values is retained on succeeding passes through the block. Before these variables are initialized, their values are unpredictable. You can override this default condition through the use of the `static` facility. A variable that is defined as *internal static* is allocated storage at the beginning of program execution and that storage is not deallocated until the program has terminated. This facility, then, provides a means for permanent private storage.

External variables are defined outside any block (and therefore outside any function) and are available for use within groups of functions. Function names are considered external and so are visible inside other functions. Controlling the extent of this visibility is important.

Separate Compilation and Information Hiding

C programs can be partitioned into several files. A good rule of thumb is to put logically related functions in the same file. Each file can be compiled independently and linked to other compiled files so that individual pieces of a large program can be debugged efficiently. If a compilation error is found, only a small portion of the program needs to be recompiled, and the edit-compile process can proceed rapidly. In addition, small *driver programs* that test the functions contained within a single file can be written. In this way, many run-time bugs can be fixed quickly on a file-by-file basis.

In addition to increasing efficiency, the careful use of separate files can be used to hide implementation details. This practice not only simplifies and modularizes the total program but also prevents the unintended overlap of variable or function names that may occur in a large program, especially if more than one person is involved in program design.

In a discussion of the visibility, or scope, of names in C, the definition of an external variable or a function must be distinguished from the variable's or function's declaration. An external element can be defined only once. In addition to specifying type information, the definition allocates storage and may also set initial values. An external name is visible in the file in which the name is defined from the point of definition until the end of the file. If the variable or function name is needed at an earlier position in that file or in an-