Software Portability

With Microcomputer Issues

Olivier Lecarme

Mireille Pellissier Gart

Mitchell Gart

Expanded Edition

Software Portability

With Microcomputer Issues

Olivier Lecarme
Laboratoire d'Informatique
Université de Nice, France

Mireille Pellissier Gart
Intermetrics
Cambridge, Massachusetts

Mitchell Gart
Alsys
Waltham, Massachusetts

Expanded Edition

McGraw-Hill Publishing Company

New York St. Louis San Francisco Auckland Bogotá
Caracas Hamburg Lisbon London Madrid Mexico
Milan Montreal New Delhi Oklahoma City
Paris San Juan São Paulo Singapore
Sydney Tokyo Toronto

Library of Congress Cataloging-in-Publication

Lecarme, Olivier
Software portability.

Bibliography: p.
Includes index.
1. Software compatibility. 2. Microcomputers—
Programming. I. Pillissier Gart, Mireille.
II. Gart, Mitchell. III. Title.
QA76.76.C64L43 1989 005.1 89-2284
ISBN 0-07-036964-X

This book was previously published by McGraw-Hill under the title *Software Portability.*

1234567890 DOC/DOC 894321098

ISBN 0-07-036964-X

The editors for this book were Theron Shreve and David E Fogarty, the designer was Naomi Auerbach, and the producing supervisor was Richard A. Ausburn It was set in Primer by TCSystems, Inc.

Printed and bound by R. R. Donnelley & Sons Company.

For more information about other McGraw-Hill materials, call 1-800-2-MCGRAW in the United States. In other countries, call your nearest McGraw-Hill office.

Preface

During the past three years, the relative importance of personal computers and workstations has continued to grow steadily. Although most of the methods, tools, and techniques presented in the book are equally applicable to personal computers, it seemed worthwhile to devote a complete new chapter in an expanded edition to this important part of the software industry.

Due to the lack of written materials on the subject, an original method was used. An inquiry was conducted, in the form of a mail and phone survey. Personal computer software vendors and users were questioned about their experiences in software portability, and their experiences were collected. This was done by Mitchell Gart, during the period between December 1986 and February 1987. Mitch is thus the main author of Chap. 8.

The other significant addition is the Appendix, "Practical Guide to Writing Portable Software." This section was also written mainly be Mitchell Gart, whose invaluable contribution to this expanded edition merits his inclusion as an additional author.

Chapters 1 to 7 have not been revised, although some shifts of emphasis between several minor points have occurred during the past three years. For example, the situation in software property rights (Sec. 2.4) has continued to evolve, punch cards and paper tape (Secs. 2.1.4 and 2.1.6) are more obsolete than they were in 1983, and Ada has emerged as an important language for writing portable software (Sec. 4.2.3).

Olivier Lecarme

Preface to First Edition

Software portability is an idea whose time has come. More and more often, potential software customers are asking whether a specific software product is portable. More and more often, advertisements and promotional literature use portability as a sales feature.

The fact that software has become more and more expensive in comparison with hardware costs has led to this state of affairs, heralded several years ago when vendors introduced "unbundling," or separate sales of hardware and software. Computer users often perceive this relatively recent development as a major advantage, because their investment in software no longer necessarily ties them to one hardware vendor, and portability is now considered one of the major features of software products.

It should also be noted that the increasing complexity of programs has more and more often required that they be written in higher-level languages. This factor has contributed greatly to (at least partial) program portability.

As frequently discussed as the notion of portability is, however, it carries many different meanings. Portability problems are often underestimated, partly because so many different aspects of computer science may affect portability. Thus, though a unifying synthesis seems especially desirable, it is presently lacking. Two books have been published about the subject: the collection of papers edited by P. J. Brown [Bro77a] and the small monograph by P. J. Wallis [Wal82]. The first book, because of its very nature, lacks unity, and it fails to treat some important aspects of the subject. It is becoming somewhat outdated. While the second book constitutes a good introduction, it is too short and too general to provide all the necessary information.

The purpose of this present book then is to answer the need that we feel for a relatively complete synthesis of software portability. Of course, we are fully aware of the imperfections of our work, and although we hope we have omitted no important facet of the subject, we also hope that our very extensive bibliography will help those readers who are eager for further study.

This book can be used in any of three ways. First, it can be read by computer users who want to self-study the subject. Whether they are application program-

iers, system programmers, or computer project managers, all interested readers hould find something of value in our text. There should be no part too difficult for a reader with a computer science degree or with equivalent experience.

Second, our book is designed to serve as a complementary textbook for a general course in software engineering, such as those found more and more often in undergraduate computer science curricula. Although it considers the vast subject of software engineering from a specific point of view, our text covers a large part of what is generally taught in this area.

And finally, our book is designed to serve as a basis for a specialized course on software portability. Such a course could occur at the graduate level in computer science curricula, probably following a general software engineering course. Or it could be used in a special seminar of one or two weeks. In such a course, lectures could emphasize new case studies, while our book could provide students with the necessary bases and references.

The idea of writing this book first occurred to us in 1980. Work in the software engineering group of the computer science laboratory at the University of Nice led to two doctoral theses [ThP77, Pel80] and two papers [LPT78, LPT82a]. We studied software portability as thoroughly as possible, and we came to regret the lack of a comprehensive textbook in the available literature. Thus, we proposed to give a course on this subject during the annual informatics summer school of AFCET, the French computer society. The idea was submitted in July 1980 and was accepted for the 1982 session of the school.

The first version of this text [LPT82b] was written as course support between December 1981 and June 1982. The course was given at the summer school in Namur, Belgium, as a sequence of eight 1½-hour lectures. Comments and criticisms from the audience, together with the experience we gained in giving the lectures, helped us to revise that first text and to enhance it with the addition of two important chapters containing case studies.

Thus the division of the present book into parts and chapters evolved rather slowly, the text itself passing through at least five stages. Broad organizational changes finally brought about a scheme that makes sequential reading of the text as easy as possible.

The most interesting and novel aspects of our work deal mainly with the collection under one cover of ordinarily scattered or inaccessible information and with the system of classification and clarification that we have devised. The size of the bibliography will give the reader an idea of the amount of documentation we examined and of the quantity of more-or-less relevant information we sifted in writing this book.

In its final state, this book is signed by only two authors. We wish to acknowledge especially, however, our debt to our colleague, Marie-Claude Thomas, of the University of Nice. She not only worked on the whole subject until 1981, but she attended all the discussions and working sessions that preceded the summer school at Namur. She wrote the French text of two very sensitive sections, those dealing with data portability (Sec. 2.3) and with software property rights (Sec. 2.4). She is presently working in a completely different research area and it is because of this, and only at her explicit request, that she does not appear as an author.

This book was first published in French [LeP84], then translated into English. This work was done by the authors themselves, with the invaluable aid of Mitchell Gart, Mireille's husband, who also wrote the first draft of the case study based on the Unix system (Sec. 7.2). Contributions by Mitchell and by Reba Krause to Sec. 2.4 were especially helpful in adapting this part of the text to the American environment.

Finally, we wish to thank the 1982 AFCET summer students, and especially the AFCET summer school director, George Stamon, as well as all our colleagues at the Informatics Laboratory at the University of Nice, at the Research Center of CII-Honeywell Bull in Louveciennes, France, and at Intermetrics in Cambridge, Massachusetts.

Olivier Lecarme

Mireille Pellissier Gart

Contents

Chapter

1

Introduction

Computer hardware is only as useful as its available software. It is very expensive, in time and in other resources, to re-create software for every new machine, and it is often desirable, from the user's point of view, for the software environment to be as similar as possible on different machines. Since the available software usually "hides" most or even all of the hardware from the user—that is, it is the software with which the user interacts—it is useful to try and implement the same familiar software on many different computers.

Thus, it is very frequently desirable, even at times necessary, to change the environment of any given software product to make it work on more than one computer and with more than one operating system. This change in environment we call "software transport" (or sometimes "software port"). A software product is more or less useful according to the relative ease of its "portability."

There are many cases in which software products, to be useful at all, must demonstrate portability. For instance, utility depends upon portability in the cases of software packages sold by a commercial vendor; a matrix inversion subprogram that is part of a library of scientific subprograms; computer games programmed and distributed by a club of hobbyists; a compiler for a new programming language, distributed by the authors of the language as a means of promoting their work; a mailing list distributed in machine-readable form. . . .

The list of examples goes on and on, but one can readily see the variety and complexity of the problems of portability involved. The purpose of this book is to demonstrate how most of these problems can be solved. The rest of this introductory chapter raises three important questions and provides some tentative answers.

1.1 Why should existing software be transported?

We shall often hereafter have occasion to note the differences in point of view between designers—those who build a software product so that it can be transported to other environments—and installers—those who install a software prod-

uct, built elsewhere, in a new environment. For now, we state the problem from the point of view of the installer; the point of view of the designer will be considered in the next section.

There are several possible answers a typical installer might have for the question asked in the title of this section. It might be said that:

- It takes less time and effort to use a program already built by somebody else than to attempt to rewrite it. This argument assumes that the transport operation itself takes less time than does the creation of a software product, thus freeing the programmer to do other work. Independent of the work saving, time will also be saved. (The distinction between time effectiveness as measured by work and by calendar is important: It is not likely that a programming job demanding one person-year could be done instead by 6 persons in two months; nor is it conceivable that such a job could be done by 52 persons in one week.)
- The designer of a software product is most likely a specialist in the field addressed by the product, while the installer is likely not a specialist but simply a user; or perhaps nobody with enough expertise to build such a product is available locally.
- The particular software product might be an excellent one, a standard in the field. (This assumes that the transport will not, as a side effect, severely damage the qualities of the product that made its transport so desirable in the first place.)
- It is advantageous to have always at hand, even when traveling, the same software environment to which one is accustomed. (This argument was originated by William Waite, who will be quoted many times in this book. It presumes a somewhat special situation, where the installer is also the designer.)
- A change of hardware (or operating system, or both) may be anticipated in the near future, and it would be desirable to continue using the same software products on the new hardware. (Reluctance to change, which sometimes causes stagnation, may have a more positive effect in this case.)
- Transporting an existing software product helps to guarantee conformity with what is done elsewhere. Programs will be compatible with other installations, and validity will be improved.
- In the case where the choice is between building one new program or choosing between several existing transportable programs, the result might be better with the wider choice.

Let us note that, of course, most of the above answers have negative counterparts that give rise to possible arguments for *not* transporting existing software. In all fairness, those counterarguments should be considered:

- If the software product has been designed specifically for portability, its performance may be inferior.
- No software product is immune from errors and weaknesses; to transport it is to perpetuate these shortcomings, perhaps even to increase them.

- Rewriting an existing software product, instead of using it as is, may be a good way to improve it. Often, when one builds and tunes a complex program, he or she learns from the experience; if then the program has to be rebuilt, it will inevitably be better.

These objections are serious, and it is not easy to answer them in a few words. But in the succeeding chapters we will give some fairly strong reasons why portable programming practices generally add to the overall quality of a program.

1.2 Why should we build portable software?

Taking here the point of view of the designer, we can suggest some answers to the question we ask in the title above:

- A software company builds and sells software: if its products are portable, they will appeal to a much wider market. Furthermore, instead of building new products for every possible computer environment, it is in the company's own interest to reduce the effort needed to change from one environment to another. The overall cost of developing a portable program once, and then transporting it several times, is bound to be advantageous compared with rewriting the same program several times.
- The designer as employee might contemplate changing employers and might want to reuse programs in the new environment. (This argument is another presentation of William Waite's argument.)
- Since software is becoming more expensive than hardware, its life cycle must be made longer; that is, it must be designed to survive hardware changes.
- In the case of a newly designed programming language, portability almost ensures broader, faster implementation.
- Further, the portability of the implementation will serve as a strong incentive to maintain compatibility in subsequent implementations.
- Even in the case of an established, well-known programming language, highly portable implementation will influence the shape of the official standard and promote the ideas included in the implementation.
- In an environment that comprises several different computer systems, the same application may be needed on each system, perhaps at the same time.
- A portable software product presents a positive sales argument.
- Anticipation that a software product will be portable is likely to have a beneficial effect on its programming: The style will have to be cleaner, more systematic, more disciplined, and more readable. Hence the reliability, adaptability, maintainability, and overall quality of the program will be improved.

Here again, of course, one can provide several counterarguments, most of them similar to those given in the preceding section:

- A portable software product may be more reliable and adaptable, but at the

same time it might be less efficient. (Still, we have to ask: Is it so important to produce an efficient but erroneous program?)

- If one of the main goals of a software company is to develop software for new customers and computer systems, and the software is *too* portable, the company's employees could conceivably be left with nothing to do. (Still, we have to point out that they would then be available for developing more new software.)
- Perhaps, as stated above, the overall cost to a company of developing portable software is less than that of rewriting the same program many times over, yet time or other resource constraints may require that the secondary goal, portability, be sacrificed so that the first version will be ready on time and/or within a fixed budget.
- If the implementation of a programming language is both easily transported and adapted, users may be encouraged to devise their own variations, thus negatively affecting compatibility and hampering the portability of programs written in this language.

1.3 What software is potentially portable?

By "software product" we mean here a set of programs with a common purpose. We will not attempt to provide a more precise definition (partially because that would require us to define precisely what a program is). But, we can give some typical examples of software products that can be built to be portable. These various examples will all appear again in the case studies found in Part 2 of this book.

- A compiler can be portable in its entirety if it generates an object language that is independent of the target computer. This is what we will call a "compiler-interpreter," of which a typical example is Snobol4 (see Sec. 6.3).
- A compiler can also be divided into two main parts, the front end depending on the source language, the back end on the object language. The interface between these two parts, if well designed, can be independent of both languages. In this case, the front end can be built to be portable (see Sec. 6.4).
- A large number of utility software products can be designed to be portable, at least in the major part. This is the case, for example, for text editors (see Sec. 7.5), general-purpose macroprocessors (see Secs. 3.1 and 7.4), various program-conversion tools (see Sec. 3.2), and tools for verifying and filtering a language (see Sec. 3.3). In fact, even file-management systems can be portable, in which case they further the portability of the above-mentioned utilities.
- Application software products that can be portable are still more numerous, and in fact their design constitutes the main reason for the existence of many software companies. We can mention, for example, company payroll, library management, stock or portfolio management, accounting, text processing, and the software packages and computer games that are increasingly popular for personal computers. But neither these applications nor the techniques used in

making them portable are usually described in scientific literature, and we have no case studies for them.

- Finally, and somewhat unexpectedly, whole operating systems are now built to be portable. The first systems of this sort were only demonstration models, but those that will be considered in Secs. 7.2 and 7.3 are full-scale systems now in actual use.

All things considered, a software product n.ay be designed to be portable if its purpose does not depend on the environment in which it will be implemented. By contrast, it is impossible to build a software product whose specifications refer to a specific environment and expect that product to be portable. This may be the case, for example, for the back end of a compiler, which generates machine language, or for a link editor or an assembler, the purpose of which is also to generate machine language. Let us note, however, that even in the cases just mentioned, an attempt is often made to design as much of the program as possible to be independent of the target machine; it is also true that we are beginning to develop the expertise to build portable code generators, link editors, and even assemblers.

Still it is not yet within the state of the art to build a portable peripheral handler, a supervisor nucleus (the part of an operating system that processes interrupts), or even a text-processing utility linked to a specific peripheral. There are still some cases where the major part of software development is too closely linked to a specific computer environment to permit portability in programming.

In addition, it must not be forgotten in many cases that it is simply not desirable to transport software products. Transport will be useless, or even of negative value, if (1) the initial qualities of the product do not warrant its broad use; (2) the transport would be too expensive; or (3) the expected performance after the transport would be calamitous.

Thus, we can summarize in one sentence: The transport of a software product can be considered for any product of satisfactory quality, independent of its environment, provided the costs involved warrant the transport.

Part

The Bases of Portability