Chapter 0
Front Matter

$Revision: 2.2 $
$Date: 1999/06/09 00:48:48 $

Dedication

To Tonia and Sarah, my women folk.

Introduction

Linkers and loaders have been part of the software toolkit almost as long as there have been computers, since they are the critical tools that permit programs to be built from modules rather than as one big monolith.

As early as 1947, programmers started to use primitive loaders that could take program routines stored on separate tapes and combine and relocate them into one program. By the early 1960s, these loaders had evolved into full-fledged linkage editors. Since program memory remained expensive and limited and computers were (by modern standards) slow, these linkers contained complex features for creating complex memory overlay structures to cram large programs into small memory, and for re-editing previously linked programs to save the time needed to rebuild a program from scratch.

During the 1970s and 1980s there was little progress in linking technology. Linkers tended to become even simpler, as virtual memory moved much of the job of storage management away from applications and overlays, into the operating system, and as computers became faster and disks larger, it became easier to recreate a linked program from scratch to replace a few modules rather than to re-link just the changes. In the 1990s linkers have again become more complex, adding support for modern features including dynamically linked shared libraries and the unusual demands of C++. Radical new processor architectures with wide instruction words and compiler-scheduled memory accesses, such as the Intel IA64, will also put new demands on linkers to ensure that the complex requirements of the code are met in linked prograsm.

Who is this book for?

This book is intended for several overlapping audiences.

- *Students:* Courses in compiler construction and operating systems have generally given scant treatment to linking and loading, often because the linking process seemed trivial or obvious. Although this was arguably true when the languages of interest were Fortran, Pascal, and C, and operating systems didn't use memory mapping or shared libraries, it's much less true now. C++, Java, and other object-oriented languages require a much more sophisticated linking environment. Memory mapped executable program, shared libraries, and dynamic linking affect many parts of an operating system, and an operating system designer disregards linking issues at his or her peril.

- *Practicing programmers* also need to be aware of what linkers do, again particularly for modern languages. C++ places unique demands on a linker, and large C++ programs are prone to develop hard-to-diagnose bugs due to unexpected things that happen at link time. (The best known are static constructors that run in an an order the programmer wasn't expecting.) Linker features such as shared libraries and dynamic linking offer great flexibility and power, when used appropriately,

- *Language designers and developers* need to be aware of what linkers do and can do as they build languages and compilers. Programming tasks had been handled hand for 30 years are automated in C++, depending on the linker to handle the details. (Consider what a programmer has to do to get the equivalent of C++ templates in C, or ensuring that the initialization routines in each of a hundred C source files are called before the body of the program starts.) Future languages will automate even more program-wide bookkeeping tasks, with more powerful linkers doing the work. Linkers will also be more involved in global program optimization, since the linker is the only stage of the compiler process that handles the entire program's code together and can do transformations that affect the entire program as a unit.

(The people who write linkers also all need this book, of course. But all the linker writers in the world could probably fit in one room and half of them already have copies because they reviewed the manuscript.)

Chapter summaries

Chapter 1, *Linking and Loading*, provides a short historical overview of the linking process, and discusses the stages of the linking process. It ends with a short but complete example of a linker run, from input object files to runnable ''Hello, world'' program.

Chapter 2, *Architectural Issues*, reviews of computer architecture from the point of view of linker design. It examines the SPARC, a representative reduced instruction set architecture, the IBM 360/370, an old but still very viable register-memory architecture. and the Intel x86, which is in a category of its own. Important architectural aspects include memory architecture, program addressing architecture, and the layout of address fields in individual instructions.

Chapter 3, *Object Files*, examines the internal structure of object and executable files. It starts with the very simplest files, MS-DOS .COM files, and goes on to examine progressively more complex files including, DOS EXE, Windows COFF and PE (EXE and DLL), Unix a.out and ELF, and Intel/Microsoft OMF.

Chapter 4, *Storage allocation*, covers the first stage of linking, allocating storage to the segments of the linked program, with examples from real linkers.

Chapter 5, *Symbol management*, covers symbol binding and resolution, the process in which a symbolic reference in one file to a name in a second file is resolved to a machine address.

Chapter 6, *Libraries*, covers object code libraries, creation and use, with issues of library structure and performance.

Chapter 7, *Relocation*, covers address relocation, the process of adjusting the object code in a program to reflect the actual addresses at which it runs. It also covers position independent code (PIC), code created in a way that avoids the need for relocation, and the costs and benefits of doing so.

Chapter 8, *Loading and overlays*, covers the loading process, getting a program from a file into the computer's memory to run. It also covers tree-structured overlays, a venerable but still effective technique to conserve address space.

Chapter 9, *Shared libraries*, looks at what's required to share a single copy of a library's code among many different programs. This chapter concentrates on static linked shared libraries.

Chapter 10, *Dynamic Linking and Loading*, continues the discussion of Chapter 9 to dynamically linked shared libraries. It treats two examples in detail, Windows32 dynamic link libraries (DLLs), and Unix/Linux ELF shared libraries.

Chapter 11, *Advanced techniques*, looks at a variety of things that sophisticated modern linkers do. It covers new features that C++ requires, including "name mangling", global constructors and destructors, template expansion, and duplicate code elimination. Other techniques include incremental linking, link-time garbage collection, link time code generation and optimization, load time code generation, and profiling and instrumentation. It concludes with an overview of the Java linking model, which is considerably more semantically complex than any of the other linkers covered.

Chapter 12, *References*, is an annotated bibliography.

The project

Chapters 3 through 11 have a continuing project to develop a small but functional linker in perl. Although perl is an unlikely implementation language for a production linker, it's an excellent choice for a term project. Perl handles many of the low-level programming chores that bog down programming in languages like C or C++, letting the student concentrate on the algorithms and data structures of the project at hand. Perl is available at no charge on most current computers, including Windows 95/98 and NT, Unix, and Linux, and many excellent books are available to teach perl to new users. (See the bibliography in Chapter 12 for some suggestions.)

The initial project in Chapter 3 builds a linker skeleton that can read and write files in a simple but complete object format, and subsequent chapters add functions to the linker until the final result is a full-fledged linker that supports shared libraries and produces dynamically linkable objects.

Perl is quite able to handle arbitrary binary files and data structures, and the project linker could if desired be adapted to handle native object formats.

Acknowledgements

Many, many, people generously contributed their time to read and review the manuscript of this book, both the publisher's reviewers and the readers of the comp.compilers usenet newsgroup who read and commented on an on-line version of the manuscript. They include, in alphabetical order, Mike Albaugh, Rod Bates, Gunnar Blomberg, Robert Bowdidge, Keith Breinholt, Brad Brisco, Andreas Buschmann, David S. Cargo, John Carr, David Chase, Ben Combee, Ralph Corderoy, Paul Curtis, Lars Duening, Phil Edwards, Oisin Feeley, Mary Fernandez, Michael Lee Finney, Peter H. Froehlich, Robert Goldberg, James Grosbach, Rohit Grover, Quinn Tyler Jackson, Colin Jensen, Glenn Kasten, Louis Krupp, Terry Lambert, Doug Landauer, Jim Larus, Len Lattanzi, Greg Lindahl, Peter Ludemann, Steven D. Majewski, John McEnerney, Larry Meadows, Jason Merrill, Carl Montgomery, Cyril Muerillon, Sameer Nanajkar, Jacob Navia, Simon Peyton-Jones, Allan Porterfield, Charles Randall, Thomas David Rivers, Ken Rose, Alex Rosenberg, Raymond Roth, Timur Safin, Kenneth G Salter, Donn Seeley, Aaron F. Stanton, Harlan Stenn, Mark Stone, Robert Strandh, Bjorn De Sutter, Ian Taylor, Michael Trofimov, Hans Walheim, and Roger Wong.

These people are responsible for most of the true statements in the book. The false ones remain the author's responsiblity. (If you find any of the latter, please contact me at the address below so they can be fixed in subsequent printings.)

I particularly thank my editors at Morgan-Kaufmann Tim Cox and Sarah Luger, for putting up with my interminable delays during the writing process, and pulling all the pieces of this book together.

Contact us

This book has a supporting web site at `http://linker.iecc.com`. It includes example chapters from the book, samples of perl code and object files for the project, and updates and errata.

You can send e-mail to the author at `linker@iecc.com`. The author reads all the mail, but because of the volume received may not be able to answer all questions promptly.