PORTABLE C SOFTWARE

PORTABLE C SOFTWARE

Mark R. Horton

AT&T Bell Laboratories
Columbus, Ohio

PRENTICE HALL
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

```
Horton, Mark.
    Portable C software / Mark Horton.
       p.   cm.
    Includes bibliographical references.
    ISBN 0-13-868050-7
    1. C (Computer program language)  2. Software compatibility.
  I. Title.
  QA76.73.C15H67  1990
  005.13'3--dc20                                           89-25578
                                                               CIP
```

Editorial/production supervision: **Brendan M. Stewart**
Cover design: **Lundgren Graphics Ltd.**
Cover photo: **Slide Graphics of New England**
Manufacturing buyer: **Ray Sintel**

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, NJ 07632

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-868050-7

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

4 [illegible]

PREFACE

The C language has become one of the most popular programming languages in use today. Many programs are written in C for the UNIX system, or for the MS DOS system, using only the distributed manuals as a guide.

I've come across an amazing variety of nonportable software. Often the software's author had no idea that the program wasn't portable, or even that portability was an issue. It's quite common, for example, to see a useful program posted to Usenet's `comp.sources.unix` newsgroup claimed to run "on the UNIX system." Someone else tries to compile it and finds out it doesn't work on their system. I have seen many statements along the lines of "This program is junk, because I tried to compile it and it said `getopt` was undefined." What has really happened is that the softwae's author was on UNIX System V, and the user has a Berkeley UNIX system with slightly different C libraries. There are just as many problems going in the other direction; for example, a program tries to call `select` and finds it undefined on System V.

The real cause of the problem is that the programmers are unaware that there are different versions of UNIX systems. Even if they are aware of different versions, they have no way to tell which features are present in every version and which features are specific to the system they have. The typical user's manual describes all features on an equal basis; unportable system extensions like Berkeley's symbolic links, or System V's `a641`, are documented exactly the same as portable features like `fopen`. Users have no way to tell whether they are using an unportable feature, so they assume everything is portable.

In recent years, it has become apparent that software written in C need not be confined to UNIX systems. A good deal of software written originally for the UNIX system has been ported to machines running the MS DOS system, and many C libraries for other machines have become complete enough to support many programs. I have written programs that ran on several operating systems. It's really not hard to design a program to be this portable.

It's much harder, on the other hand, to port a program that was written specifically for one machine, especially if it's badly written code with many hardware assumptions and bugs that were undetected on the original machine. (Programs that expect NULL pointers to point at a location containing a zero are quite common, and tend to work perfectly on some systems that put a zero in location zero. These programs abort on other systems.)

This book will make programmers aware of exactly which features are and are not portable. They can use this information when deciding which features to use. Sometimes a nonportable feature can be easily replaced with a portable one. Sometimes the nonportable feature can be used conditionally only if it is present. Sometimes the program can choose one of several possible ways to do the same thing, based on which one is supported. Sometimes the feature is absolutely required for the program, or the cost would be to great to duplicate the feature; in these cases the feature must be used, but the program can still be written modularly to allow a similar feature to be used at a later date.

This book addresses portability of programs written in the C language. This is not the same thing as portability of programs written for the UNIX system, although there is considerable overlap. I view the various standards as layers of an onion; the most general "least common denominator" standard is ANSI standard C, with POSIX (the IEEE P1003.1 Portable Operating System Interface Standard) being specific to UNIX and UNIX-like systems, and the SVID being specific to System V. (Reality is that ANSI C and POSIX will take years to catch on, and K&R C, UNIX System V, and 4.2BSD are more likely to be found as system bases in the meantime.)

The book will be useful to those concerned about portability within System V implementations, and to those writing portable code for POSIX systems, as well as those writing very portable code that runs on the UNIX system, MS DOS system, and other operating systems.

The emphasis is on avoiding duplication of development effort through source code portability. Issues such as binary machine language compatibility, documentation, and the format of binary files are usually less important, and are not addressed here.

This book concentrates on language features, rather than on specific versions of the UNIX system or C compilers. Many examples are taken from the V7, 4.2BSD, System V releases 2 and 3 UNIX systems, as well as Microsoft C and Lattice C for MS DOS. While future releases will add new features and blur the distinctions between the systems (System V release 4, for example, adds the Berkeley symbolic link feature) the intent is to keep the book from becoming dated by concentrating on the features, not the systems.

This book is intended both for professional programmers and for upper-division or graduate students to use as a textbook or supplementary text. It it not an introductory programming or introductory C language book; it assumes the reader already knows how to program in C. One semester of C programming may be sufficient background. More experienced programmers will find the book even more useful.

Several chapters of this book highlight issues about the design process, to ensure that a program can be written portably. The porting of existing, less portable programs is addressed. Many common problems are discussed, with different solutions and their advantages and disadvantages considered.

Each feature of the C environment is addressed, and its portability is rated. Widely implemented functions from the C library are discussed. System calls are also rated. External variables and macros are covered. User commands are discussed from the point of view of the C programmer, who must make use of compilation tools, makefiles, and shell scripts to compile and install each application, and may call commands from inside the C program.

This book is divided into four parts. Part I consists of introductory chapters. Part II contains four chapters giving advice about the correct way to write portable software, ''what to do.'' Part III is made up of four chapters describing many of the more common mistakes, ''what not to do.''

Part IV is a detailed reference examining several aspects of the C environment, and evaluating their portability. Chapter 11 lists the most widely available subroutines available in C libraries (traditionally, Section 3 of the UNIX system manual), rating their portability, describing the situations when each subroutine might or might not be available, and making suggestions about dealing with its absence. Chapter 12 examines operating system calls (traditionally, Section 2) in the same way. Chapter 13 covers header include files. Chapter 14 covers predefined variables in the C library. Chapter 15 rates UNIX system shell commands.

A number of appendices are also provided. The ANSI C standard lists a large number of potential portability problems, and their list is described and explained in Appendix A. Appendices B and C describe POSIX. and the AT&T System V Interface Definition (SVID), respectively. Appendices D ane E give advice for porting to MS DOS systems and 32-bit home computers such as the Apple Macintosh, Commodore Amiga, and Atari ST. Appendix F lists the source code for the public domain AT&T `getopt` function, so that programmers can feel free to incorporate its user interface into their applications without worrying about whether their system supports `getopt`. Appendix G is a table of functions, showing which of several representative systems support each function, and therefore how portable the function is. Appendix H is a similar table for shell commands.

I would like to thank the many people who have helped make this book a reality. Andy Duff encouraged me to go ahead with the original idea. The reviewers and editors have given me valuable feedback, although many of them have remained anonymous. I would especially like to thank John Blair, Clayton Elwell, Nancy Firestone, Chris Fox, Jeff Furrow, Steve Johnson, David Korn, Dwight Marshall, Elizabeth Marshburn, P. J. McCormac, J. Reilly, Dennis Ritchie, Clayton Rose, Eric Rosenthal, Chris Seiwald, Gary Sitzmann, Steve Sommars, Henry Spencer, Brendan Stewart, John Wait, J. Whiteside, and Thomas Williams.

CONTENTS

PART I
INTRODUCTION

CHAPTER 1: **WHY WRITE PORTABLE SOFTWARE?**

You will soon embark upon an enormous task. You must develop a program with product-quality code. The initial target machine has been selected, based on careful market studies. Getting the product running on the target machine is the top priority right now, and a lot of work will go into the finished system. But there are long-term plans to support the product on other systems. It would be nice to be able to get the program running on another machine with little effort. If the program is written portably in the first place, this work can be easily done with little extra development work on the initial system.

As with any software development effort, there is a huge amount of work to be done. It may take one or two years to develop. You don't know yet what processors will be available when the product is sent to the users. There are many different computer systems available, each with many users that represent potential customers. Each of these has very different hardware. This hardware is controlled by different operating systems. Some run the MS DOS system. Some run UNIX System V. Some run other AT&T releases of the UNIX system. Still others run operating systems derived from an AT&T UNIX system release, such as Berkeley's 4.2BSD.

In the future, the situation may be entirely different. New, bigger, and better machines will be available for less money. There may be customers who want the product ported to their existing hardware and are willing to pay a premium price for it. Processors difficult to imagine today may be available at an attractive price. The latest versions of popular operating systems may be very different from today's versions.

If you write code that assumes one particular processor running one specific release of the UNIX system, you may find yourself in five years with a large body of software that only runs on such a processor. The computing field moves very rapidly. Five years is a reasonable lifetime for a product, and the most successful programs are around much longer. Even though vendors make strong efforts to ensure that each release is upward compatible with the previous release, the environment can change considerably over a program's lifetime. Even if the older environment is still supported 5 or 10 years later, new features will almost certainly be available that users will want with their programs.

Suppose your project had made a commitment years ago to a locally enhanced derivative of an old release of the UNIX system and the 16-bit processor on which it ran. A few years later, you might have been struggling to port your code to a distributed environment of 32-bit processors running a later UNIX system release. The target system hardware and software would be obsolete in a few years. Another upgrade would be years off, because the pain of the port would be remembered.

A much smarter alternative would be to design your products to be portable in the first place. By expending a small amount of effort now, you can save yourself and your successors a lot of future work.

Portable code is also reusable code. Reusable code is valuable code. When a program is first written, the author usually has a particular combination of hardware, software, application, and user in mind. In a year or two, you may find a customer with a similar application requiring changes only to the user interface, or only to the hardware. Your code can be easily adapted if you avoid making unnecessary assumptions.

If you ensure that your program is portable while in the early stages of development, it is usually easier than a large-scale port at a later date. While the program is first being developed, it is fresh in your mind, and hence easier to change. Six months later, you will have forgotten the details of some coding trick, or why you made a particular decision. Sometimes even well-written comments are not enough to explain to you why a particular section of code is written in a way that may look strange.

It is important to do a port early in the development process to ensure that you won't make unportable assumptions that pervade the rest of your program. When you do your first port, you'll probably discover something you did that you thought was portable, but that turns out to be specific to one system. If you find this out early, you won't perpetuate unportable code and programming practices.

What Is Portability?

Some people believe that a portable program is one that can be carried to a new environment, compiled, and run, with no changes at all. More pragmatic is a definition from Steve Johnson. *A program is portable to the extent that it can be easily moved to a new computing environment with much less effort than would be required to write it afresh.* [1]

There are degrees of portability. A totally portable C program is one that plugs into any C environment and works without any changes. Such programs tend to be small. Many filters fit into this category. (A filter is a program that copies from standard input to standard output, making some transformation. For example, a program that translates from upper-case to lower-case could be a filter.)

Some programs are unportable. A program written entirely in assembly language is generally not useful on a different machine. A program with some machine-specific assumption scattered throughout the code is extremely unportable. An example might be a program containing many calls to `printf` with device-specific command sequences like

```
        printf("\033EYear\033H (Amount");
```

These are one terminal vendor's commands to clear the screen, print `Year`, address the cursor to row 0, column 8, and print `Amount`.

[1] S. C. Johnson and D. M. Ritchie. Portability of C Programs and the UNIX System, *Bell System Technical Journal*, Vol 57, No. 6, July—August 1978, page 2021. Definition originally from W. S. Brown.

Most programs fall somewhere in between. They may require some work to port, but do not need to be rewritten. A well-written program[2] can be ported by someone other than the original author. The porter should not have to understand more of the program than the part being changed.

Portability can be measured in many ways. One way is the amount of work necessary to port the program to a new environment. Another is the number of systems on which the program will run unchanged. (This may mean the number of systems to which the program has already been ported.) A third is the amount of system-specific code present in the program, for example, the number of `#ifdefs` (see Section 3.3); portable programs should require fewer sections of conditionally compiled code, and these sections should be concentrated in one source file.

A portable program should have only one master copy. If separate copies are maintained for each system, improvements to one copy are not automatically made to the other copies. If another person or group ports the program to another environment, or makes other enhancements, it is often useful if the original author (or other owner of the master copy) can "buy them back" by incorporating them into the master copy. This ensures that future versions of the program will port easily to the other environment.

A portable program is ported frequently. Before any major release of source code, the program is carried to each alternate environment and tested. Often this process turns up new bugs or unportable constructions that have been recently introduced.

Overall, a portable program is one that requires very little effort to keep running in different environments. The more effort necessary to keep it running on different systems, or the more effort spent by others because the program has not been ported to another environment, the less portable the program.

But Won't It Be a Lot of Work?

Perhaps you fear that writing the code portably will result in a lot of extra work. The extra effort, however, is usually quite small. Indeed, if it makes it too hard for you to write a program portably, it may not be worth it. Usually, no superhuman efforts are necessary; a little awareness of the issues, combined with experience and common sense, will do nicely. Even when significant extra effort is needed, it is often worthwhile in order to keep one version of the program that will run in different environments.

The most important part of portable programming is knowing which features can be safely used and which ones are specific to the particular implementation you are using. Unfortunately, most programmer's manuals give no clue; they imply that their implementation is the only one in the world, and that all features are equally portable.

A few manuals will include a footnote that indicates when a function is experimental, obsolete, or otherwise not recommended. Very few will tell you which features are their

[2] A *well-written program* is one that is modular, portable, easily modified, readable, efficient, meets the original requirements, and that the users are happy with.

own inventions. This makes it very difficult to write portable software using just one manual.

How Portable is Portable?

There are different degrees of portability. One program might run only on UNIX System V release 3 on a 3B2 processor. Another program might run on any UNIX System V on any 3B processor. Another might run on several versions of the UNIX system. A very portable program might run on the UNIX, MS DOS, and QNX systems. An extremely portable program might also run on systems with significantly different user-interface models, such as the Apple Macintosh, IBM OS/MVS, and Honeywell GCOS. (See Chapter 11 for definitions of degrees of portability used here.)

There are different sorts of features provided by operating systems. One system might support a windowing environment and a mouse. Another might provide only an ASCII terminal. A third might only have a half-duplex terminal or a card reader and printer. A mouse-oriented graphics application doesn't make sense on a card reader and printer.

This book uses certain precise terms to define degrees of portability. An *unportable* feature is specific to one particular system. A feature may be *portable among UNIX systems*, meaning that it is generally present on UNIX systems, but should not be expected to work on other operating systems. A *fairly portable* feature is present on many systems, but is missing from many others. A *very portable* feature is present on most, but not all, C implementations. An *extremely portable* feature is supported essentially everywhere.

Rating portability of programs is different than rating portability of features used by programs. One measure of a program's portability is the amount of effort needed to port it to another system. This will depend on the number and degree of unportable and somewhat portable features it uses. Another measure is more "turnkey" in nature: a program is as portable as the least portable feature it depends on without modifications to the source code.

A portable program should have only one master copy. If separate copies are maintained for each system, improvements to one copy are not automatically made to the other copies. If another person or group ports the program to another environment, or makes other enhancements, it is often useful if they are "bought back" by incorporating them into the master copy. This ensures that future versions of the program will port easily to the other environment.

It's important to decide how portable your application needs to be. Some features, such as a CRT screen or a network, may be absolutely required. Other features, such as a mouse or a special keyboard, may make your application more powerful, but the program is still useful on systems without them. Some decisions must be made about which features are critical; and about which features should be used only if present, and simulated or avoided otherwise.

Some environments, such as 8-bit home computers, may not offer a lot of functionality, but have a huge installed base. This may make it worthwhile to support a version of your program for the environment anyway. A program might make good use of the UNIX system and a large hard disk, but if you can make it run in 256K of RAM on a floppy-disk-based MS DOS environment, you've opened up a large market. A very large base of 8-bit machines exists in homes and schools, but many of these machines have only 48K or less of RAM. There are millions of home computers, but many of them don't even have a floppy disk drive. Some computers may have only a cassette drive, or a cartridge slot.

The proper way to make the decision is not to ask "*How much machine is needed to run my application?*" Instead, ask yourself, "*Can a stripped down version of my application run on this machine?*"

In the early days of screen oriented editors, it was generally felt that you needed a 9600-bits-per-second (BPS) hardwired connection, and an intelligent terminal with certain keys (such as arrow keys) to make a screen editor useful. Bill Joy, the original author of the UNIX system's `vi` screen editor, found himself with a 300-BPS modem and a "dumb" terminal at home, and wanted to use a screen editor. Rather than giving up, he found a way to make the `vi` editor useful at 300 BPS on such a terminal. The decision made was *given that I'm stuck with 300 BPS, would I rather use a line-oriented editor such as* `ed`, or a modified `vi`? Innovative features came from this challenge, such as cursor-motion optimization, repainting only the bottom 8 lines of the screen, and not updating the screen fully until the user gives a command to clean up the display. Within a few months, the `vi` editor was quite usable at 300 BPS on a "dumb" terminal.

By thinking creatively, it is often possible to stretch a program to work in an environment that was never originally thought possible. As a result of such thinking, `vi` is very popular on dialup lines today. Similar thinking made it possible to run `vi` on a printing terminal! Of course, you must temper the decision with other considerations, such as the resources you have for development. But plan to keep your options open in the future.