C as a Second Language

For Native Speakers of Pascal

Tomasz Müldner

Peter W. Steele

C as a Second Language

For Native Speakers of Pascal

Tomasz Müldner

Acadia University
Nova Scotia, Canada

Peter W. Steele

Acadia University
Nova Scotia, Canada

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts   Menlo Park, California   New York
Don Mills, Ontario   Wokingham, England   Amsterdam   Bonn   Sydney   Singapore
Tokyo   Madrid   Bogotá   Santiago   San Juan

This book is in the Addison-Wesley Series in Computer Science

Michael A. Harrison, Consulting Editor

Library of Congress Cataloging-in-Publication Data

Müldner, Tomasz.
C as a second language.

1. C (Computer program language) I. Steele, Peter W.
II. Title.
QA76.73.C15M85 1988 005.13'3 86-28837
ISBN 0-201-19210-1

ABCDEFGHIJ-HA-8987

Preface

C was originally designed in 1972 by Dennis Ritchie at AT&T Bell Laboratories [Rit78]. The first implementation was on a DEC PDP-11; today, C compilers are available for almost all computers, including mainframes, minicomputers, and microcomputers. The popularity of C has grown tremendously over the past years; indeed, it is one of the most popular languages in use today, with a wide variety of applications, such as editors, compilers, databases, and operating systems (in particular, Unix [Rit78]), being written in the language.

This book is called *C as a Second Language* because it is designed for readers who already have experience in programming in Pascal [Jen78] and want to learn to program in C. It is still possible to read the book without knowing Pascal, but a good knowledge of some high-level programming language is necessary.

The book is for students, professional programmers, and computer hobbyists studying C on their own. It is especially suited to students who are either required to study C or who will be entering some high-level computer science courses, such as operating systems or compilers, in which a knowledge of C may be needed.

Our main goal in this text is to show how to write reliable programs in C and how to translate programs from Pascal to C.

A secondary (but not neglected) goal in designing this book was to provide the reader with easy reference to various topics when programming in C.

Another goal is to use a portable C, which in practice means that we have used the Kernighan and Ritchie standard [Ker78]. Many extensions of this standard have appeared, for example, those described in ANSI C [Ans86]: void procedures, enumeration types, and so on. We have described most of the common extensions but have also made it clear that they are nonstandard. (As of the writing of this book, the ANSI C standard has not been completed.)

The first step in learning C is to understand its syntax. We help the reader through this step by comparing C constructs with Pascal's corresponding constructs.

The second step is to understand thoroughly the semantics of the language. Here, our approach is to show several small C programs, explain their semantics, and warn what the typical pitfalls are.

The last step is to develop a programming style. For this, we show larger programs written in a style often used by C programmers. This style is in places concise, but we have avoided "tricky" code even if this could result in less efficient code.

The book has two basic parts. The first part (Chapters 2–11) is an introduction to C that encompasses the goals of the first two steps mentioned above. Since C is likely to be new to the reader, these chapters concentrate on the language itself. The algorithms are fairly simple, sometimes even trivial, so the reader does not have to spend time trying to understand the data structures and algorithms being used.

The second part (Chapters 12–14) assumes a basic understanding of the underlying concepts of C that were introduced in the first part. We continue to discuss programming techniques in C and present additional features of C through examples of larger programs. The topics presented and the data structures and algorithms covered in this part are more advanced.

Important features of the book include:

- **A thorough and detailed description of the C programming language and programming techniques,** including a comparison of C and Pascal constructs and coverage of advanced topics such as the dangling reference problem, heap compaction, and others.
- **Early introduction of pointers and complete,** detailed description of relations between pointers and arrays, functions, and strings (Chapters 7, 8, 10).
- **Implementation in C of data structures and algorithms** such as: linked lists (both pointer based and array based), an array implementation of a stack, an open hash table implementation of a dictionary, an adjacency list representation of a graph, a bit-vector implementation of a set, techniques to manage files containing structured data, memory management with heap compaction, internal and external search techniques, a "virtual" sort that can be applied to arrays of any type of element, and an implementation of an external binary search tree with file compaction.
- **Various implementations** of C under three different environments: Unix, MS-DOS, and the Macintosh. We show how to use specific compilers in each of these environments (Chapter 14). Then through **programming examples,** we discuss such topics as: I/O redirection, error handling, and low-level file I/O under Unix; the ANSI C standard, MS-DOS services, directories, and file attributes under MS-DOS; window and menu management on the Macintosh.

Pedagogical Aids

Each of the ten introductory chapters begins with a review of Pascal constructs, which describe such concepts as the system stack and the heap as they relate to standard Pascal [Coo83]. Following each review is a glossary of terms, which should be understood before studying the rest of the chapter. In some instances, the reader may start reading a chapter from the glossary, referring to the Pascal review only if needed. (For more detail on the basic concepts of programming languages, refer to "Programming Languages: Design and Implementation" by Pratt [Pra84].)

Our book teaches C through examples. We offer more than 100 complete programs, varying from one as simple as finding the maximum of two integers to one that implements an external binary search tree. Most examples (with the exception of those in Chapter 14 on real systems) are written in a portable style. Each chapter closes with lists of things to remember and common errors made by novice C programmers. We also provide over 100 exercises and give solutions to all odd-numbered exercises.

Chapter 1 gives some basic definitions used throughout the book, such as the run-time system of a language. It also briefly compares Pascal and C. Chapter 2 is on primitive data types and basic terminal I/O. Chapter 3 describes control structures, and Chapter 4 is an introduction to file I/O (which is discussed further in Chapters 12 and 14). Chapter 5 is on preprocessing; Chapter 6 describes functions and scope rules; Chapter 7 is on pointers; and Chapters 8 and 9 define arrays, structures, unions, and enumeration types. Chapter 10 describes C string operations, and Chapter 11, bitwise operations. Chapter 12, a continuation of Chapter 4, discusses additional details of text and binary file I/O. Chapter 13 presents some application programs:

- A calculator program, which shows the implementation of linked lists and open hash tables
- An implementation of Dijkstra's algorithm to find the shortest path in a graph, which also shows the implementation of sets and graphs
- A database program, which shows the implementation of external binary search trees

Chapter 14 describes through several examples the use of C under some real systems—Unix, MS-DOS, and the Macintosh.

Appendices contain a list of C keywords and a complete description of C's syntax (Appendix A), precedence and associativity tables (Appendix B), a detailed description of formatted I/O (Appendix C), and an ASCII table (Appendix D). Appendix E contains solutions to the odd-numbered programming exercises. Appendix F consists of a table comparing Pascal and C, and Appendix G contains a summary of the standard library functions. We provide a program index to allow easy reference to any program in the book.

A disk containing complete source code of all examples may be ordered from the authors. A check for $30 (payable to one of the authors) may be sent to Doctor Tomasz Müldner or Professor Peter Steele, c/o School of Computer Science, Acadia University, BOP 1XO Wolfville, NS Canada. For classroom adoptions of 25 or more, contact your local Addison-Wesley representative.

Acknowledgments
We wish to thank a number of people who helped us complete our work. Our colleague Dr. Rick Giles carefully read the first draft and made a number of useful suggestions. Many thanks to our students: Bill Nickerson, who helped us debug most of our programs; Donna Cleveland, who helped prepare some of the exercises; Gilbert Verghese and Barbara Müldner, who suggested many corrections in the early versions of the manuscript; Steven Langlois, our last but most thorough proofreader; and finally, the winter 1985 class of Computer Science 2023 at Acadia, the first students we tested our book on. Also, special thanks to our colleague Dr. Wayne Brehaut for his help with many grammatical aspects of the manuscript.

We would as well like to thank the four software companies that allowed us to review their products: Lattice Incorporated for Lattice C; Rational Systems, Inc. for Instant-C; Manx Software Systems, Inc. for Aztec C; and THINK Technologies, Inc. for LightspeedC.

We would also like to express our appreciation to the following reviewers of the manuscripts: Andrew W. Appel, Princeton University; Judith D. Boxler, Vancouver Community College; David Cohrs, University of Wisconsin, Madison; Jon Forrest, Britton Lee, Inc.; Michael A. Harrison, University of California, Berkeley; Raj Nagendra, Ryerson Polytechnical Institute; Paul W. Ross, Millersville University; Henry Ruston, Polytechnic University, Brooklyn.

Nova Scotia *Tomasz Müldner*
Canada *Peter Steele*

Contents

1 Introduction

PREVIEW *This chapter introduces some basic notions that you may need to understand later chapters. First of all, we briefly describe C and then compare Pascal with C. Next, we discuss the representation of the integer and character data types in memory. Finally, we present the concept of a run-time system, include files, and conditional and separate compilation in C.*

1.1 ABOUT C

C is often referred to as a *low-level* high-level language because it provides many tools that allow low-level operations to be performed. As a result, C programs have gained a reputation for being unstructured and difficult to read. In some respects, this reputation may be deserved, but to say it is an exclusive trait of C is unfair. Unreadable programs can be written in any programming language without much difficulty. Furthermore, like a true high-level language, C provides many high-level features, and with some discipline, programs written in C can be as structured and readable as programs written in Pascal. Another important characteristic of C is that most implementations are very efficient.

C provides the typical primitive data types: **character, integer,** and **real**. It also provides structured data types: **arrays, records,** and **unions**. A complete set of control structures is provided, including **conditional**, **selective**, and **iterative** statements. Functions may be recursive, although they may not be textually nested. Programs may be divided into **separately compiled** modules, and a flexible set of scope rules exists to assist in this sort of modularization. One of the main strengths of C is its approach to **pointers,** and many language constructs (such as call by reference) are implemented using pointers. Standard I/O is provided by **run-time libraries**, which makes it easier to port compilers to other machines. C does not provide any high-level data types like those in Ada [Ada83], nor any tools for parallel programming.

For many years, the definition of the C language given by Kernighan and Ritchie [Ker78] was the accepted standard definition. Today, C is being standardized by the American National Standards Institute Subcommittee (ANSI) [Ans86]. Fortunately, most of the proposals in the ANSI C definition are extensions to the Kernighan and Ritchie standard, and most existing implementations today comply with Kernighan and Ritchie, with some newer compilers adopting aspects of ANSI C. Therefore, with a little effort, it is possible to write "almost" portable programs, and only minor changes, if any, are needed to port programs from one machine to another.

1.2 A BRIEF COMPARISON OF PASCAL AND C

In this section, we briefly compare Pascal and C constructs. Our purpose is not to give you detailed information about any specific constructs but to give you a general idea of what the C programming language is like.

Identifiers and Comments

The syntax of **identifiers** is almost identical in both languages except that in C an underscore _ is allowed, and they are *case sensitive*. **Comments** are similar as well:

Pascal	C
`(* comment *)`	`/* comment */`

Program Modules and Scope

A program in C is a sequence of functions. One function must be called `main`; it is from this function that the execution of the program starts. Although the **scope rules** in C are Pascal-like, C functions may not be textually nested; however, they may contain **blocks** (compound statements containing declarations as *well* as statements). Unlike Pascal, subroutines in C are always defined as functions, although any function may be called as a procedure, in which case the value returned by the function is disregarded. As in Pascal, functions in C may be recursive. A simple example:

```
PROGRAM one;

PROCEDURE proc;                    void proc()
BEGIN                              {
    WRITE('Hello');                    printf("Hello");
END;                               }

FUNCTION func : INTEGER;           int func()
BEGIN                              {
    func := 3;                         return(3);
END;                               }
```

```
VAR i : INTEGER;                  int i;
BEGIN                             main() {
    proc;                             proc();
    i := func;                        i = func();
    WRITE(i);                         printf("%d", i);
END.                              }
```

The preceding C program has three functions defined: `main`, `proc`, and `func`.

Unlike standard Pascal, C supports **separate compilation** of functions; the keyword `extern` specifies an object that is defined in another file.

Data Types and Declarations, Parameters

Pascal supports the primitive data types **integer**, **real**, **boolean**, and **character**. C supports the types `int`, `float` (which is the same as **real** in Pascal), and `char`. C has no type boolean, although any nonzero integer value is equivalent to true, and zero is equivalent to false. Moreover, C provides a special qualifier `long`, which can be used to define **double precision integers** having approximately twice as many significant digits; similarly, C provides a type `double` to define **double precision reals.**

The syntax of declarations in C is the reverse of that in Pascal; for example,

```
VAR i : INTEGER;                  int i;

VAR c : CHAR;                     char c;
```

In C, data can be *initialized* in definitions; for example,

```
int i = 3;
```

Pascal supports structured data types **array, record,** and **file**, and C supports the same types, although files are not predefined in the language; instead they are defined in the system library.

Examples

```
VAR arr : ARRAY [0..3] OF INTEGER    int arr[4];

rec : RECORD                         struct {
        i : INTEGER;                     int i;
        r : REAL;                        float r;
      END                            } rec;
```

Function definitions are similar, although formal parameter specifications are in different places; for example,

```
PROCEDURE proc(i:INTEGER);    void proc(i)
                              int i;
                              {
VAR j:INTEGER;                    int j;
BEGIN
    j := i + 3;                   j = i + 3;
END                           }
```

Pascal supports two modes of parameter transmission—call by value and call by reference. C supports only call by value; call by reference must be *simulated* using pointers.

Expressions and Statements
Expressions in C are similar to those in Pascal except that types may be freely intermixed with *automatic type conversion* taking place.
  Operators and other constructs are similar, but there are some minor syntactic differences:

Equality:	=	==
Inequality:	<>	!=
Logical AND:	AND	&&
OR:	OR	||
NOT:	NOT	!

```
Assignment:               x := 3*(y+6);            x = 3*(y+6);

Conditional statement:    IF a>5 THEN a:=a-b       if (a>5) a=a-b;
                          ELSE b:=a;               else b=a;

Iteration statements:     WHILE x>0 DO x:=x-1;     while (x>0) x=x-1;

                          REPEAT x:=x+2            do x=x+2;
                          UNTIL x>100;             while (!(x>100));

                          FOR i:=1 TO 10 DO        for (i=1; i<=10; i++)
                            x[i] := 0;               x[i] = 0;

Selection statement:      CASE i OF                switch (i) {
                          1:   j := 3;             case 1: j = 3;
                                                           break;
                          2:   j := j-1;           case 2: j--;
                                                           break;
                          END;                     }

Compound statement:       BEGIN s1; . . . ;sk END  { s1; . . . ;sk }

Return from a function:   func := exp;             return(exp);
                          end (* function *)
```

Pointer declaration:	`VAR p : ^INTEGER;`	`int *p;`
Pointer dereferencing:	`p^`	`*p`
Access to record fields:	`rec.`*field*	`rec.`*field*
Record pointer access:	`recpnter^.`*field*	`recpnter->`*field*

A single dimensional array in C is considered a pointer to the memory block allocated for this array. C allows arithmetic on pointers; for example, if x is a single dimensional array, x+1 refers to the second element of the array. A function name without parentheses is also a pointer, in this case a pointer to the code of this function.

Other C Constructs
C supports **macros** with parameters. These macros are expanded before compilation of the program. The C preprocessor also handles **file inclusion** and **conditional compilation**.

1.3 THE COMPUTER'S MEMORY

A computer's memory consists of a number of **words**. Each word consists of a number of **bytes**, and each byte consists of a number of bits (usually 8, though not always). For example, 16-bit words consist of two 8-bit bytes, and 64-bit words consist of eight 8-bit bytes.

For the most part, we assume that each memory location is one byte in size and that a single character may be stored in a byte. This is usually referred to as a **byte-oriented** memory architecture and is the most common architecture in use today, with machines such as the VAX-11 and IBM 360 and their successors using it. Machines that support so-called **word-oriented** architectures are less common. In these architectures, the smallest addressable unit is a word rather than a byte. For example, on the DEC-20, each memory location is 36 bits in size, and on the Cyber 180, each location is 60 bits in size.

From a language designer's viewpoint, the primary difference between a byte-oriented architecture and a word-oriented one is that in a byte-oriented machine, as we mentioned, a single memory location typically can hold only one character value, whereas in a word-oriented machine, a single memory location can hold several characters. For example, on the DEC-20, each memory location can hold five 7-bit characters, with one bit left over. This may cause certain difficulties for a language designer if portability is a concern. Many of the examples in this text should work on any machine regardless of its memory architecture. However, some examples are designed for the more common byte-ori-