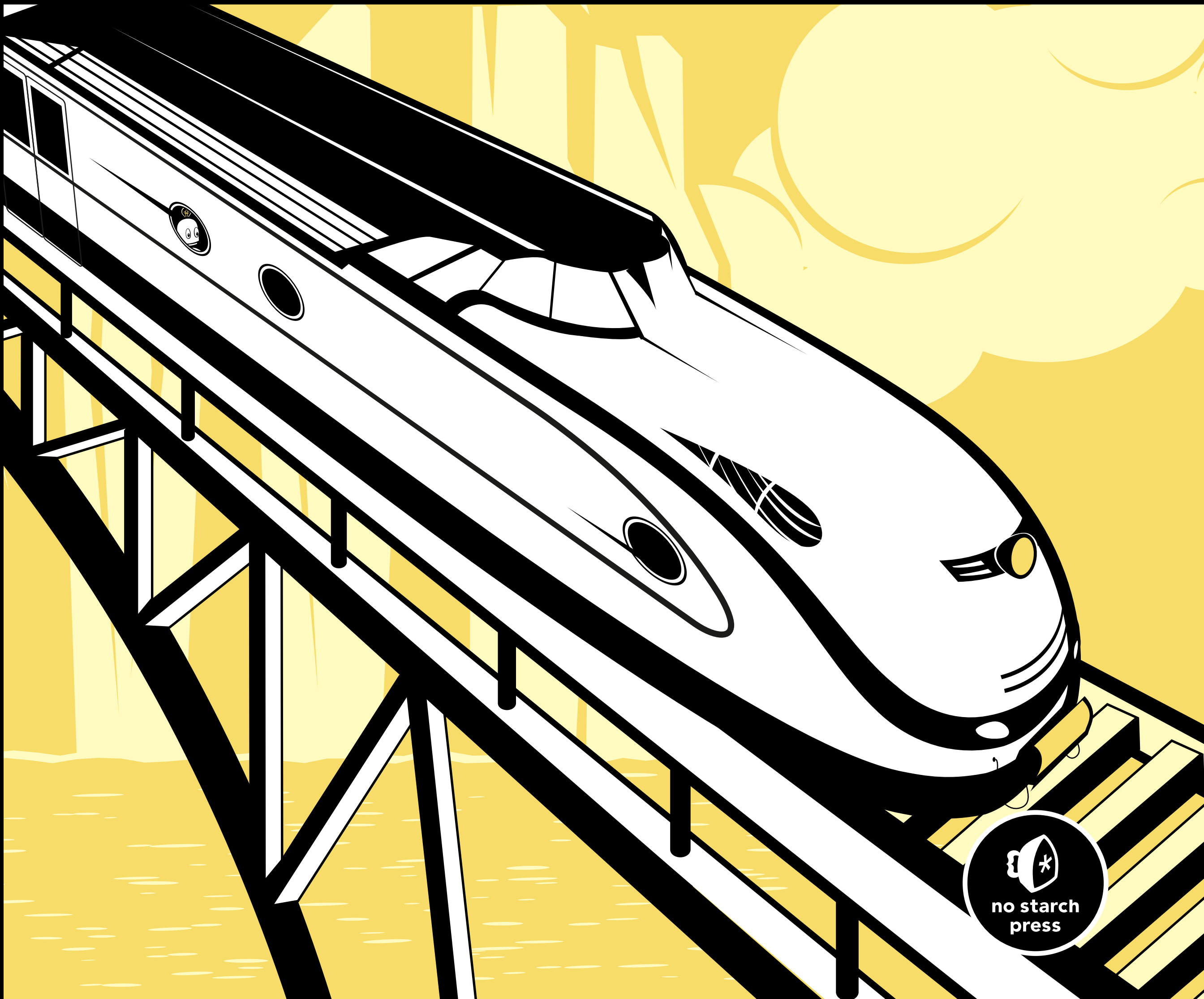
2ND EDITION

WRITE GREAT CODE / VOLUME 2

THINKING LOW LEVEL, WRITING HIGH LEVEL

RANDALL HYDE



no starch press

PRAISE FOR THE FIRST EDITION OF *WRITE GREAT CODE, VOLUME 2*

WRITE GREAT CODE

VOLUME 2

2ND EDITION

Thinking Low-Level, Writing High-Level

by Randall Hyde

no starch press

San Francisco

WRITE GREAT CODE, Volume 2: Thinking Low-Level, Writing High-Level, 2nd Edition.
Copyright © 2020 by Randall Hyde.

ISBN-10: 1-718-50038-6
ISBN-13: 978-1-71850-038-9

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editor: Rachel Monaghan
Developmental Editor: Athabasca Witschi
Project Editor: Dapinder Dosanjh
Cover and Interior Design: Octopod Studios
Technical Reviewer: Anthony Tribelli
Copyeditor: Rachel Monaghan
Compositor: Danielle Foster
Proofreader: James Fraleigh
Illustrator: David Van Ness

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

The Library of Congress issued the following Cataloging-in-Publication Data for the first edition of Volume 1:

```
Hyde, Randall.
  Write great code : understanding the machine / Randall Hyde.
       p. cm.
  ISBN 1-59327-003-8
1. Computer programming.  2. Computer architecture.  I. Title.
   QA76.6.H94 2004
   005.1--dc22
                                                    2003017502
```

About the Author

Randall Hyde is the author of *The Art of Assembly Language* and *Write Great Code, Volumes 1, 2, and 3* (all from No Starch Press), as well as *Using 6502 Assembly Language* and *P-Source* (Datamost). He is also the coauthor of *Microsoft Macro Assembler 6.0 Bible* (The Waite Group). Over the past 40 years, Hyde has worked as an embedded software/hardware engineer developing instrumentation for nuclear reactors, traffic control systems, and other consumer electronics devices. He has also taught computer science at California State Polytechnic University, Pomona, and at the University of California, Riverside. His website is *www.randallhyde.com/*.

About the Technical Reviewer

Tony Tribelli has more than 35 years of experience in software development, including work on embedded device kernels and molecular modeling. He developed video games for 10 years at Blizzard Entertainment. He is currently a software development consultant and privately develops applications utilizing computer vision.

BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

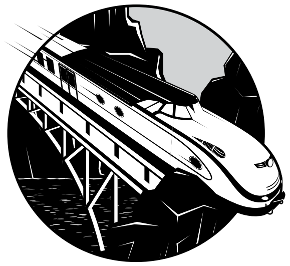
Many people have read and reread every word, symbol, and punctuation mark in this book in order to produce a better result. Kudos to the following people for their careful work on the second edition: development editor Athabasca Witschi, copyeditor/production editor Rachel Monaghan, and proofreader James Fraleigh.

I would like to take the opportunity to graciously thank Anthony Tribelli, a longtime friend, who went well beyond the call of duty when doing a technical review of this book. He pulled every line of code out of this book (including snippets) and compiled and ran it to make sure it worked properly. His suggestions and opinions throughout the technical review process have dramatically improved the quality of this work.

Of course, I would also like to thank all the countless readers over the years who've emailed suggestions and corrections, many of which have found their way into this second edition.

Thanks to all of you,
Randall Hyde

INTRODUCTION



What do we mean by *great code*? Different programmers will have different opinions. Therefore, it is impossible to provide an all-encompassing definition that will satisfy everyone. Here is the definition this book will use:

> Great code is software that is written using a consistent and prioritized set of good software characteristics. In particular, great code follows a set of rules that guide the decisions a programmer makes when implementing an algorithm as source code.

However, as I noted in *Write Great Code, Volume 1: Understanding the Machine* (hereafter, *WGC1*), there are some attributes of great code that nearly everyone can agree on. Specifically, great code:

- Uses the CPU efficiently (that is, it's fast)
- Uses memory efficiently (that is, it's small)
- Uses system resources efficiently
- Is easy to read and maintain
- Follows a consistent set of style guidelines
- Uses an explicit design that follows established software engineering conventions
- Is easy to enhance
- Is well tested and robust (that is, it works)
- Is well documented

We could easily add dozens of items to this list. Some programmers, for example, may feel that great code must be portable, must follow a given set of programming style guidelines, or must be written in a certain language (or *not* be written in a certain language). Some may feel that great code must be written as simply as possible, while others believe that it must be written quickly. Still others may feel that great code is created on time and under budget.

Given that there are so many aspects of great code—too many to describe properly in a single book—this second volume of the *Write Great Code* series concentrates primarily on one: efficient performance. Although efficiency might not always be the primary goal of a software development effort—nor does it have to be for code to qualify as great—people generally agree that inefficient code is *not* great code. And inefficiency is one of the major problems with modern applications, so it's an important topic to emphasize.

Performance Characteristics of Great Code

As computer system performance has increased from megahertz to hundreds of megahertz to gigahertz, computer software performance has taken a back seat to other concerns. Today, it's not at all uncommon for software engineers to exclaim, "You should never optimize your code!" Funny, you don't hear too many software *users* making such statements.

Although this book describes how to write efficient code, it's not a book about optimization. *Optimization* is a phase near the end of the Software Development Life Cycle (SDLC) in which software engineers determine why their code does not meet performance specifications and then refine it accordingly. But unfortunately, if they don't put any thought into the application's performance until the optimization phase, it's unlikely that optimization will prove practical. The time to ensure that an application meets reasonable performance benchmarks is at the *beginning* of the SDLC, during the design and implementation phases. Optimization can fine-tune a system's performance, but it can rarely deliver a miracle.

Although the quote is often attributed to Donald Knuth, who popularized it, it was Tony Hoare who originally said, "Premature optimization is the root of all evil." This statement has long been the rallying cry of software engineers who neglect application performance until the very end of the SDLC—at which point optimization is typically ignored for economic or time-to-market reasons. However, Hoare did not say, "Concern about application performance during the early stages of an application's development is the root of all evil." He specifically said *premature optimization*, which, back then, meant counting cycles and instructions in assembly language code—not the type of coding you want to do during initial program design, when the code base is fluid. Thus, Hoare's comments were on the mark.

The following excerpt from a short essay by Charles Cook (*https://bit.ly/38NhZkT*) further describes the problem with reading too much into Hoare's statement:

> I've always thought this quote has all too often led software designers into serious mistakes because it has been applied to a different problem domain to what was intended.
>
> The full version of the quote is "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." and I agree with this. It's usually not worth spending a lot of time micro-optimizing code before it's obvious where the performance bottlenecks are. But, conversely, when designing software at a system level, performance issues should always be considered from the beginning. A good software developer will do this automatically, having developed a feel for where performance issues will cause problems. An inexperienced developer will not bother, misguidedly believing that a bit of fine tuning at a later stage will fix any problems.

Indeed, Hoare was saying that software engineers should worry about other issues, like good algorithm design and implementation, before they worry about traditional optimizations, like how many CPU cycles a particular statement requires for execution.

Although you could certainly apply many of this book's concepts during an optimization phase, most of the techniques here really need to be applied during the initial coding. An experienced software engineer may argue that doing so produces only minor improvements in performance. In some cases, this is true—but keep in mind that these minor effects accumulate. If you put off these ideas until you reach "code complete," it's unlikely that they'll ever find their way into your software. It's just too much work to implement them after the fact (and too risky to make such changes to otherwise working code).

The Goal of This Book

This book (and *WGC1*) attempts to fill the gaps in the education of the current generation of programmers so they can write quality code. In particular, it covers the following concepts:

- Why it's important to consider the low-level execution of your high-level programs
- How compilers generate machine code from high-level language (HLL) statements
- How compilers represent various data types using low-level, primitive data types
- How to write your HLL code to help the compiler produce better machine code
- How to take advantage of a compiler's optimization facilities
- How to "think" in assembly language (low-level terms) while writing HLL code

This book will teach you how to choose appropriate HLL statements that translate into efficient machine code with a modern optimizing compiler. In most cases, different HLL statements provide many ways to achieve a given result, some of which, at the machine level, are naturally more efficient than others. Though there may be a very good reason for choosing a less efficient statement sequence over a more efficient one (for example, readability), the truth is that most software engineers have no idea about HLL statement runtime costs and thus are unable to make an educated choice. The goal of this book is to change that.

Again, this book is not about choosing the most efficient statement sequence no matter what. It is about understanding the cost of various HLL constructs so that, when faced with multiple options, you can make an informed decision about which sequence is most appropriate to use.

Chapter Organization

Though you don't need to be an expert assembly language programmer in order to write efficient code, you'll need at least a basic knowledge of it to understand the compiler output in this book. Chapters 1 and 2 discuss several aspects of learning assembly language, covering common misconceptions, considerations around compilers, and available resources. Chapter 3 provides a quick primer for 80x86 assembly language. Online appendixes (*http://www.randallhyde.com/*) provide primers for the PowerPC, ARM, Java bytecode, and Common Intermediate Language (CIL) assembly languages.

In Chapters 4 and 5, you'll learn about determining the quality of your HLL statements by examining compiler output. These chapters describe disassemblers, object code dump tools, debuggers, various HLL compiler options for displaying assembly language code, and other useful software tools.

The remainder of the book, Chapters 6 through 15, describes how compilers generate machine code for different HLL statements and data types. Armed with this knowledge, you'll be able to choose the most appropriate data types, constants, variables, and control structures to produce efficient applications.

Assumptions and Prerequisites

This book was written with certain assumptions about your prior knowledge. You'll reap the greatest benefit from this material if your personal skill set matches the following:

- You should be reasonably competent in at least one imperative (procedural) or object-oriented programming language. This includes C and C++, Pascal, Java, Swift, BASIC, Python, and assembly, as well as languages like Ada, Modula-2, and FORTRAN.
- You should be capable of taking a small problem description and working through the design and implementation of a software solution for that problem. A typical semester or quarter course at a college or university (or several months of experience on your own) should be sufficient preparation.
- You should have a basic grasp of machine organization and data representation. You should know about the hexadecimal and binary numbering systems. You should understand how computers represent various high-level data types such as signed integers, characters, and strings in memory. Although the next couple of chapters provide a primer on machine language, it would help considerably if you've picked up this information along the way. *WGC1* fully covers the subject of machine organization if you feel your knowledge in this area is a little weak.

The Environment for This Book

Although this book presents generic information, parts of the discussion will necessarily be system specific. Because the Intel Architecture PCs are, by far, the most common in use today, that's the platform I'll use when discussing specific system-dependent concepts in this book. However, those concepts still apply to other systems and CPUs—such as the PowerPC CPU in the older Power Macintosh systems, ARM CPUs in mobile phones, tablets and single-board computers (SBCs; like the Raspberry Pi or higher-end Arduino boards), and other RISC CPUs in a Unix box—although you may need to research the particular solution for an example on your specific platform.

Most of the examples in this book run under macOS, Windows, and Linux. When creating the examples, I tried to stick with standard library interfaces to the OS wherever possible and make OS-specific calls only when the alternative was to write "less than great" code.

Most of the specific examples in this text will run on a late-model Intel Architecture (including AMD) CPU under Windows, macOS, and Linux, with a reasonable amount of RAM and other system peripherals normally found on a modern PC. The concepts, if not the software itself, will apply to Macs, Unix boxes, SBCs, embedded systems, and even mainframes.

For More Information

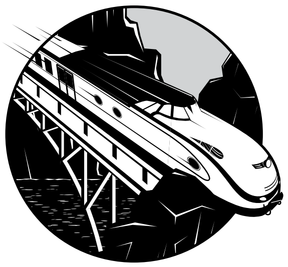
Mariani, Rico. "Designing for Performance." December 11, 2003. *https://docs.microsoft.com/en-us/archive/blogs/ricom/designing-for-performance/*.

Wikipedia. "Program Optimization." *https://en.wikipedia.org/wiki/Program_optimization*.

1

THINKING LOW-LEVEL, WRITING HIGH-LEVEL

If you want to write the best high-level language code,
learn assembly language.
—Common programming advice



This book doesn't teach anything revolutionary. Rather, it describes a time-tested, well-proven approach to writing great code—understanding how the code you write will actually execute on a real machine. The journey to that understanding begins with this chapter. In it, we'll explore these topics:

- Misconceptions programmers have about the code quality produced by typical compilers
- Why learning assembly language is still a good idea
- How to think in low-level terms while writing HLL code

So, without further ado, let's begin!

1.1 Misconceptions About Compiler Quality

In the early days of the personal computer revolution, high-performance software was written in assembly language. As time passed, optimizing compilers for high-level languages were improved, and their authors began claiming that the performance of compiler-generated code was within 10 to 50 percent of hand-optimized assembly code. Such proclamations ushered the ascent of HLLs for PC application development, sounding the death knell for assembly language. Many programmers began citing statistics like "my compiler achieves 90 percent of assembly's speed, so it's insane to use assembly language." The problem is that they never bothered to write hand-optimized assembly versions of their applications to check their claims. Often, their assumptions about their compiler's performance were wrong. Worse still, as compilers for languages such as C and C++ matured to the point that they were producing very good output code, programmers began favoring more high-level languages, such as Java, Python, and Swift, that were either interpreted (or semi-interpreted) or had very immature code generators producing terrible output code.

The authors of optimizing compilers weren't lying. Under the right conditions, an optimizing compiler *can* produce code that is almost as good as hand-optimized assembly language. However, the HLL code has to be written in an appropriate way to achieve these performance levels. Writing HLL code in this manner requires a firm understanding of how computers operate and execute software.

1.2 Why Learning Assembly Language Is Still a Good Idea

When programmers first began giving up assembly language in favor of using HLLs, they generally understood the low-level ramifications of the HLL they were using and could choose their HLL statements appropriately. Unfortunately, the generation of computer programmers that followed them did not have the benefit of mastering assembly language. As a result, they were not equipped to wisely choose statements and data structures that HLLs could efficiently translate into machine code. Their applications, if measured against the performance of a comparable hand-optimized assembly language program, surely proved inferior.

Veteran programmers who recognized this problem offered sage advice to the new programmers: "If you want to learn how to write good HLL code, you need to learn assembly language." By learning assembly language, programmers can understand the low-level implications of their code and make informed decisions about the best way to write applications in an HLL.[1] Chapter 2 will discuss assembly language further.

1. A knowledge of assembly may also come in handy for debugging in that programmers could examine the assembly instructions and registers to see where the HLL code went wrong.

1.3 Why Learning Assembly Language Isn't Absolutely Necessary

While it's a good idea for any well-rounded programmer to learn to program in assembly language, it isn't a necessary condition for writing great, efficient code. What's most important is to understand how HLLs translate statements into machine code so that you can choose appropriate HLL statements. And while one way to do this is to become an expert assembly language programmer, that approach requires considerable time and effort.

The question, then, is, "Can a programmer just study the low-level nature of the machine and improve the HLL code they write without becoming an expert assembly programmer in the process?" The answer, given the preceding point, is a qualified yes. That's the purpose of this book: to teach you what you need to know to write great code without having to become an expert assembly language programmer.

1.4 Thinking Low-Level

When Java was first becoming popular in the late 1990s, the language received complaints like the following:

> Java's interpreted code is forcing me to take a lot more care when writing software; I can't get away with using linear searches the way I could in C/C++. I have to use good (and more difficult to implement) algorithms like binary search.

Such statements demonstrate the major problem with using optimizing compilers: they allow programmers to get lazy. Although optimizing compilers have made tremendous strides over the past several decades, none of them can make up for poorly written HLL source code.

Of course, many novice HLL programmers read about how marvelous the optimization algorithms are in modern compilers and assume that the compiler will produce efficient code regardless of what it's fed. However, that's not the case: although compilers can do a great job of translating well-written HLL code into efficient machine code, poorly written source code stymies the compiler's optimization algorithms. In fact, it's not uncommon to hear C/C++ programmers praising their compiler, never realizing how poor a job it's actually doing because of how they've written their code. The problem is that they've never actually looked at the machine code the compiler produces from their HLL source code. They assume that the compiler is doing a great job because they've been told that compilers produce code that's almost as good as what an expert assembly language programmer can produce.

1.4.1 Compilers Are Only as Good as the Source Code You Feed Them

It goes without saying that a compiler won't change your algorithms in order to improve your software's performance. For example, if you use a linear search rather than a binary search, you can't expect the compiler to use a better algorithm for you. Certainly, the optimizer may improve the speed of your linear search by a constant factor (for example, double or triple the speed of your code), but this improvement may be nothing compared to using a better algorithm. In fact, it's very easy to show that, given a sufficiently large database, a binary search processed by an interpreter with no optimization will run faster than a linear search algorithm processed by the best compiler.

1.4.2 How to Help the Compiler Produce Better Machine Code

Let's assume that you've chosen the best possible algorithm(s) for your application and you've spent the extra money to get the best compiler available. Is there something you can do to write HLL code that is more efficient than you would otherwise produce? Generally, the answer is yes.

One of the best-kept secrets in the compiler world is that most compiler benchmarks are rigged. Most real-world compiler benchmarks specify an algorithm to use, but they leave it up to the compiler vendors to actually implement the algorithm in their particular language. These compiler vendors generally know how their compilers behave when fed certain code sequences, so they will write the code sequence that produces the best possible executable.

Some may feel that this is cheating, but it's really not. If a compiler is capable of producing that same code sequence under normal circumstances (that is, the code generation trick wasn't developed specifically for the benchmark), there's nothing wrong with showing off its performance. And if the compiler vendor can pull little tricks like this, so can you. By carefully choosing the statements you use in your HLL source code, you can "manually optimize" the machine code the compiler produces.

Several levels of manual optimization are possible. At the most abstract level, you can optimize a program by selecting a better algorithm for the software. This technique is independent of the compiler and the language.

Dropping down a level of abstraction, the next step is to manually optimize your code based on the HLL that you're using while keeping the optimizations independent of the particular implementation of that language. While such optimizations may not apply to other languages, they should apply across different compilers for the same language.

Dropping down yet another level, you can start thinking about structuring the code so that the optimizations are applicable only to a certain vendor or perhaps only to a specific version of a compiler.

Finally, at perhaps the lowest level, you can consider the machine code that the compiler emits and adjust how you write statements in an HLL to force the compiler to generate some sequence of machine instructions. The Linux kernel is an example of this approach. Legend has it that the kernel developers were constantly tweaking the C code they wrote in the Linux kernel in order to control the 80x86 machine code that the GNU C Compiler (GCC) was producing.

Although this development process may be a bit overstated, one thing is for sure: programmers who employ it will produce the best possible machine code from a compiler. This is the type of code that's comparable to what decent assembly language programmers produce, and the kind of compiler output that HLL programmers like to cite when arguing that compilers produce code that's comparable to handwritten assembly. The fact that most people do not go to these extremes to write their HLL code never enters into the argument. Nevertheless, the fact remains that carefully written HLL code can be nearly as efficient as decent assembly code.

Will compilers ever produce code that is as good as or better than what an expert assembly language programmer can write? The correct answer is no; after all, an expert assembly language programmer can always look at a compiler's output and improve on that. However, careful programmers writing code in HLLs like C/C++ can come close if they write their program such that the compiler can easily translate it into efficient machine code. Thus, the real question is, "How do I write my HLL code so that the compiler can translate it most efficiently?" Well, answering that question is the subject of this book. But the short answer is, "Think in assembly; write in a high-level language." Let's take a quick look at how to do that.

1.4.3 How to Think in Assembly While Writing HLL Code

HLL compilers translate statements in that language to a sequence of one or more machine language (or assembly language) instructions. The amount of space in memory that an application consumes, and the amount of time that an application spends in execution, are directly related to the number and type of machine instructions that the compiler emits.

However, the fact that you can achieve the same result with two different code sequences in an HLL does not imply that the compiler generates the same sequence of machine instructions for each approach. The HLL `if` and `switch`/`case` statements are classic examples. Most introductory programming texts suggest that a chain of `if-elseif-else` statements is equivalent to a `switch`/`case` statement. Consider the following trivial C example:

```
switch( x )
    {
        case 1:
            printf( "X=1\n" );
            break;

        case 2:
            printf( "X=2\n" );
            break;

        case 3:
            printf( "X=3\n" );
            break;

        case 4:
            printf( "X=4\n" );
            break;
        default:
            printf( "X does not equal 1, 2, 3, or 4\n" );
    }

/* equivalent if statement */

    if( x == 1 )
        printf( "X=1\n" );
    else if( x== 2 )
        printf( "X=2\n" );
    else if( x==3 )
        printf( "X=3\n" );
    else if( x==4 )
        printf( "X=4\n" );
    else
        printf( "X does not equal 1, 2, 3, or 4\n" );
```

Although these two code sequences might be semantically equivalent (that is, they compute the same result), there is no guarantee that the compiler will generate the same sequence of machine instructions for both.

Which one will be better? Unless you understand how the compiler translates statements like these into machine code, and have a basic knowledge of the different efficiencies between various machines, you probably can't answer that. Programmers who fully understand how a compiler will translate these two sequences can evaluate both and then judiciously choose one based on the quality of the expected output code.

By thinking in low-level terms when writing HLL code, a programmer can help an optimizing compiler approach the level of code quality achieved by hand-optimized assembly language code. Sadly, the converse is usually true as well: if a programmer does not consider the low-level ramifications of their HLL code, the compiler will rarely generate the best possible machine code.

1.5 Writing High-Level

One problem with thinking in low-level terms while writing high-level code is that it's almost as much work to write HLL code this way as it is to write assembly code. This negates many of the familiar benefits of writing programs in HLLs, such as faster development time, better readability, and easier maintenance. If you're sacrificing the benefits of writing applications in an HLL, why not simply write them in assembly language to begin with?

As it turns out, thinking in low-level terms won't lengthen your overall project schedule as much as you would expect. Although it does slow down the initial coding process, the resulting HLL code will still be readable and portable, and it will maintain the other attributes of well-written, great code. But more importantly, it will also gain some efficiency that it wouldn't otherwise have. Once the code is written, you won't have to constantly think about it in low-level terms during the maintenance and enhancement phases of the Software Development Life Cycle (SDLC). In short, thinking in low-level terms during the initial software development stage retains the advantages of both low-level and high-level coding (efficiency plus ease of maintenance) without the corresponding disadvantages.

1.6 Language-Neutral Approach

Although this book assumes you are conversant in at least one imperative language, it is not entirely language specific; its concepts transcend whatever programming language(s) you're using. To help make the examples more accessible, the programming examples we'll use will rotate among several languages, such as C/C++, Pascal, BASIC, Java, Swift, and assembly. When presenting examples, I'll explain exactly how the code operates so that even if you're unfamiliar with the specific programming language, you'll be able to understand its operation by reading the accompanying description.

This book uses the following languages and compilers in various examples:

- **C/C++:** GCC and Microsoft's Visual C++
- **Pascal:** Borland's Delphi, and Free Pascal
- **Assembly language:** Microsoft's MASM, HLA (High-Level Assembly), and Gas (the GNU Assembler)
- **Basic:** Microsoft's Visual Basic

If you're not comfortable working with assembly language, don't worry: the primer on 80x86 assembly language and the online reference (*http://www.writegreatcode.com/*) will allow you to read compiler output. If you'd like to extend your knowledge of assembly language, see the resources listed at the end of this chapter.

1.7 Additional Tips

No single book can completely cover everything you need to know in order to write great code. This book, therefore, concentrates on the areas that are most pertinent for writing great software, providing the 90 percent solution for those who are interested in writing the best possible code. To get that last 10 percent you'll need additional help. Here are some suggestions:

Become an expert assembly language programmer. Fluency in at least one assembly language will fill in many missing details that you just won't get from this book. As noted, the purpose of this book is to teach you how to write the best possible code *without* actually becoming an assembly language programmer. However, the extra effort will improve your ability to think in low-level terms.

Study compiler construction theory. Although this is an advanced topic in computer science, there's no better way to understand how compilers generate code than to study the theory behind compilers. While there's a wide variety of textbooks on this subject, many of them require considerable prerequisite knowledge. Carefully review any book before you purchase it in order to determine if it was written at an appropriate level for your skill set. You can also search online to find some excellent web tutorials.

Study advanced computer architecture. Machine organization and assembly language programming are a subset of the study of computer architecture. While you may not need to know how to design your own CPUs, studying computer architecture may help you discover additional ways to improve the HLL code that you write.

1.8 For More Information

Duntemann, Jeff. *Assembly Language Step-by-Step.* 3rd ed. Indianapolis: Wiley, 2009.

Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach.* 5th ed. Waltham, MA: Morgan Kaufmann, 2012.

Hyde, Randall. *The Art of Assembly Language.* 2nd ed. San Francisco: No Starch Press, 2010.

2

SHOULDN'T YOU LEARN ASSEMBLY LANGUAGE?

Although this book will teach you how to write better code without mastering assembly language, the absolute best HLL programmers do know assembly, and that knowledge is one of the reasons they write great code. As mentioned in Chapter 1, although this book can provide a 90 percent solution if you just want to write great HLL code, to fill in that last 10 percent you'll need to learn assembly language. While teaching you assembly language is beyond the scope of this book, it's still an important subject to discuss. To that end, this chapter will explore the following:

- The problem with learning assembly language
- High-level assemblers and how they can make learning assembly language easier

- How you can use real-world products like Microsoft Macro Assembler (MASM), Gas (Gnu Assembler), and HLA (High-Level Assembly) to easily learn assembly language programming
- How an assembly language programmer *thinks* (that is, the assembly language programming paradigm)
- Resources available to help you learn assembly language programming

2.1 Benefits and Roadblocks to Learning Assembly Language

Learning assembly language—*really* learning assembly language—offers two benefits. First, you'll gain a complete understanding of the machine code that a compiler can generate. By mastering assembly language, you'll achieve the 100 percent solution just described and be able to write better HLL code. Second, you'll be able to code critical parts of your application in assembly language when your HLL compiler is incapable, even with your help, of producing the best possible code. Once you've absorbed the lessons of the following chapters to hone your HLL skills, moving on to learn assembly language is a very good idea.

There's one catch to learning assembly language, though. In the past, it's been a long, difficult, and frustrating task. The assembly language programming paradigm is sufficiently different from HLL programming that most people feel like they're starting over from square one when learning it. It's very frustrating when you know how to do something in a programming language like C/C++, Java, Swift, Pascal, or Visual Basic, but you can't yet figure out the solution in assembly language.

Most programmers like being able to apply past experience when learning something new. Unfortunately, traditional approaches to learning assembly language programming tend to force HLL programmers to forget what they've learned in the past. This book, in contrast, offers a way for you to efficiently leverage your existing knowledge while learning assembly language.

2.2 How This Book Can Help

Once you've read through this book, there are three reasons you'll find it much easier to learn assembly language:

- You'll be more motivated to learn it because you'll understand why doing so can help you write better code.
- You'll have had five brief primers on assembly language (80x86, PowerPC, ARM, Java bytecode, and Microsoft IL), so even if you'd never seen it before, you'll have learned some by the time you finish this book.
- You'll have already seen how compilers emit machine code for all the common control and data structures, so you'll have learned one of the most difficult lessons for a beginning assembly programmer—how to achieve things in assembly language that they already know how to do in an HLL.

Though this book won't teach you how to become an expert assembly language programmer, the large number of example programs that demonstrate how compilers translate HLLs into machine code will acquaint you with many assembly language programming techniques. You'll find these useful should you decide to learn assembly language after reading this book.

Certainly, you'll find this book easier to read if you already know assembly language. However, you'll also find assembly language easier to master once you've read this book. Since learning assembly language is probably more time-consuming than reading this book, the more efficient approach is to start with the book.

2.3 High-Level Assemblers to the Rescue

Way back in 1995, I had a discussion with the University of California, Riverside, computer science department chair. I was lamenting the fact that students had to start over when taking the assembly course, spending precious time to relearn so many things. As the discussion progressed, it became clear that the problem wasn't with assembly language, per se, but with the syntax of existing assemblers (like Microsoft Macro Assembler, or MASM). Learning assembly language entailed a whole lot more than learning a few machine instructions. First of all, you have to learn a new programming style. Mastering assembly language involves learning not only the semantics of a few machine instructions but also how to put those instructions together to solve real-world problems. And *that's* the hard part.

Second, *pure* assembly language is not something you can efficiently pick up a few instructions at a time. Writing even the simplest programs requires considerable knowledge and a repertoire of a couple dozen or more machine instructions. When you add that repertoire to all the other machine organization topics students must learn in a typical assembly course, it's often several weeks before they are prepared to write anything other than "spoon-fed" trivial applications in assembly language.

One important feature of MASM back in 1995 was support for HLL-like control statements such as `.if` and `.while`. While these statements are not true machine instructions, they do allow students to use familiar programming constructs early in the course, until they've had time to learn enough low-level machine instructions that they can use them in their applications. By using these high-level constructs early on in the term, students can concentrate on other aspects of assembly language programming and not have to assimilate everything all at once. This allows them to start writing code much sooner in the course and, as a result, they wind up covering more material by the end of the term.

An assembler like MASM (32-bit v6.0 and later) that provides control statements similar to those found in HLLs—in addition to the traditional low-level machine instructions that do the same thing—is called a *high-level assembler*. In theory, with an appropriate textbook that teaches assembly language programming using these high-level assemblers, students could begin writing simple programs during the very first week of the course.

The only problem with high-level assemblers like MASM is that they provide just a few HLL control statements and data types. Almost everything else is foreign to someone who is familiar with HLL programming. For example, data declarations in MASM are completely different from data declarations in most HLLs. Beginning assembly programmers still have to relearn a considerable amount of information, despite the presence of HLL-like control statements.

2.4 High-Level Assembly Language

Shortly after the discussion with my department chair, it occurred to me that there is no reason an assembler couldn't adopt a more high-level syntax without changing the semantics of assembly language. For example, consider the following statements in C/C++ and Pascal that declare an integer array variable:

```
int intVar[8]; // C/C++

var intVar: array[0..7] of integer; (* Pascal *)
```

Now consider the MASM declaration for the same object:

```
intVar sdword 8 dup (?) ;MASM
```

While the C/C++ and Pascal declarations differ from each other, the assembly language version is radically different from both. A C/C++ programmer will probably be able to figure out the Pascal declaration even if they have never seen Pascal code before, and vice versa. However, Pascal and C/C++ programmers probably won't be able to make heads or tails of the assembly language declaration. This is but one example of the problems HLL programmers face when first learning assembly language.

The sad part is that there's really no reason a variable declaration in assembly language has to be so radically different from one in an HLL. It makes absolutely no difference in the final executable file which syntax an assembler uses for variable declarations. Given that, why shouldn't an assembler use a more high-level-like syntax so people switching over from HLLs will find the assembler easier to learn? Pondering this question led me to develop a new assembly language, specifically geared toward teaching assembly language programming to students who had already mastered an HLL, called *High-Level Assembly (HLA)*. In HLA, the aforementioned array declaration looks like this:

```
var intVar:int32[8]; // HLA
```

Though the syntax is slightly different from C/C++ and Pascal (actually, it's a combination of the two), most HLL programmers can probably figure out the meaning of this declaration.

The whole purpose of HLA's design is to provide an assembly language programming environment as similar as possible to that of traditional (imperative) high-level programming languages, without sacrificing the capability to write *real* assembly language programs. Those components of the language that have nothing to do with machine instructions use a familiar high-level language syntax, while the machine instructions still map one-to-one to the underlying 80x86 machine instructions.

Making HLA as similar as possible to various HLLs means that students learning assembly language programming don't have to spend as much time assimilating a radically different syntax. Instead, they can apply their existing HLL knowledge, which makes the process of learning assembly language easier and faster.

A comfortable syntax for declarations and a few HLL-like control statements aren't all you need to make learning assembly language as efficient as possible, however. One very common complaint about learning assembly language is that it provides very little support for programmers, who must constantly reinvent the wheel while writing assembly code. For example, when learning assembly language programming using MASM, you'll quickly discover that assembly language doesn't provide useful I/O facilities such as the ability to print integer values as strings to the user's console. Assembly programmers are responsible for writing such code themselves. Unfortunately, writing a decent set of I/O routines requires sophisticated knowledge of assembly language programming. The only way to gain that knowledge is by writing a fair amount of code first, but doing so without having any I/O routines is difficult. Therefore, a good assembly language educational tool also needs to provide a set of I/O routines that allow beginning assembly programmers to do simple I/O tasks, like reading and writing integer values, before they have the programming sophistication to write such routines themselves. HLA accomplishes this with the *HLA Standard Library*, a collection of subroutines and macros that make it very easy to write complex applications.

Because of HLA's popularity and the fact that HLA is a free, open source, and public domain product available for Windows and Linux, this book uses HLA syntax for compiler-neutral examples involving assembly language. Despite the fact that it is now over 20 years old and supports only the 32-bit Intel instruction set, HLA is still an excellent way to learn assembly language programming. Although the latest Intel CPUs directly support 64-bit registers and operations, learning 32-bit assembly language is just as relevant for HLL programmers as 64-bit assembly.

2.5 Thinking High-Level, Writing Low-Level

The goal of HLA is to allow a beginning assembly programmer to think in HLL terms while writing low-level code (in other words, the exact opposite of what this book is trying to teach). For students first approaching assembly language, being able to think in high-level terms is a godsend—they can apply techniques they've already learned in other languages when faced with a particular assembly language programming problem. Controlling the rate at which a student has to learn new concepts in this way can make the educational process more efficient.

Ultimately, of course, the goal is to learn the low-level programming paradigm. This means gradually giving up HLL-like control structures and writing pure low-level code (that is, "thinking low-level and writing low-level"). Nevertheless, starting out by "thinking high-level while writing low-level" is a great, incremental way to learn assembly language programming.

2.6 The Assembly Programming Paradigm (Thinking Low-Level)

It should be clear now that programming in assembly language is quite different from programming in common HLLs. Fortunately, for this book, you don't need to be able to write assembly language programs from scratch. Nevertheless, if you know how assembly programs are written, you'll be able to understand why a compiler emits certain code sequences. To that end, I'll take some time here to describe how assembly language programmers (and compilers) "think."

The most fundamental aspect of the assembly language programming paradigm—that is, the model for how assembly programming is accomplished—is that large projects are broken up into mini-tasks that the machine can handle. Fundamentally, a CPU can do only one tiny task at a time; this is true even for complex instruction set computers (CISC). Therefore, complex operations, like statements you'll find in an HLL, have to be broken down into smaller components that the machine can execute directly. As an example, consider the following Visual Basic (VB) assignment statement:

```
profits = sales - costOfGoods - overhead - commissions
```

No practical CPU will allow you to execute this entire VB statement as a single machine instruction. Instead, you have to break this assignment statement down to a sequence of machine instructions that compute individual components of it. For example, many CPUs provide a *subtract* instruction that lets you subtract one value from a machine register. Because the assignment statement in this example consists of three subtractions, you'll have to break the assignment operation down into at least three different subtract instructions.

The 80x86 CPU family provides a fairly flexible subtract instruction: `sub()`. This particular instruction allows the following forms (in HLA syntax):

```
sub( constant, reg );       // reg = reg - constant
sub( constant, memory );    // memory = memory - constant
sub( reg1, reg2 );          // reg2 = reg2 - reg1
sub( memory, reg );         // reg = reg - memory
sub( reg, memory );         // memory = memory - reg
```

Assuming that all of the identifiers in the original VB code represent variables, we can use the 80x86 `sub()` and `mov()` instructions to implement the same operation with the following HLA code sequence:

```
// Get sales value into EAX register:

mov( sales, eax );

// Compute sales-costOfGoods (EAX := EAX - costOfGoods)

sub( costOfGoods, eax );

// Compute (sales-costOfGoods) - overhead
// (note: EAX contains sales-costOfGoods)

sub( overhead, eax );

// Compute (sales-costOfGoods-overhead)-commissions
// (note: EAX contains sales-costOfGoods-overhead)

sub( commissions, eax );

// Store result (in EAX) into profits:

mov( eax, profits );
```

This code breaks down the single VB statement into five different HLA statements, each of which does a small part of the total calculation. The secret behind the assembly language programming paradigm is knowing how to break down complex operations like this into a simple sequence of machine instructions. We'll take another look at this process in Chapter 13.

HLL control structures are another big area where complex operations are broken down into simpler statement sequences. For example, consider the following Pascal `if()` statement:

```
if( i = j ) then begin

    writeln( "i is equal to j" );

end;
```

CPUs do not support an `if` machine instruction. Instead, you compare two values that set *condition-code flags* and then test the result of these condition codes by using *conditional jump* instructions. A common way to translate an HLL `if` statement into assembly language is to test the opposite condition (`i <> j`) and then jump over the statements that would be executed if the original condition (`i = j`) evaluates to `true`. For example, here's a translation of the former Pascal `if` statement into HLA (using pure assembly language—that is, no HLL-like constructs):

```
    mov( i, eax );      // Get i's value into eax register
    cmp( eax, j );      // Compare eax to j's value
    jne skipIfBody;     // Skip body of if statement if i <> j

    << code to print string >>

skipIfBody:
```

As the Boolean expressions in the HLL control structures increase in complexity, the number of corresponding machine instructions also increases. But the process remains the same. Later, we'll take a look at how compilers translate HLL control structures into assembly language (see Chapters 13 and 14).

Passing parameters to a procedure or function, accessing those parameters, and then accessing other data local to that procedure or function is another area where assembly language is quite a bit more complex than typical HLLs. This is an important subject, but it's beyond the scope of this chapter, so we'll revisit it in Chapter 15.

The bottom line is that when converting an algorithm from a high-level language, you have to break down the problem into much smaller pieces in order to code it in assembly language. As noted earlier, the good news is that you don't have to figure out which machine instructions to use when all you're doing is reading assembly code—the compiler (or assembly programmer) that originally created the code will have already done this for you. All you have to do is draw a correspondence between the HLL code and the assembly code. How you accomplish that is the subject of much of the rest of this book.

2.7 For More Information

Bartlett, Jonathan. *Programming from the Ground Up*. Edited by Dominick Bruno, Jr. Self-published, 2004. An older, free version of this book, which teaches assembly language programming using Gas, can be found online at *http://www.plantation-productions.com/AssemblyLanguage/ProgrammingGroundUp-1-0-booksize.pdf*.

Blum, Richard. *Professional Assembly Language*. Indianapolis: Wiley, 2005.

Carter, Paul. *PC Assembly Language*. Self-published, 2019. *https://pacman128.github.io/static/pcasm-book.pdf*.

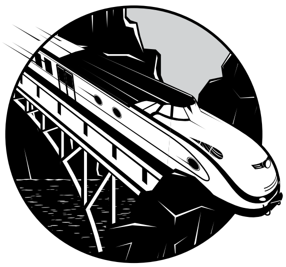
Duntemann, Jeff. *Assembly Language Step-by-Step*. 3rd ed. Indianapolis: Wiley, 2009.

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

———. "Webster: The Place on the Internet to Learn Assembly." *http://plantation-productions.com/Webster/index.html*.

3

80X86 ASSEMBLY FOR THE HLL PROGRAMMER



Throughout this book, you'll examine high-level language code and compare it to the machine code that a compiler generates for it. Making sense of a compiler's output requires some knowledge of assembly language, but fortunately, you don't need to be an expert assembly programmer for this. As discussed in previous chapters, all you really need is the ability to read code generated by compilers and other assembly language programmers.

This chapter provides a primer specifically on the 80x86 assembly language, covering the following topics:

- The basic 80x86 machine architecture
- How to read the 80x86 output produced by various compilers
- The addressing modes that the 32-bit and 64-bit 80x86 CPUs support
- The syntax that several common 80x86 assemblers (HLA, MASM, and Gas) use
- How to use constants and declare data in assembly language programs

3.1 Learning One Assembly Language Is Good, Learning More Is Better

If you intend to write code for a processor other than the 80x86, you should really learn how to read at least two different assembly languages. By doing so, you'll avoid the pitfall of coding for the 80x86 in an HLL and then finding that your "optimizations" work only on the 80x86 CPU. For this reason, this book includes several online appendixes that provide additional resources:

- Appendix A covers the minimal x86 instruction set.
- Appendix B is a primer on the PowerPC CPU.
- Appendix C examines the ARM processor.
- Appendix D describes the Java bytecode assembly language.
- Appendix E covers the Microsoft Intermediate Language.

You'll see that all five architectures rely on many of the same concepts, but there are some important differences among them, and advantages and disadvantages to each.

Perhaps the main difference between *complex instruction set computer (CISC)* and *reduced instruction set computer (RISC)* architectures is the way they use memory. RISC architectures limit memory access to certain instructions, so applications go to great lengths to avoid accessing memory. The 80x86 architecture, on the other hand, allows most instructions to access memory, and applications generally take advantage of this facility.

The Java bytecode (JBC) and Microsoft Intermediate Language (IL) architectures differ from the 80x86, PowerPC, and ARM families in that JBC and IL are *virtual machines*, not actual CPUs. Generally, software interprets or attempts to compile JBC at runtime (IL code is always compiled at runtime).[1] This means JBC and IL code tends to run much slower than true machine code.

3.2 80x86 Assembly Syntaxes

While 80x86 programmers can choose from a wide variety of program development tools, this abundance has a minor drawback: syntactical incompatibility. Different compilers and debuggers for the 80x86 family output different assembly language listings for the exact same program. This is because those tools emit code for different assemblers. For example, Microsoft's Visual C++ package generates assembly code compatible with Microsoft Macro Assembler (MASM). The GNU Compiler Suite (GCC) generates Gas-compatible source code (Gas is the GNU Assembler

1. The JBC interpreter can provide *just-in-time (JIT)* compilation that translates the interpreted bytecode to machine code at runtime. Microsoft's intermediate code is always JIT-compiled. However, the quality of a JIT compiler is rarely as good as the machine code produced by a native code compiler.

from the Free Software Foundation). In addition to the code that compilers emit, you'll find tons of assembly programming examples written with assemblers like FASM, NASM, GoAsm, and HLA (High-Level Assembly).

It would be nice to use just a single assembler syntax throughout this book, but because our approach is not compiler specific, we must consider the syntaxes for several different common assemblers. This book will generally present non-compiler-specific examples using HLA. Therefore, this chapter will discuss the syntaxes for HLA as well as two other common assemblers, MASM and Gas. Fortunately, once you master the syntax for one assembler, learning the syntax of other assemblers is very easy.

3.2.1 Basic 80x86 Architecture

The Intel CPU is generally classified as a *Von Neumann machine*. Von Neumann computer systems contain three main building blocks: the *central processing unit (CPU)*, *memory*, and *input/output (I/O) devices*. These three components are connected via the *system bus* (consisting of the address, data, and control buses). Figure 3-1 shows this relationship.

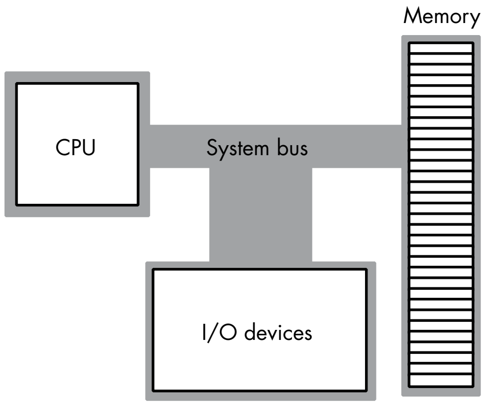


Figure 3-1: Block diagram of a Von Neumann system

The CPU communicates with memory and I/O devices by placing a numeric value on the *address bus* to select one of the memory locations or I/O device port locations, each of which has a unique binary numeric address. Then the CPU, I/O, and memory devices pass data among themselves by placing the data on the *data bus*. The *control bus* contains signals that determine the direction of the data transfer (to or from memory, and to or from an I/O device).

3.2.2 Registers

The register set is the most prominent feature within the CPU. Almost all calculations on the 80x86 CPU involve at least one register. For example, to add the value of two variables and store their sum in a third variable, you must load one of the variables into a register, add the second operand to the register, and then store the register's value in the destination variable.

Registers are middlemen in almost every calculation and thus are very important in 80x86 assembly language programs.

The 80x86 CPU registers can be broken down into four categories: general-purpose registers, special-purpose application-accessible registers, segment registers, and special-purpose kernel-mode registers. We won't consider the last two categories, because the segment registers are not used very much in modern operating systems (for example, Windows, BSD, macOS, and Linux), and the special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools—a topic well beyond the scope of this book.

3.2.3 80x86 32-Bit General-Purpose Registers

The 32-bit 80x86 (Intel family) CPUs provide several general-purpose registers for application use. These include eight 32-bit registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP.

The *E* prefix on each name stands for *extended*. This prefix differentiates the 32-bit registers from the original eight 16-bit registers: AX, BX, CX, DX, SI, DI, BP, and SP.

Finally, the 80x86 CPUs provide eight 8-bit registers: AL, AH, BL, BH, CL, CH, DL, and DH.

The most important thing to note about the general-purpose registers is that they are not independent. That is, the 80x86 architecture does not provide 24 separate registers. Instead, it overlaps the 32-bit registers with the 16-bit registers, and it overlaps the 16-bit registers with the 8-bit registers. Figure 3-2 shows this relationship.

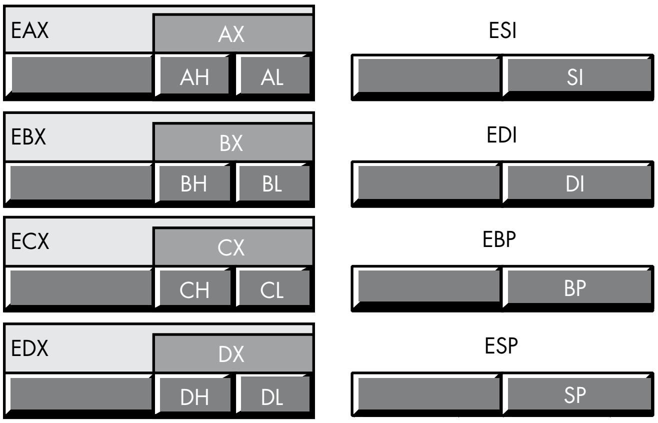


Figure 3-2: Intel 80x86 CPU general-purpose registers

The fact that modifying one register may modify as many as three other registers cannot be overemphasized. For example, modifying the EAX register may also modify the AL, AH, and AX registers. You will often see compiler-generated code using this feature of the 80x86. For example, a compiler may clear (set to `0`) all the bits in the EAX register and then load

AL with a `1` or `0` in order to produce a 32-bit `true` (`1`) or `false` (`0`) value. Some machine instructions manipulate only the AL register, yet the program may need to return those instructions' results in EAX. By taking advantage of the register overlap, the compiler-generated code can use an instruction that manipulates AL to return that value in all of EAX.

Although Intel calls these registers *general purpose*, that's not to suggest that you can use any register for any purpose. The SP/ESP register pair, for example, has a very special purpose that effectively prevents you from using it for any other reason (it's the *stack pointer*). Likewise, the BP/EBP register has a special purpose that limits its usefulness as a general-purpose register. All the 80x86 registers have their own special purposes that limit their use in certain contexts; we'll consider these special uses as we discuss the machine instructions that use them (see the online resources).

Contemporary versions of the 80x86 CPU (typically known as the *x86-64 CPU*) provide two important extensions to the 32-bit register set: a set of 64-bit registers and a second set of eight registers (64-bit, 32-bit, 16-bit, and 8-bit). The main 64-bit registers have the following names: RAX, RBX, RCX, RDX, RSI, RDI, RBP, and RSP.

These 64-bit registers overlap the 32-bit "E" registers. That is, the 32-bit registers comprise the LO (low-order) 32 bits of each of these registers. For example, EAX is the LO 32 bits of RAX. Similarly, AX is the LO 16 bits of RAX, and AL is the LO 8 bits of RAX.

In addition to providing 64-bit variants of the existing 80x86 32-bit registers, the x86-64 CPUs also add eight other 64/32/16/8-bit registers: R15, R14, R13, R12, R11, R10, R9, and R8.

You can refer to the LO 32 bits of each of these registers as R15d, R14d, R13d, R12d, R11d, R10d, R9d, and R8d.

You can refer to the LO 16 bits of each of these registers as R15w, R14w, R13w, R12w, R11w, R10w, R9w, and R8w.

Finally, you can refer to the LO byte of each of these registers as R15b, R14b, R13b, R12b, R11b, R10b, R9b, and R8b.

3.2.4 The 80x86 EFLAGS Register

The 32-bit EFLAGS register encapsulates numerous single-bit Boolean (`true`/`false`) values (or *flags*). Most of these bits are either reserved for kernel-mode (operating system) functions or of little interest to application programmers. There are, however, 8 bits relevant to application programmers reading (or writing) assembly language code: the overflow, direction, interrupt disable,[2] sign, zero, auxiliary carry, parity, and carry flags. Figure 3-3 shows their layout within the EFLAGS register.

Of the eight flags that application programmers can use, four flags in particular are extremely valuable: the overflow, carry, sign, and zero flags. We call these four flags the *condition codes*. Each flag has a state—set or cleared—that you can use to test the result of previous computations. For

2. Application programs cannot modify the interrupt flag, but I've mentioned it here because it's discussed later in this text.

example, after comparing two values, the condition-code flags will tell you if one value is less than, equal to, or greater than the other.

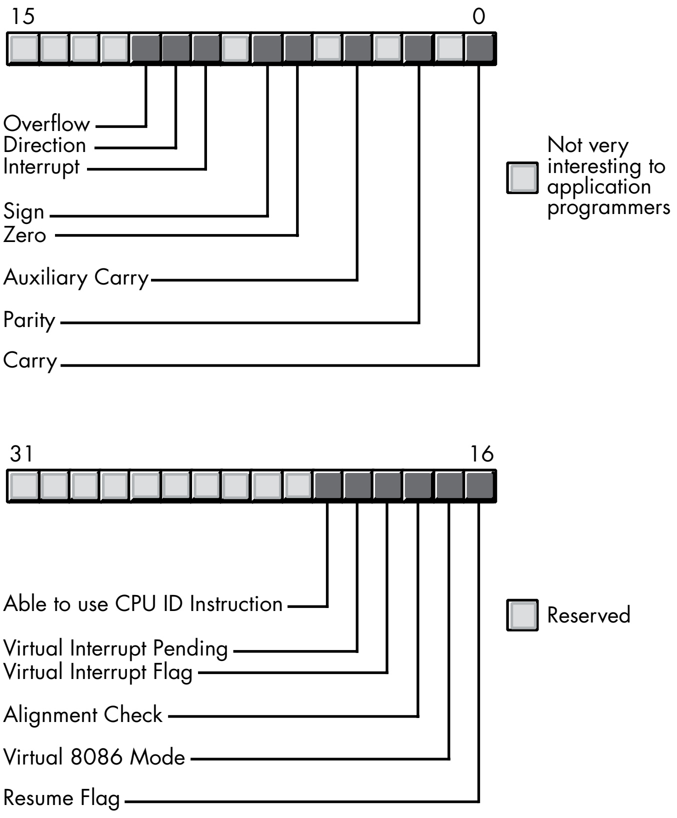


Figure 3-3: Layout of the 80x86 flags register (LO 16 bits)

The x86-64 64-bit RFLAGS register reserves all bits from bit 32 through bit 63. The upper 16 bits of the EFLAGS register are generally useful only to operating systems code.

Because the RFLAGS register doesn't contain anything of interest when reading compiler output, this book will simply refer to the x86 and x86-64 flags register as EFLAGs, even on 64-bit variants of the CPU.

3.3 Literal Constants

Most assemblers support literal numeric (binary, decimal, and hexadecimal), character, and string constants. Unfortunately, just about every assembler out there uses a different syntax for literal constants. This section describes the syntax for the assemblers we'll be using in this book.

3.3.1 Binary Literal Constants

All assemblers provide the ability to specify base-2 (binary) literal constants. Few compilers emit binary constants, so you probably won't see these values in the output a compiler produces, but you may see them in handwritten assembly code. C++ 14 supports binary literals (`0bxxxxx`) as well.

3.3.1.1 Binary Literal Constants in HLA

Binary literal constants in HLA begin with the percent character (`%`) followed by one or more binary digits (`0` or `1`). Underscore characters may appear between any two digits in a binary number. By convention, HLA programmers separate each group of four digits with an underscore. For example:

```
%1011
%1010_1111
%0011_1111_0001_1001
%1011001010010101
```

3.3.1.2 Binary Literal Constants in Gas

Binary literal constants in Gas begin with the special `0b` prefix followed by one or more binary digits (`0` or `1`). For example:

```
0b1011
0b10101111
0b0011111100011001
0b1011001010010101
```

3.3.1.3 Binary Literal Constants in MASM

Binary literal constants in MASM consist of one or more binary digits (`0` or `1`) followed by the special `b` suffix. For example:

```
1011b
10101111b
0011111100011001b
1011001010010101b
```

3.3.2 Decimal Literal Constants

Decimal constants in most assemblers take the standard form—a sequence of one or more decimal digits without any special prefix or suffix. This is one of the two common numeric formats that compilers emit, so you'll often see decimal literal constants in compiler output code.

3.3.2.1 Decimal Literal Constants in HLA

HLA allows you to optionally insert underscores between any two digits in a decimal number. HLA programmers generally use underscores to separate groups of three digits in a decimal number. For example, for the following numbers:

```
123
1209345
```

an HLA programmer could insert underscores as follows:

```
1_024
1_021_567
```

3.3.2.2 Decimal Literal Constants in Gas and MASM

Gas and MASM use a string of decimal digits (the standard "computer" format for decimal values). For example:

```
123
1209345
```

Unlike HLA, Gas and MASM do not allow embedded underscores in decimal literal constants.

3.3.3 Hexadecimal Literal Constants

Hexadecimal (base-16) literal constants are the other common numeric format you'll find in assembly language programs (especially those that compilers emit).

3.3.3.1 Hexadecimal Literal Constants in HLA

Hexadecimal literal constants in HLA consist of a string of hexadecimal digits (`0..9`, `a..f`, or `A..F`) with a `$` prefix. Underscores may optionally appear between any two hexadecimal digits in the number. By convention, HLA programmers separate sequences of four digits with underscores. For example:

```
$1AB0
$1234_ABCD
$dead
```

3.3.3.2 Hexadecimal Literal Constants in Gas

Hexadecimal literal constants in Gas consist of a string of hexadecimal digits (`0..9`, `a..f`, or `A..F`) with a `0x` prefix. For example:

```
0x1AB0
0x1234ABCD
0xdead
```

3.3.3.3 Hexadecimal Literal Constants in MASM

Hexadecimal literal constants in MASM consist of a string of hexadecimal digits (`0..9`, `a..f`, or `A..F`) with an `h` suffix. The values must begin with a decimal digit (`0` if the constant would normally begin with a digit in the range `a..f`). For example:

```
1AB0h
1234ABCDh
0deadh
```

3.3.4 Character and String Literal Constants

Character and string data are also common data types that you'll find in assembly programs. MASM does not differentiate between character or string literal constants. HLA and Gas, however, use a different internal representation for characters and strings, so the distinction between the two kinds of literal constants is very important in those assemblers.

3.3.4.1 Character and String Literal Constants in HLA

Character literal constants in HLA take a few different forms. The most common is a single printable character surrounded by a pair of apostrophes, such as `'A'`. To specify an actual apostrophe as a character literal constant, HLA requires that you surround one pair of apostrophes by another (`''''`). Finally, you can also indicate a character constant using the `#` symbol followed by a binary, decimal, or hexadecimal numeric value that specifies the ASCII code of the character you want to use. For example:

```
'a'
''''
' '
#$d
#10
#%0000_1000
```

String literal constants in HLA consist of a sequence of zero or more characters surrounded by quotation marks. To indicate an actual quotation mark character within a string constant, you use two adjacent quotation marks. For example:

```
"Hello World"
"" -- The empty string
"He said ""Hello"" to them"
"""" -- string containing one quote character
```

3.3.4.2 Character and String Literal Constants in Gas

Character literal constants in Gas consist of an apostrophe followed by a single character. More modern versions of Gas (and Gas on the Mac) also allow character constants of the form `'a'`. For example:

```
'a
''
'!
'a'   // Modern versions of Gas and Mac's assembler
'!'   // Modern versions of Gas and Mac's assembler
```

String literal constants in Gas consist of a sequence of zero or more characters surrounded by quotes, and use the same syntax as C strings. You use the `\` escape sequence to embed special characters in a Gas string. For example:

```
"Hello World"
"" -- The empty string
"He said \"Hello\" to them"
"\"" -- string containing one quote character
```

3.3.4.3 Character and String Literal Constants in MASM

Character and string literal constants in MASM take the same form: a sequence of one or more characters surrounded by either apostrophes or quotes. MASM does not differentiate character constants and string constants. For example:

```
'a'
"'" - An apostrophe character
'"' - A quote character
"Hello World"
"" -- The empty string
'He said "Hello" to them'
```

3.3.5 Floating-Point Literal Constants

Floating-point literal constants in assembly language typically take the same form you'll find in HLLs (a sequence of digits, possibly containing a decimal point, optionally followed by a signed exponent). For example:

```
3.14159
2.71e+2
1.0e-5
5e2
```

3.4 Manifest (Symbolic) Constants in Assembly Language

Almost every assembler provides a mechanism for declaring symbolic (named) constants. In fact, most assemblers provide several ways to associate a value with an identifier in the source file.

3.4.1 Manifest Constants in HLA

The HLA assembler, true to its name, uses a high-level syntax for declaring named constants in the source file. You may define constants in one of three ways: in a `const` section, in a `val` section, or with the `?` compile-time operator. The `const` and `val` sections appear in the declaration section of an HLA program, and their syntax is very similar. The difference between them is that you may reassign values to identifiers you define in the `val` section, but you may not reassign values to identifiers appearing in a `const` section. Although HLA supports a wide range of options in these declaration sections, the basic declaration takes the following form:

```
const
    someIdentifier := someValue;
```

Wherever *`someIdentifier`* appears in the source file (after this declaration), HLA will substitute the value *`someValue`* in the identifier's place. For example:

```
const
    aCharConst := 'a';
    anIntConst := 12345;
    aStrConst := "String Const";
    aFltConst := 3.12365e-2;

val
    anotherCharConst := 'A';
    aSignedConst := -1;
```

In HLA, the `?` statement allows you to embed `val` declarations anywhere whitespace is allowed in the source file. This is sometimes useful because it isn't always convenient to declare constants in a declaration section. For example:

```
?aValConst := 0;
```

3.4.2 Manifest Constants in Gas

Gas uses the `.equ` ("equate") statement to define a symbolic constant in the source file. This statement has the following syntax:

```
.equ        symbolName, value
```

Here are some examples of equates within a Gas source file:

```
.equ       false, 0
.equ       true, 1
.equ       anIntConst, 12345
```

3.4.3 Manifest Constants in MASM

MASM also provides a couple of different ways to define manifest constants within a source file. One way is with the `equ` directive:

```
false      equ    0
true       equ    1
anIntConst equ    12345
```

Another is with the `=` operator:

```
false      =    0
true       =    1
anIntConst =    12345
```

The difference between the two is minor; see the MASM documentation for details.

NOTE *For the most part, compilers tend to emit the* `equ` *form rather than the* `=` *form.*

3.5 80x86 Addressing Modes

An *addressing mode* is a hardware-specific mechanism for accessing instruction operands. The 80x86 family provides three different classes of operands: register, immediate, and memory operands. This section discusses each of these addressing modes.

3.5.1 80x86 Register Addressing Modes

Most 80x86 instructions can operate on the 80x86's general-purpose register set. You access a register by specifying its name as an instruction operand.

Let's consider some examples of how our assemblers implement this strategy, using the 80x86 `mov` (move) instruction.

3.5.1.1 Register Access in HLA

The HLA `mov` instruction looks like this:

```
mov( source, destination );
```

This instruction copies the data from the *`source`* operand to the *`destination`* operand. The 8-bit, 16-bit, and 32-bit registers are valid operands for this instruction; the only restriction is that both operands must be the same size.

Now let's look at some actual 80x86 mov instructions:

```
mov( bx, ax );      // Copies the value from BX into AX
mov( al, dl );      // Copies the value from AL into DL
mov( edx, esi );    // Copies the value from EDX into ESI
```

Note that HLA supports only the 32-bit 80x86 register set, not the 64-bit register set.

3.5.1.2 Register Access in Gas

Gas prepends each register name with percent sign (%). For example:

```
%al, %ah, %bl, %bh, %cl, %ch, %dl, %dh
%ax, %bx, %cx, %dx, %si, %di, %bp, %sp
%eax, %ebx, %ecx, %edx, %esi, %edi, %ebp, %esp
%rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp
%r15b, %r14b, %r13b, %r12b, %r11b, %r10b, %r9b, %r8b
%r15w, %r14w, %r13w, %r12w, %r11w, %r10w, %r9w, %r8w
%r15d, %r14d, %r13d, %r12d, %r11d, %r10d, %r9d, %r8d
%r15, %r14, %r13, %r12, %r11, %r10, %r9, %r8
```

The Gas syntax for the mov instruction is similar to HLA's, except that it drops the parentheses and semicolons and requires the assembly language statements to fit completely on one physical line of source code. For example:

```
mov %bx, %ax       // Copies the value from BX into AX
mov %al, %dl       // Copies the value from AL into DL
mov %edx, %esi     // Copies the value from EDX into ESI
```

3.5.1.3 Register Access in MASM

The MASM assembler uses the same register names as HLA but adds support for the 64-bit register set:

```
al, ah, bl, bh, cl, ch, dl, dh
ax, bx, cx, dx, si, di, bp, sp
eax, ebx, ecx, edx, esi, edi, ebp, esp
rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp
r15b, r14b, r13b, r12b, r11b, r10b, r9b, r8b
r15w, r14w, r13w, r12w, r11w, r10w, r9w, r8w
r15d, r14d, r13d, r12d, r11d, r10d, r9d, r8d
r15, r14, r13, r12, r11, r10, r9, r8
```

MASM uses a basic syntax that's similar to that of Gas, except that MASM reverses the operands (which is the standard Intel syntax). That is, a typical instruction like mov takes this form:

```
mov destination, source
```

Here are some examples of the mov instruction in MASM syntax:

```
mov ax, bx       ; Copies the value from BX into AX
mov dl, al       ; Copies the value from AL into DL
mov esi, edx     ; Copies the value from EDX into ESI
```

3.5.2 Immediate Addressing Mode

Most instructions that allow register and memory operands also allow immediate, or *constant*, operands. The following HLA mov instructions, for example, load appropriate values into the corresponding destination registers:

```
mov( 0, al );
mov( 12345, bx );
mov( 123_456_789, ecx );
```

Most assemblers allow you to specify a wide variety of literal constant types when using the immediate addressing mode. For example, you can supply numbers in hexadecimal, decimal, or binary form. You can also supply character constants as operands. The rule is that the constant must fit in the size specified for the destination operand.

Here are some additional examples with HLA, Gas, and MASM (note that Gas requires a $ before immediate operands):

```
mov( 'a', ch );  // HLA
mov $'a', %ch    // Gas
mov ch, 'a'       ; MASM

mov( $1234, ax ); // HLA
mov $0x1234, %ax  // Gas
mov ax, 1234h      ; MASM

mov( 4_012_345_678, eax ); // HLA
mov $4012345678, %eax      // Gas
mov eax, 4012345678         ; MASM
```

Almost every assembler lets you create symbolic constant names and supply them as source operands. For example, HLA predefines the two Boolean constants true and false, so you can supply those names as mov instruction operands:

```
mov( true, al );
mov( false, ah );
```

Some assemblers even allow pointer constants and other abstract data type constants. (See the reference manual for your assembler for details.)

3.5.3 Displacement-Only Memory Addressing Mode

The most common 32-bit addressing mode, and the one that's the easiest to understand, is the *displacement-only* (or *direct*) addressing mode, in which a

32-bit constant specifies the address of the memory location, which may be either the source or the destination operand. Note that this addressing mode is available only on 32-bit x86 processors or when operating in 32-bit mode on a 64-bit processor.

For example, assuming that variable `J` is a byte variable appearing at address `$8088`, the HLA instruction `mov(J,al);` loads the AL register with a copy of the byte at memory location `$8088`. Likewise, if the byte variable `K` is at address `$1234` in memory, then the instruction `mov(dl,K);` writes the value in the DL register to memory location `$1234` (see Figure 3-4).

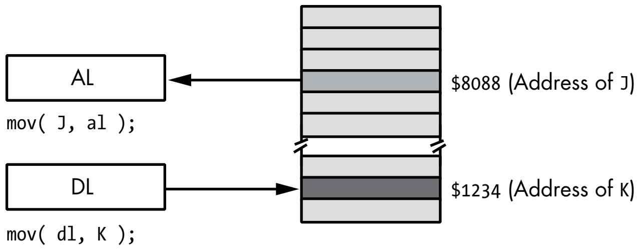


Figure 3-4: Displacement-only (direct) addressing mode

The displacement-only addressing mode is perfect for accessing simple scalar variables. It is the addressing mode you'd normally use to access static or global variables in an HLL program.

NOTE *Intel named this addressing mode "displacement-only" because a 32-bit constant (displacement) follows the* `mov` *opcode in memory. On the 80x86 processors, this displacement is an offset from the beginning of memory (that is, address* `0`*).*

The examples in this chapter will often access byte-sized objects in memory. Don't forget, however, that you can also access words and double words on the 80x86 processors by specifying the address of their first byte (see Figure 3-5).

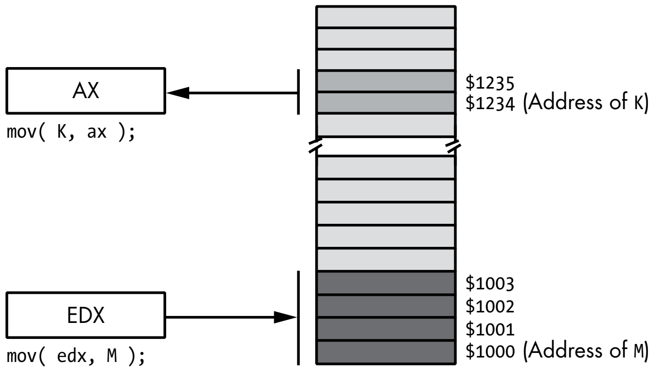


Figure 3-5: Accessing a word or double word using the direct addressing mode

MASM and Gas use the same syntax for the displacement addressing mode as HLA: for the operand, you simply specify the name of the object you want to access. Some MASM programmers put brackets around the variable names, although that isn't strictly necessary with those assemblers.

Here are several examples using HLA, Gas, and MASM syntax:

```
mov( byteVar, ch );  // HLA
movb byteVar, %ch    // Gas
mov ch, byteVar       ; MASM

mov( wordVar, ax ); // HLA
movw wordVar, %ax   // Gas
mov ax, wordVar      ; MASM

mov( dwordVar, eax );   // HLA
movl dwordVar, %eax     // Gas
mov eax, dwordVar        ; MASM
```

3.5.4 RIP-Relative Addressing Mode

The x86-64 CPUs, when operating in 64-bit mode, do not support the 32-bit direct addressing mode. Not wanting to add a 64-bit constant to the end of the instruction (to support the entire 64-bit address space), AMD engineers chose to create an RIP-relative addressing mode that computes the effective memory address by adding a signed 32-bit constant (replacing the direct address) to the value in the RIP (instruction pointer) register. This allows for accessing data within a ±2GB range around the current instruction.[3]

3.5.5 Register Indirect Addressing Mode

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. These modes are called *indirect* because the operand is not the actual address; rather, its value specifies the memory address to use. In the case of the register indirect addressing modes, the register's value is the address to access. For example, the HLA instruction `mov(eax,[ebx]);` tells the CPU to store EAX's value at the location whose address is held in EBX.

The x86-64 CPUs also support a register indirect addressing mode in 64-bit mode using one of the 64-bit registers (for example, RAX, RBX, . . . , R15). The register indirect addressing mode allows full access to the 64-bit address space. For example, the MASM instruction `mov eax, [rbx]` tells the CPU to load the EAX register from the location whose address is in RBX.

3. Technically, the x86-64 does allow you to load/store the AL, AX, EAX, or RAX register using a 64-bit displacement. This form exists mainly to access memory-mapped I/O devices and isn't an instruction typical applications would use.

3.5.5.1 Register Indirect Modes in HLA

There are eight forms of this addressing mode on the 80x86. Using HLA syntax, they look like this:

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

These eight addressing modes reference the memory location at the offset found in the register enclosed by brackets (EAX, EBX, ECX, EDX, EDI, ESI, EBP, or ESP, respectively).

NOTE *The HLA register indirect addressing modes require a 32-bit register. You cannot specify a 16-bit or 8-bit register when using an indirect addressing mode.*

3.5.5.2 Register Indirect Modes in MASM

MASM uses exactly the same syntax as HLA for the register indirect addressing modes in 32-bit mode (though keep in mind that MASM reverses the instruction operands; only the addressing mode syntax is identical). In 64-bit mode the syntax is the same—a pair of brackets around a register name—although this mode uses 64-bit registers rather than 32-bit registers.

Here are the MASM equivalents of the instructions given earlier:

```
mov al, [eax]
mov al, [ebx]
mov al, [ecx]
mov al, [edx]
mov al, [edi]
mov al, [esi]
mov al, [ebp]
mov al, [esp]
```

Here are the MASM 64-bit register indirect addressing mode examples:

```
mov al,   [rax]
mov ax,   [rbx]
mov eax,  [rcx]
mov rax,  [rdx]
mov r15b, [rdi]
mov r15w, [rsi]
mov r15d, [rbp]
mov r15,  [rsp]
mov al,   [r8]
mov ax,   [r9]
mov eax,  [r10]
mov rax,  [r11]
mov r15b, [r12]
mov r15w, [r13]
mov r15d, [r14]
mov r15,  [r15]
```

3.5.5.3 Register Indirect Modes in Gas

Gas uses parentheses instead of brackets around the register names. Here are the Gas variants of the previous 32-bit HLA `mov` instructions:

```
movb (%eax), %al
movb (%ebx), %al
movb (%ecx), %al
movb (%edx), %al
movb (%edi), %al
movb (%esi), %al
movb (%ebp), %al
movb (%esp), %al
```

Here are Gas's 64-bit register indirect variants:

```
movb (%rax), %al
movb (%rbx), %al
movb (%rcx), %al
movb (%rdx), %al
movb (%rdi), %al
movb (%rsi), %al
movb (%rbp), %al
movb (%rsp), %al
movb (%r8),  %al
movb (%r9),  %al
movb (%r10), %al
movb (%r11), %al
movb (%r12), %al
movb (%r13), %al
movb (%r14), %al
movb (%r15), %al
```

3.5.6 Indexed Addressing Mode

The *effective address* is the ultimate address in memory that an instruction will access once all the address calculations are complete. The indexed addressing mode computes an effective address by adding the address (also called the *displacement* or *offset*) of the variable to the value held in the 32-bit or 64-bit register within the square brackets. Their sum provides the memory address that the instruction accesses. For example, if `VarName` is at address `$1100` in memory and EBX contains `8`, then `mov(VarName[ebx],al);` loads the byte at address `$1108` into the AL register (see Figure 3-6).

On the x86-64 CPUs, the addressing mode uses one of the 64-bit registers. Note, however, that the displacement encoded as part of the instruction is still 32 bits. Thus, the register must hold the base address while the displacement provides an offset (index) from the base address.

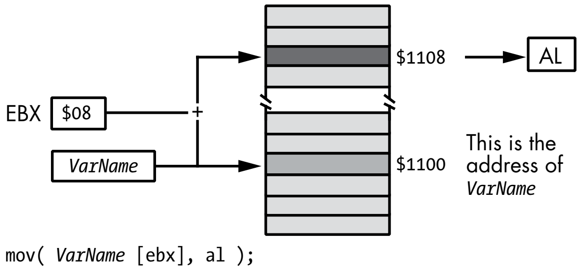


Figure 3-6: Indexed addressing mode

3.5.6.1 Indexed Addressing Mode in HLA

The indexed addressing modes use the following HLA syntax, where `VarName` is the name of some static variable in your program:

```
mov( VarName[ eax ], al );
mov( VarName[ ebx ], al );
mov( VarName[ ecx ], al );
mov( VarName[ edx ], al );
mov( VarName[ edi ], al );
mov( VarName[ esi ], al );
mov( VarName[ ebp ], al );
mov( VarName[ esp ], al );
```

3.5.6.2 Indexed Addressing Mode in MASM

MASM supports the same syntax as HLA in 32-bit mode, but it also allows several variations of this syntax for specifying the indexed addressing mode. The following are equivalent formats that demonstrate some of the variations MASM supports:

```
varName[reg32]
[reg32 + varName]
[varName][reg32]
[varName + reg32]
[reg32][varName]
varName[reg32 + const]
[reg32 + varName + const]
[varName][reg32][const]
varName[const + reg32]
[const + reg32 + varName]
[const][reg32][varName]
varName[reg32 - const]
[reg32 + varName - const]
[varName][reg32][-const]
```

Thanks to the commutative nature of addition, MASM also allows many other combinations. It treats two juxtaposed items within brackets as though they were separated by the + operator.

Here are the MASM equivalents to the previous HLA example:

```
mov  al, VarName[ eax ]
mov  al, VarName[ ebx ]
mov  al, VarName[ ecx ]
mov  al, VarName[ edx ]
mov  al, VarName[ edi ]
mov  al, VarName[ esi ]
mov  al, VarName[ ebp ]
mov  al, VarName[ esp ]
```

In 64-bit mode, MASM requires that you specify 64-bit register names for the indexed addressing mode. In 64-bit mode, the register holds the base address of the variable in memory, and the displacement encoded into the instruction provides an offset from that base address. This means that you cannot use a register as an index into a global array (which would normally use the RIP-relative addressing mode).

Here are examples of the valid MASM indexed address modes in 64-bit mode:

```
mov  al, [ rax + SomeConstant ]
mov  al, [ rbx + SomeConstant ]
mov  al, [ rcx + SomeConstant ]
mov  al, [ rdx + SomeConstant ]
mov  al, [ rdi + SomeConstant ]
mov  al, [ rsi + SomeConstant ]
mov  al, [ rbp + SomeConstant ]
mov  al, [ rsp + SomeConstant ]
```

3.5.6.3 Indexed Addressing Mode in Gas

As with the register indirect addressing mode, Gas uses parentheses rather than brackets. Here is the Gas syntax for the indexed addressing mode:

```
varName(%reg32)
const(%reg32)
varName + const(%reg32)
```

Here are the Gas equivalents to the HLA instructions given earlier:

```
movb VarName( %eax ), al
movb VarName( %ebx ), al
movb VarName( %ecx ), al
movb VarName( %edx ), al
movb VarName( %edi ), al
movb VarName( %esi ), al
movb VarName( %ebp ), al
movb VarName( %esp ), al
```

In 64-bit mode, Gas requires that you specify 64-bit register names for the indexed addressing mode. The same rules apply as for MASM.

Here are examples of the valid Gas indexed address modes in 64-bit mode:

```
mov  %al, SomeConstant(%rax)
mov  %al, SomeConstant(%rbx)
mov  %al, SomeConstant(%rcx)
mov  %al, SomeConstant(%rdx)
mov  %al, SomeConstant(%rsi)
mov  %al, SomeConstant(%rdi)
mov  %al, SomeConstant(%rbp)
mov  %al, SomeConstant(%rsp)
```

3.5.7 Scaled-Index Addressing Modes

The scaled-index addressing modes are similar to the indexed addressing modes, but with two differences. The scaled-index addressing modes allow you to:

- Combine two registers plus a displacement
- Multiply the index register by a (scaling) factor of 1, 2, 4, or 8

To see what makes this possible, consider the following HLA example:

```
mov( eax, VarName[ ebx + esi*4 ] );
```

The primary difference between the scaled-index addressing mode and the indexed addressing mode is the inclusion of the `esi*4` component. This example computes the effective address by adding in the value of ESI multiplied by 4 (see Figure 3-7).

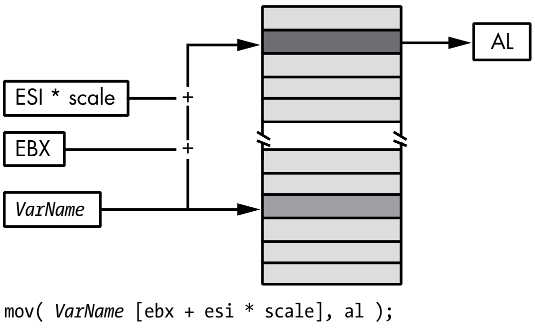


Figure 3-7: Scaled-index addressing mode

In 64-bit mode, substitute 64-bit registers for the base and index registers.

3.5.7.1 Scaled-Index Addressing in HLA

HLA's syntax provides several different ways to specify the scaled-index addressing mode. Here are the various syntactical forms:

```
VarName[ IndexReg32*scale ]
VarName[ IndexReg32*scale + displacement ]
VarName[ IndexReg32*scale - displacement ]

[ BaseReg32 + IndexReg32*scale ]
[ BaseReg32 + IndexReg32*scale + displacement ]
[ BaseReg32 + IndexReg32*scale - displacement ]

VarName[ BaseReg32 + IndexReg32*scale ]
VarName[ BaseReg32 + IndexReg32*scale + displacement ]
VarName[ BaseReg32 + IndexReg32*scale - displacement ]
```

In these examples, *BaseReg*$_{32}$ represents any general-purpose 32-bit register, *IndexReg*$_{32}$ represents any general-purpose 32-bit register except ESP, and *scale* must be one of the constants `1`, `2`, `4`, or `8`. *VarName* represents a static variable name and *displacement* represents a 32-bit constant.

3.5.7.2 Scaled-Index Addressing in MASM

MASM supports the same syntax for these addressing modes as HLA, but with additional forms comparable to those presented for the indexed addressing mode. Those forms are just syntactical variants based on the commutativity of the `+` operator.

MASM also supports 64-bit scaled-index addressing, which has the same syntax as the 32-bit mode except you swap in 64-bit register names. The major difference between the 32-bit and 64-bit scaled-index addressing modes is that there is no 64-bit `disp[reg*index]` addressing mode. On 64-bit addressing modes, this is a PC-relative indexed addressing mode, where the displacement is a 32-bit offset from the current instruction pointer value.

3.5.7.3 Scaled-Index Addressing in Gas

As usual, Gas uses parentheses rather than brackets to surround scaled-index operands. Gas also uses a three-operand syntax to specify the *base register*, the *index register*, and the *scale factor*, rather than the arithmetic expression syntax that the other assemblers employ. The generic syntax for the Gas scaled-index addressing mode is:

```
expression( baseReg32, indexReg32, scaleFactor )
```

More specifically:

```
VarName( ,IndexReg32, scale )
VarName + displacement( ,IndexReg32, scale )
VarName - displacement( ,IndexReg32, scale )
```

```
( BaseReg$_{32}$, IndexReg$_{32}$, scale )
displacement( BaseReg$_{32}$, IndexReg$_{32}$, scale)

VarName( BaseReg$_{32}$, IndexReg$_{32}$, scale )
VarName + displacement( BaseReg$_{32}$, IndexReg$_{32}$, scale )
VarName - displacement( BaseReg$_{32}$, IndexReg$_{32}$, scale )
```

where *scale* is one of the values 1, 2, 4, or 8.

Gas also supports 64-bit scaled-index addressing. It uses the same syntax as the 32-bit mode except you swap in 64-bit register names. When using 64-bit addressing, you cannot also specify an RIP-relative variable name (*VarName* in these examples); only a 32-bit *displacement* is legal.

3.6 Declaring Data in Assembly Language

The 80x86 architecture provides only a few low-level machine data types on which individual machine instructions operate:

byte	Holds arbitrary 8-bit values.
word	Holds arbitrary 16-bit values.
dword	"Double word"; holds arbitrary 32-bit values.
qword	"Quad word"; holds arbitrary 64-bit values.
real32 (aka real4)	Holds 32-bit single-precision floating-point values.
real64 (aka real8)	Holds 64-bit double-precision floating-point values.

NOTE *80x86 assemblers typically support* tbyte *("ten byte") and* real80/real10 *data types, but we won't cover those types here because most modern (64-bit) HLL compilers don't use them. (However, certain C/C++ compilers support* real80 *values using the* long double *data type; Swift also supports* real80 *values on Intel machines using the* float80 *type.)*

3.6.1 Data Declarations in HLA

The HLA assembler, true to its high-level nature, provides a wide variety of single-byte data types including character, signed integer, unsigned integer, Boolean, and enumerated types. Were you to write an application in assembly language, having all these different data types (along with the type checking that HLA provides) would be quite useful. For our purposes, however, we can simply allocate storage for byte variables and set aside a block of bytes for larger data structures. The HLA byte type is all we really need for 8-bit and array objects.

You can declare byte objects in an HLA static section as follows:

```
static
    variableName : byte;
```

To allocate storage for a block of bytes, you'd use the following HLA syntax:

```
static
    blockOfBytes : byte[ sizeOfBlock ];
```

These HLA declarations create *uninitialized* variables. Technically speaking, HLA always initializes `static` objects with `0`s, so they aren't truly uninitialized, but the main point is that this code does not explicitly initialize these byte objects with a value. You can, however, tell HLA to initialize your byte variables with a value when the operating system loads the program into memory using statements like the following:

```
static
    // InitializedByte has the initial value 5:

    InitializedByte : byte := 5;

    // InitializedArray is initialized with 0, 1, 2, and 3;

    InitializedArray : byte[4] := [0,1,2,3];
```

3.6.2 Data Declarations in MASM

In MASM, you would normally use the `db` or `byte` directives within a `.data` section to reserve storage for a byte object or an array of byte objects. The syntax for a single declaration would take one of these equivalent forms:

```
variableName    db      ?
variableName    byte    ?
```

The preceding declarations create uninitialized objects (which are actually initialized with `0`s, just as with HLA). The `?` in the operand field of the `db`/`byte` directive informs the assembler that you don't want to explicitly attach a value to the declaration.

To declare a variable that is a block of bytes, you'd use syntax like the following:

```
variableName    db      sizeOfBlock dup (?)
variableName    byte    sizeOfBlock dup (?)
```

To create objects with an initial value other than zero, you could use syntax like the following:

```
                        .data
InitializedByte         db      5
InitializedByte2        byte    6
InitializedArray0       db      4 dup (5)   ; array is 5,5,5,5
InitializedArray1       db      5 dup (6)   ; array is 6,6,6,6,6
```

To create an initialized array of bytes whose values are not all the same, you simply specify a comma-delimited list of values in the operand field of the MASM db/byte directive:

```
                    .data
InitializedArray2   byte    0,1,2,3
InitializedArray3   byte    4,5,6,7,8
```

3.6.3 Data Declarations in Gas

Gas uses the .byte directive in a .data section to declare a byte variable. The generic form of this directive is:

```
variableName: .byte 0
```

Gas doesn't provide an explicit format for creating uninitialized variables; instead, you just supply a 0 operand for uninitialized variables. Here are two actual byte variable declarations in Gas:

```
InitializedByte: .byte   5
ZeroedByte       .byte   0  // Zeroed value
```

Gas does not provide an explicit directive for declaring an array of byte objects, but you can use the .rept/.endr directives to create multiple copies of the .byte directive as follows:

```
variableName:
        .rept   sizeOfBlock
        .byte   0
        .endr
```

NOTE *You can also supply a comma-delimited list of values if you want to initialize the array with different values.*

Here are a couple of array declaration examples in Gas:

```
            .section    .data
InitializedArray0:      // Creates an array with elements 5,5,5,5
            .rept       4
            .byte       5
            .endr

InitializedArray1:
            .byte       0,1,2,3,4,5
```

3.6.3.1 Accessing Byte Variables in Assembly Language

When accessing byte variables, you simply use the variable's declared name in one of the 80x86 addressing modes. For example, given a byte object named byteVar and an array of bytes named byteArray, you could use any of the following instructions to load that variable into the AL register using the mov instruction (these examples assume 32-bit code):

```
// HLA's mov instruction uses "src, dest" syntax:

mov( byteVar, al );
mov( byteArray[ebx], al ); // EBX is the index into byteArray

// Gas's movb instruction also uses a "src, dest" syntax:

movb byteVar, %al
movb byteArray(%ebx), %al

; MASM's mov instructions use "dest, src" syntax

mov al, byteVar
mov al, byteArray[ebx]
```

For 16-bit objects, HLA uses the word data type, MASM uses either the dw or word directives, and Gas uses the .int directive. Other than the size of the object these directives declare, their use is identical to the byte declarations. For example:

```
// HLA example:

static

    // HLAwordVar: 2 bytes, initialized with 0s:

    HLAwordVar : word;

    // HLAwordArray: 8 bytes, initialized with 0s:

    HLAwordArray : word[4];

    // HLAwordArray2: 10 bytes, initialized with 0, ..., 5:

    HLAwordArray2 : word[5] := [0,1,2,3,4];

; MASM example:

                    .data
MASMwordVar         word    ?
MASMwordArray       word    4 dup (?)
MASMwordArray2      word    0,1,2,3,4

// Gas example:

                    .section    .data
GasWordVar:         .int    0
```

```
GasWordArray:
                    .rept   4
                    .int    0
                    .endr

GasWordArray2:      .int    0,1,2,3,4
```

For 32-bit objects, HLA uses the `dword` data type, MASM uses the `dd` or `dword` directives, and Gas uses the `.long` directive. For example:

```
// HLA example:

static
    // HLAdwordVar: 4 bytes, initialized with 0s:

    HLAdwordVar : dword;

    // HLAdwordArray: 16 bytes, initialized with 0s.

    HLAdwordArray : dword[4];

    // HLAdwordArray: 20 bytes, initialized with 0, ..., 4:

    HLAdwordArray2 : dword[5] := [0,1,2,3,4];

; MASM/TASM example:

                    .data
MASMdwordVar        dword   ?
MASMdwordArray      dword   4 dup (?)
MASMdwordArray2     dword   0,1,2,3,4

// Gas example:

                    .section    .data
GasDWordVar:        .long   0
GasDWordArray:
                    .rept   4
                    .long   0
                    .endr

GasDWordArray2:     .long   0,1,2,3,4
```

3.7 Specifying Operand Sizes in Assembly Language

80x86 assemblers use two mechanisms to specify their operand sizes:

- The operands specify the size using type checking (most assemblers do this).
- The instructions themselves specify the size (Gas does this).

For example, consider the following three HLA `mov` instructions:

```
mov( 0, al );
mov( 0, ax );
mov( 0, eax );
```

In each case, the register operand specifies the size of the data that the `mov` instruction copies into that register. MASM uses a similar syntax (though the operands are reversed):

```
mov al,  0 ; 8-bit data movement
mov ax,  0 ; 16-bit data movement
mov eax, 0 ; 32-bit data movement
```

The takeaway here is that the instruction mnemonic (`mov`) is exactly the same in all six cases. The operand, not the instruction mnemonic, specifies the size of the data transfer.

NOTE *Modern versions of Gas also allow you to specify the size of the operation by operand (register) size without using a suffix such as* `b` *or* `w`*. This book, however, will continue to use mnemonics such as* `movb` *or* `movw` *to avoid confusion with older variants of Gas. "Type Coercion in Gas" on page 45.*

3.7.1 Type Coercion in HLA

There is one problem with the preceding approach to specifying the operand size. Consider the following HLA example:

```
mov( 0, [ebx] );  // Copy 0 to the memory location
                  // pointed at by EBX.
```

This instruction is ambiguous. The memory location to which EBX points could be a byte, a word, or a double word. Nothing in the instruction tells the assembler the size of the operand. Faced with an instruction like this, the assembler will report an error, and you'll have to explicitly tell it the size of the memory operand. In HLA's case, this is done with a type coercion operator as follows:

```
mov( 0, (type word [ebx]) );  // 16-bit data movement.
```

In general, you can coerce any memory operand to an appropriate size using the following HLA syntax:

```
(type new_type memory)
```

where *`new_type`* represents a data type (such as `byte`, `word`, or `dword`) and *`memory`* represents the memory address whose type you would like to override.

3.7.2 Type Coercion in MASM

MASM suffers from this same problem. You will need to coerce the memory location using a coercion operator like the following:

```
mov  word ptr [ebx], 0   ; 16-bit data movement.
```

Of course, you can substitute `byte` or `dword` for `word` in these two examples to coerce the memory location to a byte or double word size.

3.7.3 Type Coercion in Gas

Gas doesn't require type coercion operators, because it uses a different technique to specify the size of its operands. Rather than using the single mnemonic `mov`, Gas uses four mnemonics consisting of `mov` plus a single-character suffix that indicates the size:

`movb`	Copy an 8-bit (`byte`) value
`movw`	Copy a 16-bit (`word`) value
`movl`	Copy a 32-bit (`long`) value
`movq`	Copy a 64-bit (`long long`) value

There is never any ambiguity when you use these mnemonics, even if their operands don't have an explicit size. For example:

```
movb $0, (%ebx) // 8-bit data copy
movw $0, (%ebx) // 16-bit data copy
movl $0, (%ebx) // 32-bit data copy
movq $0, (%rbx) // 64-bit data copy
```

With this basic information, you should now be able to understand the output from a typical compiler.

3.8 For More Information

Bartlett, Jonathan. *Programming from the Ground Up*. Edited by Dominick Bruno, Jr. Self-published, 2004. An older, free version of this book, which teaches assembly language programming using Gas, can be found online at *http://www.plantation-productions.com/AssemblyLanguage/ProgrammingGroundUp-1-0-booksize.pdf*.

Blum, Richard. *Professional Assembly Language*. Indianapolis: Wiley, 2005.

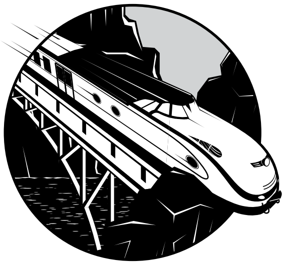
Duntemann, Jeff. *Assembly Language Step-by-Step*. 3rd ed. Indianapolis: Wiley, 2009.

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

Intel. "Intel 64 and IA-32 Architectures Software Developer Manuals." Updated November 11, 2019. *https://software.intel.com/en-us/articles/intel-sdm/*.

4

COMPILER OPERATION AND CODE GENERATION



In order to write HLL code that produces efficient machine code, you must first understand how compilers and linkers translate high-level source statements into executable machine code. Complete coverage of compiler theory is beyond the scope of this book; however, this chapter explains the basics of the translation process so you can understand and work within the limitations of HLL compilers. We'll cover the following topics:

- The different types of input files programming languages use
- Differences between compilers and interpreters
- How typical compilers process source files to produce executable programs
- The process of optimization and why compilers cannot produce the best possible code for a given source file
- Different types of output files that compilers produce
- Common object file formats, such as COFF and ELF

- Memory organization and alignment issues that affect the size and efficiency of executable files a compiler produces
- How linker options can affect the efficiency of your code

This material provides the foundation for all the chapters that follow, and is crucial to helping a compiler produce the best possible code. We'll begin with a discussion of file formats used by programming languages.

4.1 File Types That Programming Languages Use

A typical program can take many forms. A *source file* is a human-readable form that a programmer creates and supplies to a language translator (such as a compiler). A typical compiler translates the source file or files into an *object code* file. A *linker program* combines separate object modules to produce a relocatable or executable file. Finally, a *loader* (usually the operating system) loads the executable file into memory and makes the final modifications to the object code prior to execution. Note that the modifications are made to the object code that is now in memory; the actual file on the disk does not get modified. These are not the only types of files that language processing systems manipulate, but they are typical. To fully understand compiler limitations, it's important to know how the language processor deals with each of these file types. We'll look at source files first.

4.2 Source Files

Traditionally, source files contain pure ASCII or Unicode text (or some other character set) that a programmer has created with a text editor. One advantage to using pure text files is that a programmer can manipulate a source file using any program that processes text files. For example, a program that counts the number of lines in an arbitrary text file will also count the number of source lines in a program. Because there are hundreds of little filter programs that manipulate text files, maintaining source files in a pure text format is a good approach. This format is sometimes called *plain vanilla text*.

4.2.1 Tokenized Source Files

Some language processing systems (especially interpreters) maintain their source files in a *tokenized* form. Tokenized source files generally use special single-byte *token* values to compress reserved words and other lexical elements in the source language, and thus they are often smaller than text source files. Furthermore, interpreters that operate on tokenized code are generally an order of magnitude faster than interpreters that operate on pure text, because processing strings of single-byte tokens is far more efficient than recognizing reserved word strings.

Generally, the tokenized file from the interpreter consists of a sequence of bytes that map directly to strings such as `if` and `print` in the source file.

So, by using a table of strings and a little extra logic, you can decipher a tokenized program to produce the original source code. (Usually, you lose any extra whitespace you inserted into the source file, but that's about the only difference.) Many of the original BASIC interpreters found on early PC systems worked this way. You'd type a line of BASIC source code into the interpreter, and the interpreter would immediately tokenize that line and store the tokenized form in memory. Later, when you executed the `LIST` command, the interpreter would *detokenize* the source code in memory to produce the listing.

On the flip side, tokenized source files often use a proprietary format. This means they can't take advantage of general-purpose text-manipulation tools like `wc` (word count), `entab`, and `detab` (which count the number of lines, words, and characters in a text file; replace spaces with tabs; and replace tabs with spaces, respectively).

To overcome this limitation, most languages that operate on tokenized files enable you to detokenize a source file to produce a standard text file. (They also allow you to retokenize a source file, given an input text file.) You then run the resulting text file through some filter program, and retokenize the output of the filter program to produce a new tokenized source file. Although this takes considerable work, it allows language translators that work with tokenized files to take advantage of various text-based utility programs.

4.2.2 Specialized Source Files

Some programming languages, such as Embarcadero's Delphi and Free Pascal's comparable Lazarus program, do not use a traditional text-based file format at all. Instead, they often use graphical elements like flowcharts and forms to represent the instructions the program is to perform. Other examples are the Scratch programming language, which allows you to write simple programs using graphical elements on a bitmapped display, and the Microsoft Visual Studio and Apple Xcode integrated development environments (IDEs), which both allow you to specify a screen layout using graphical operations rather than a text-based source file.

4.3 Types of Computer Language Processors

Computer language processing systems generally fall into one of four categories: pure interpreters, interpreters, compilers, and incremental compilers. These systems differ in how they process the source program and execute the result, which affects their respective efficiency.

4.3.1 Pure Interpreters

Pure interpreters operate directly on a text source file and tend to be very inefficient. They continuously scan the source file (usually an ASCII text file), processing it as string data. Recognizing *lexemes* (language components such as reserved words, literal constants, and the like) consumes time. Indeed, many pure interpreters spend more time processing the lexemes (that is, performing *lexical analysis*) than they do actually executing the program. Because the actual on-the-fly execution of the lexeme takes only a little additional effort beyond the lexical analysis, pure interpreters tend to be the smallest of the computer language processing programs. For this reason, pure interpreters are popular when you need a very compact language processor. They are also popular for scripting languages and very high-level languages that let you manipulate the language's source code as string data during program execution.

4.3.2 Interpreters

An *interpreter* executes some representation of a program's source file at runtime. This representation isn't necessarily a text file in human-readable form. As noted in the previous section, many interpreters operate on tokenized source files in order to avoid lexical analysis during execution. Some interpreters read a text source file as input and translate the input file to a tokenized form prior to execution. This allows programmers to work with text files in their favorite editor while enjoying the fast execution of a tokenized format. The only costs are an initial delay to tokenize the source file (which is unnoticeable on most modern machines) and the fact that it may not be possible to execute strings containing program statements.

4.3.3 Compilers

A *compiler* translates a source program in text form into executable machine code. This is a complex process, particularly in optimizing compilers. There are a couple of things to note about the code a compiler produces. First, a compiler produces machine instructions that the underlying CPU can execute directly. Therefore, the CPU doesn't waste any cycles decoding the source file while executing the program—all of the CPU's resources are dedicated to executing the machine code. Thus, the resulting program generally runs many times faster than an interpreted version does. Of course, some compilers do a better job of translating HLL source code into machine code than other compilers, but even low-quality compilers do a better job than most interpreters.

A compiler's translation from source code to machine code is a one-way function. In contrast to interpreters, it is very difficult, if not impossible, to reconstruct the original source file if you're given only the machine code output from a program.

4.3.4 Incremental Compilers

An *incremental compiler* is a cross between a compiler and an interpreter. There are many different types of incremental compilers, but in general, they operate like an interpreter in that they do not compile the source file directly into machine code; instead, they translate the source code into an intermediate form. Unlike the tokenized code from interpreters, however, this intermediate form usually is not strongly correlated to the original source file. The intermediate form is generally the machine code for a *virtual machine language*—"virtual" in that there is no real CPU that can execute this code. However, it is easy to write an interpreter that can execute it. Because interpreters for virtual machines (VMs) are usually much more efficient than interpreters for tokenized code, executing VM code is usually much faster than executing a list of tokens in an interpreter. Languages like Java use this compilation technique, along with a *Java bytecode engine* (an interpreter program), to interpretively execute the Java "machine code" (see Figure 4-1). The big advantage to VM execution is that the VM code is portable; that is, programs running on the virtual machine can execute anywhere an interpreter is available. True machine code, by contrast, executes only on the CPU (family) for which it was written. Generally, interpreted VM code runs about 2 to 10 times faster than interpreted code (tokenized), and pure machine code runs about 2 to 10 times faster than interpreted VM code.

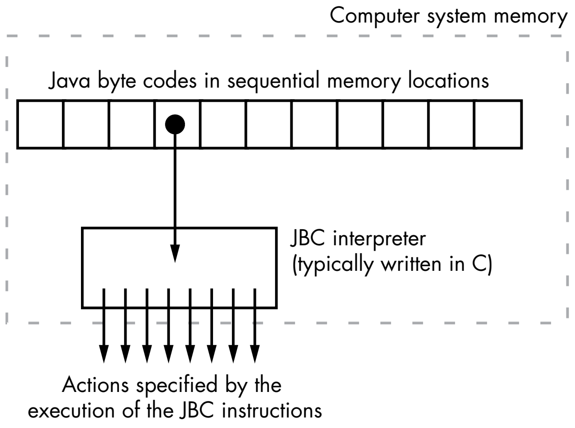


Figure 4-1: The JBC interpreter

In an attempt to improve the performance of programs compiled via an incremental compiler, many vendors (particularly Java systems vendors) have turned to a technique known as *just-in-time (JIT) compilation*. The concept is based on the fact that the time spent in interpretation is largely consumed by fetching and deciphering the VM code at runtime. This interpretation occurs repeatedly as the program executes. JIT compilation translates the VM code to actual machine code whenever it encounters a VM instruction for the first time. This spares the interpreter from repeating the interpretation process the next time it encounters the same statement in the program (for example, in a loop). Although JIT compilation is nowhere near as good as a true compiler, it can typically improve the performance of a program by a factor of two to five.

NOTE *Older compilers and some freely available compilers compile the source code to assembly language, and then a separate compiler, known as an* assembler*, assembles this output to the desired machine code. Most modern and highly efficient compilers skip this step altogether. See "Compiler Output" on page 67 for more on this subject.*

Of the four categories of computer language processors just described, this chapter will focus on compilers. By understanding how a compiler generates machine code, you can choose appropriate HLL statements to generate better, more efficient machine code. If you want to improve the performance of programs written with an interpreter or incremental compiler instead, the best approach is to use an optimizing compiler to process your application. For example, GNU provides a compiler for Java that produces optimized machine code rather than interpreted Java bytecode (JBC); the resulting executable files run much faster than interpreted JBC or even JIT-compiled bytecode.

4.4 The Translation Process

A typical compiler is broken down into several logical components called *phases*. Although their exact number and names may vary somewhat among different compilers, the five most common phases are *lexical analysis*, *syntax analysis*, *intermediate code generation*, *native code generation*, and, for compilers that support it, *optimization*.

Figure 4-2 shows how the compiler logically arranges these phases to translate source code in the HLL into machine (object) code.

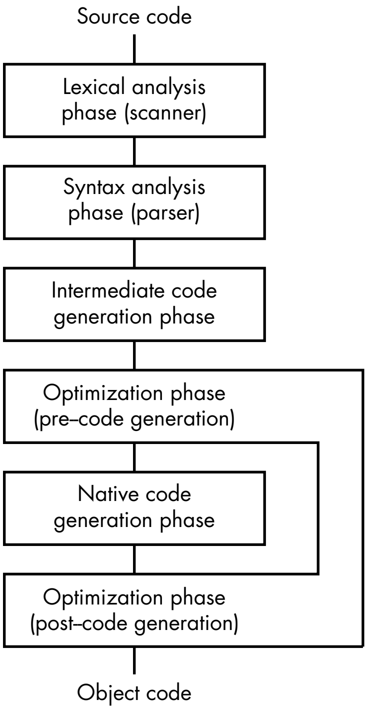

Source code
Lexical analysis phase (scanner)
Syntax analysis phase (parser)
Intermediate code generation phase
Optimization phase (pre–code generation)
Native code generation phase
Optimization phase (post–code generation)
Object code

Figure 4-2: Phases of compilation

Although Figure 4-2 suggests that the compiler executes these phases sequentially, most compilers do not. Instead, the phases tend to execute in parallel, with each phase doing a small amount of work, passing its output to the next phase, and then waiting for input from the previous phase. In a typical compiler, the *parser* (the syntax analysis phase) is probably the closest thing you'll find to the main program or the master process. The parser usually drives the compilation process in that it calls the *scanner* (lexical analysis phase) to obtain input and calls the intermediate code generator to process its own output. The intermediate code generator may (optionally) call the optimizer and then call the native code generator. The native code generator may (optionally) call the optimizer as well. The output from the native code generation phase is the executable code. After the native code generator/optimizer emits some code, execution returns to the intermediate code generator, then to the parser, which requests more input from the scanner, starting the whole process over.

NOTE *Other compiler organizations are possible. Some compilers, for example, allow the user to choose whether the compiler runs the optimization phase, while others don't have an optimization phase at all. Similarly, some compilers dispense with intermediate code generation and directly call a native code generator. Some compilers include additional phases that process object modules compiled at different times.*

Thus, although Figure 4-2 doesn't accurately depict the typical (parallel) execution path, the *data flow* it shows is correct. The scanner reads the source file, translates it to a different form, and then passes this translated data on to the parser. The parser accepts its input from the scanner, translates that input to a different form, and then passes this new data to the intermediate code generator. Similarly, the remaining phases read their input from the previous phase, translate the input to a (possibly) different form, and then pass that input on to the next phase. The compiler writes the output of its last phase to the executable object file.

Let's take a closer look at each phase of the code translation process.

4.4.1 Scanning (Lexical Analysis)

The scanner (aka the *lexical analyzer*, or *lexer*) is responsible for reading the character/string data found in the source file and breaking up this data into tokens that represent the lexical items, or lexemes, in the source file. As mentioned previously, lexemes are the character sequences in the source file that we would recognize as atomic components of the language. For example, a scanner for the C language would recognize substrings like `if` and `while` as C reserved words. The scanner would not, however, pick out the "if " within the identifier `ifReady` and treat it as a reserved word. Instead, the scanner considers the context in which a reserved word is used so that it can differentiate between reserved words and identifiers. For each lexeme, the scanner creates a small data package—a token—and passes it on to the parser. A token typically contains several values:

- A small integer that uniquely identifies the token's class (whether it's a reserved word, identifier, integer constant, operator, or character string literal)

- Another value that differentiates the token within a class (for example, this value would indicate which reserved word the scanner has processed)
- Any other attributes the scanner might associate with the lexeme

NOTE *Do not confuse this reference to a token with the compressed-style tokens in an interpreter discussed previously. In this context, tokens are simply a variable-sized data structure that holds information associated with a lexeme for the interpreter/compiler.*

When the scanner sees the character string `12345` in the source file, for example, it might identify the token's class as a literal constant, the token's second value as an integer typed constant, and the token's attribute as the numeric equivalent of the string (that is, twelve thousand, three hundred, forty-five). Figure 4-3 demonstrates what this token might look like in memory.

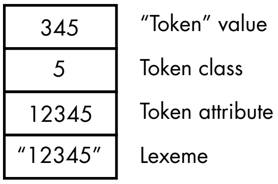


Figure 4-3: A token for the lexeme "12345"

The token's enumerated value is `345` (indicating an integer constant), the token class's value is `5` (indicating a literal constant), the token's attribute value is `12345` (the numeric form of the lexeme), and the lexeme string is `"12345"` as returned by the scanner. Different code sequences in the compiler can refer to this token data structure as appropriate.

Strictly speaking, the lexical analysis phase is optional. A parser could work directly with the source file. However, tokenization makes the compilation process more efficient, because it allows the parser to deal with tokens as integer values rather than as string data. Because most CPUs can handle small integer values much more efficiently than string data, and because the parser has to refer to the token data multiple times during processing, lexical analysis saves considerable time during compilation. Generally, pure interpreters are the only language processors that rescan each token during parsing, and this is one major reason why they are so slow (compared to, say, an interpreter that stores the source file in a tokenized form to avoid constantly processing a pure-text source file).

4.4.2 Parsing (Syntax Analysis)

The parser is the part of the compiler that is responsible for checking whether the source program is syntactically (and semantically) correct. If there's an error in the source file, it's usually the parser that discovers and reports it. The parser is also responsible for reorganizing the token stream (that is, the source code) into a more complex data structure that represents the meaning or semantics of the program. The scanner and parser generally process the source file in a linear fashion from the beginning to the end of the file, and the compiler usually reads the source file only once. Later phases, however, need to refer to the body of the source program in a more ad hoc way. By building up a data structure representation of the source code (often called an *abstract syntax tree*, or *AST*), the parser enables the code generation and optimization phases to easily reference different parts of the program.

Figure 4-4 shows how a compiler might represent the expression `12345+6` using three nodes in an AST (`43` is the value for the addition operator and `7` is the subclass representing arithmetic operators).

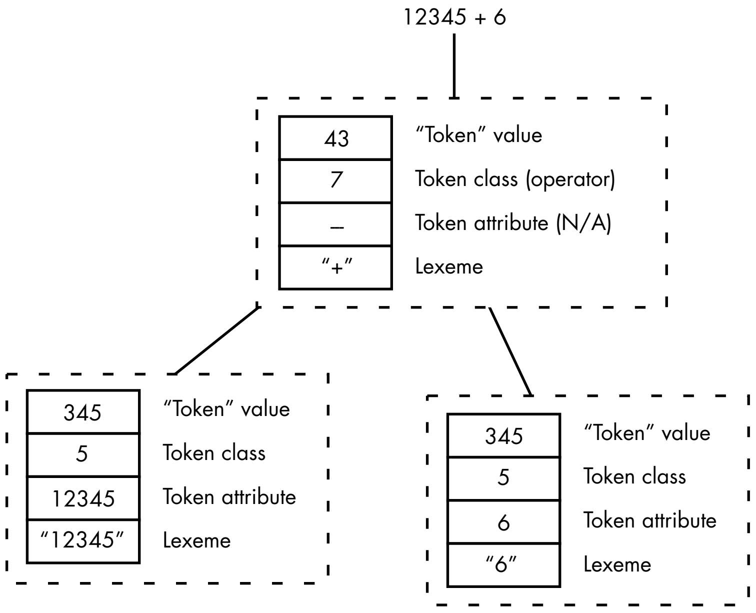


Figure 4-4: A portion of an abstract syntax tree

4.4.3 Intermediate Code Generation

The intermediate code generation phase is responsible for translating the AST representation of the source file into a quasi–machine code form. There are two reasons compilers typically translate a program into an intermediate form rather than converting it directly to native machine code.

First, the compiler's optimization phase can do certain types of optimizations, such as common subexpression elimination, much more easily on this intermediate form.

Second, many compilers, known as *cross-compilers*, generate executable machine code for several different CPUs. By breaking the code generation phase into two pieces—the intermediate code generator and the native code generator—the compiler writer can move all the CPU-independent activities into the intermediate code generation phase and write this code only once. This simplifies the native code generation phase. That is, because the compiler needs only one intermediate code generation phase but may need separate native code generation phases for each CPU the compiler supports, moving as much of the CPU-independent code as possible into the intermediate code generator will reduce the size of the native code generators. For the same reason, the optimization phase is often broken into two components (refer back to Figure 4-2): a CPU-independent component (the part following the intermediate code generator) and a CPU-dependent component.

Some language systems, such as Microsoft's VB.NET and C#, actually emit the intermediate code as the output of the compiler (in the .NET system, Microsoft calls this code *Common Intermediate Language*, or *CIL*). Native code generation and optimization are actually handled by the Microsoft *Common Language Runtime (CLR)* system, which performs JIT compilation on the CIL code the .NET compilers produce.

4.4.4 Optimization

The optimization phase, which follows intermediate code generation, translates the intermediate code into a more efficient form. This generally involves eliminating unnecessary entries from the AST. For example, the compiler's optimizer might transform the following intermediate code:

```
move the constant 5 into the variable i
move a copy of i into j
move a copy of j into k
add k to m
```

to something like:

```
move the constant 5 into k
add k to m
```

If there are no more references to `i` and `j`, the optimizer can eliminate all references to them. Indeed, if `k` is never used again, the optimizer can replace these two instructions with the single instruction `add 5 to m`. Note that this type of transformation is valid on nearly all CPUs. Therefore, this type of transformation/optimization is perfect for the first optimization phase.

4.4.4.1 The Problem with Optimization

Transforming intermediate code "into a more efficient form" is not a well-defined process—what makes one form of a program more efficient than another? The primary definition of efficiency is that the program minimizes the use of some system resource, usually memory (space) or CPU cycles (speed). A compiler's optimizer could manage other resources, but space and speed are the principal considerations for programmers. But even if we consider only these two facets of optimization, describing the "optimal" result is difficult. The problem is that optimizing for one goal (say, better performance) may create conflicts with another optimization goal (such as reduced memory usage). For this reason, the optimization process is usually a matter of compromise, where you make trade-offs and sacrifice certain subgoals (for example, running certain sections of the code a little slower) in order to create a reasonable result (like creating a program that doesn't consume too much memory).

4.4.4.2 Optimization's Effect on Compile Time

You might think that it's possible to set a single goal (for example, highest possible performance) and optimize strictly for that. However, the compiler must also be capable of producing an executable result in a reasonable amount of time. The optimization process is an example of what complexity theory calls an *NP-complete problem.* These are problems that are, as far as we know, intractable; that is, you cannot produce a guaranteed correct result (for example, an optimal version of a program) without first computing all possibilities and choosing from among them. Unfortunately, the time generally required to solve an NP-complete problem increases exponentially with the size of the input, which in the case of compiler optimization means roughly the number of lines of source code.

This means that in the worst case, producing a truly optimal program would take longer than it was worth. Adding one line of source code could approximately *double* the amount of time it takes to compile and optimize the code. Adding two lines could *quadruple* the amount of time. In fact, a full guaranteed optimization of a modern application could take longer than the known lifetime of the universe.

For all but the smallest source files (a few dozen lines), a perfect optimizer would take far too long to be of any practical value (and such optimizers have been written; search online for "superoptimizers" for details). For this reason, compiler optimizers rarely produce a truly optimal program. They simply produce the best result they can, given the limited amount of CPU time the user is willing to allow for the process.

NOTE *Languages that rely on JIT compilation (such as Java, C#, and VB.Net) move part of the optimization phase to runtime. Therefore, the optimizer's performance has a direct impact on the application's runtime. Because the JIT compiler system is running concurrently with the application, it cannot spend considerable time optimizing the code without having a huge impact on runtime. This is why languages such as Java and C#, even when ultimately compiled to low-level machine code, rarely perform as well as highly optimized code compiled by traditional languages such as C/C++ and Pascal.*

Rather than trying all possibilities and choosing the best result, modern optimizers use heuristics and case-based algorithms to determine the transformations they will apply to the machine code they produce. You need to be aware of these techniques so you can write your HLL code in a manner that allows an optimizer to easily process it and produce better machine code.

4.4.4.3 Basic Blocks, Reducible Code, and Optimization

Understanding how the compiler organizes the intermediate code (to output better machine code in later phases) is very important if you want to be able to help the optimizer do its job more efficiently. As control flows through the program, the optimizer keeps track of variable values in a process known as *data flow analysis (DFA)*. After careful DFA, a compiler can determine where a variable is uninitialized, when the variable contains certain values, when the program no longer uses the variable, and (just as importantly) when the compiler simply doesn't know anything about the variable's value. For example, consider the following Pascal code:

```
path := 5;
if( i = 2 ) then begin

    writeln( 'Path = ', path );

end;
i := path + 1;
if( i < 20 ) then begin

    path := path + 1;
    i := 0;

end;
```

A good optimizer will replace this code with something like the following:

```
if( i = 2 ) then begin

    (* Because the compiler knows that path = 5 *)

    writeln( 'path = ', 5 );

end;
i := 0;     (* Because the compiler knows that path < 20 *)
path := 6;  (* Because the compiler knows that path < 20 *)
```

In fact, the compiler probably would not generate code for the last two statements; instead, it would substitute the value `0` for `i` and `6` for `path` in later references. If this seems impressive to you, note that some compilers can even track constant assignments and expressions through nested function calls and complex expressions.

Although a complete description of how a compiler analyzes data flow is beyond the scope of this book, you should have a basic understanding of the process, because a sloppily written program can thwart the compiler's optimization abilities. Great code works synergistically with the compiler, not against it.

Some compilers can do some truly amazing things when it comes to optimizing high-level code. However, optimization is an inherently slow process. As noted earlier, it is an intractable problem. Fortunately, most programs don't require full optimization. Even if it runs a little slower than the optimal program, a good approximation is an acceptable compromise when compared to intractable compilation times.

The major concession to compilation time that compilers make during optimization is that they search for only so many possible improvements to a section of code before they move on. Therefore, if your programming style confuses the compiler, it may not be able to generate an optimal (or even close to optimal) executable because it has too many possibilities to consider. The trick, then, is to learn how compilers optimize the source file so you can accommodate them.

To analyze data flow, compilers divide the source code into sequences known as *basic blocks*—machine instructions into and out of which there are no branches except at the beginning and end. For example, consider the following C code:

```
x = 2;              // Basic block 1
j = 5;
i = f( &x, j );     // End of basic block 1
j = i * 2 + j;      // Basic block 2
if( j < 10 )        // End of basic block 2
{
    j = 0;          // Basic block 3
    i = i + 10;
    x = x + i;      // End of basic block 3
}
else
{
    temp = i;       // Basic block 4
    i = j;
    j = j + x;
    x = temp;       // End of basic block 4
}
x = x * 2;          // Basic block 5
++i;
--j;



printf( "i=%d, j=%d, x=%d\n", i, j, x ); // End basic block 5

// Basic block 6 begins here
```

This code snippet contains five basic blocks. Basic block 1 starts with the beginning of the source code. A basic block ends at the point where there is a jump into or out of the sequence of instructions. Basic block 1 ends at the call to the `f()` function. Basic block 2 starts with the statement following the call to the `f()` function, then ends at the beginning of the `if` statement because the `if` can transfer control to either of two locations. The `else` clause terminates basic block 3. It also marks the beginning of basic block 4 because there is a jump (from the `if`'s `then` clause) to the first statement following the `else` clause. Basic block 4 ends not because the code transfers control somewhere else, but because there is a jump from basic block 2 to the first statement that begins basic block 5 (from the `if`'s `then` section). Basic block 5 ends with a call to the C `printf()` function.

The easiest way to determine where the basic blocks begin and end is to consider the assembly code that the compiler will generate. Wherever there is a conditional branch/jump, unconditional jump, or call instruction, a basic block will end. Note, however, that the basic block includes the instruction that transfers control to a new location. A new basic block begins immediately after the instruction that transfers control to a new location. Also note that the target label of any conditional branch, unconditional jump, or call instruction begins a basic block.

Basic blocks make it easy for the compiler to track what's happening to variables and other program objects. As the compiler processes each statement, it can (symbolically) track the values that a variable will hold based upon their initial values and the computations on them within the basic block.

A problem occurs when the paths from two basic blocks join into a single code stream. For example, at the end of basic block 3 in the current example, the compiler could easily determine that the variable `j` contains zero because code in the basic block assigns the value `0` to `j` and then makes no other assignments to `j`. Similarly, at the end of basic block 3, the program knows that `j` contains the value `j0 + x0` (assuming `j0` represents the initial value of `j` upon entry into the basic block and `x0` represents the initial value of `x` upon entry into the block). But when the paths merge at the beginning of basic block 4, the compiler probably can't determine whether `j` will contain zero or the value `j0 + x0`. This means the compiler has to note that `j`'s value could be either of two different values at this point.

While keeping track of two possible values that a variable might contain at a given point is easy for a decent optimizer, it's not hard to imagine a situation where the compiler would have to keep track of many different possible values. In fact, if you have several `if` statements that the code executes sequentially, and each path through these `if` statements modifies a given variable, then the number of possible values for each variable doubles with each `if` statement. In other words, the number of possibilities increases exponentially with the number of `if` statements in a code sequence. At some point, the compiler cannot keep track of all the possible values a variable might contain, so it has to stop monitoring that information for the given variable. When this happens, there are fewer optimization possibilities that the compiler can consider.

Fortunately, although loops, conditional statements, `switch`/`case` statements, and procedure/function calls can increase the number of possible paths through the code exponentially, in practice compilers have few problems with typical well-written programs. This is because as paths from basic blocks converge, programs often make new assignments to their variables (thereby eliminating the old values the compiler was tracking). Compilers generally assume that programs rarely assign a different value to a variable along every distinct path in the program, and their internal data structures reflect this. Keep in mind that if you violate this assumption, the compiler may lose track of variable values and generate inferior code as a result.

Poorly structured programs can create control flow paths that confuse the compiler, reducing the opportunities for optimization. Good programs produce *reducible flow graphs*, pictorial depictions of the control flow path. Figure 4-5 is a flow graph for the previous code fragment.

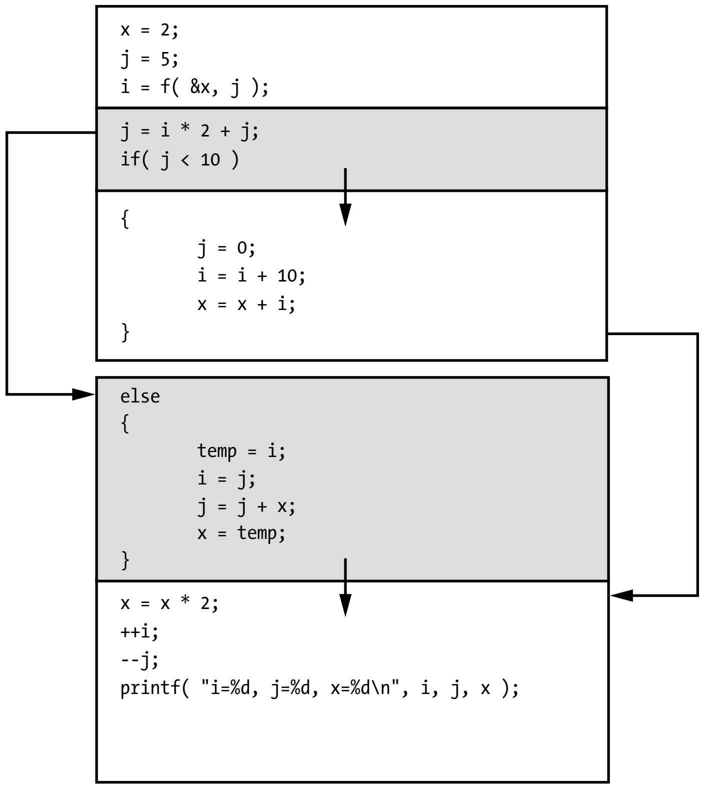


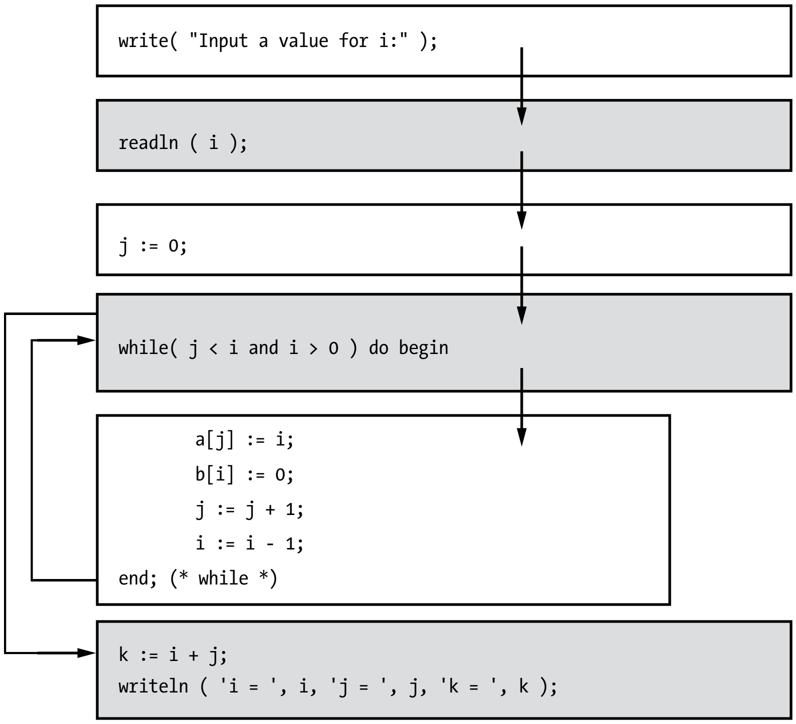
Figure 4-5: An example flow graph

As you can see, arrows connect the end of each basic block with the beginning of the basic block into which they transfer control. In this particular example, all of the arrows flow downward, but this isn't always the case. Loops, for example, transfer control backward in the flow graph. As another example, consider the following Pascal code:

```
write( 'Input a value for i:' );
readln( i );
j := 0;
while( ( j < i ) and ( i > 0 ) ) do begin

    a[j] := i;
    b[i] := 0;
    j := j + 1;
    i := i - 1;
end; (* while *)
k := i + j;
writeln( 'i = ', i, 'j = ', j, 'k = ', k );
```

Figure 4-6 shows the flow graph for this simple code fragment.


```
write( "Input a value for i:" );

readln ( i );

j := 0;

while( j < i and i > 0 ) do begin

       a[j] := i;
       b[i] := 0;
       j := j + 1;
       i := i - 1;
end; (* while *)

k := i + j;
writeln ( 'i = ', i, 'j = ', j, 'k = ', k );
```

Figure 4-6: Flow graph for a `while` *loop*

As mentioned, flow graphs in well-structured programs are *reducible.* Although a complete description of what constitutes a reducible flow graph is beyond the scope of this book, any program that consists only of structured control statements (`if`, `while`, `repeat..until`, and so on) and avoids `goto` statements will be reducible. This is an important point because compiler optimizers generally do a much better job when working on reducible programs. In contrast, programs that are not reducible tend to confuse them.

What makes reducible programs easier for optimizers to deal with is that their basic blocks can be collapsed in an outline fashion, with enclosing blocks inheriting properties (for example, which variables the block modifies) from the enclosed blocks. By processing the source file this way, the optimizer can deal with a small number of basic blocks rather than a large number of statements. This hierarchical approach to optimization is more efficient and allows the optimizer to maintain more information about the program's state. Furthermore, the exponential time complexity of the optimization problem works for us in this case. By reducing the number of blocks the code has to deal with, you dramatically decrease the amount of work the optimizer must do. Again, the exact details of how the compiler achieves this are not important here. The takeaway is that making your programs reducible enables the optimizer to do its job more effectively. Attempts to "optimize" your code by sticking in lots of `goto` statements—to avoid duplicating code and executing unnecessary tests—may actually work against you. While you may save a few bytes or a few cycles in the immediate area you're working on, the end result might confuse the compiler enough that it cannot optimize as well, causing an overall loss of efficiency.

4.4.4.4 Common Compiler Optimizations

Chapter 12 will provide complete definitions and examples of common compiler optimizations in programming contexts where compilers typically use them. But for now, here's a quick preview of the basic types:

Constant folding
Constant folding computes the value of constant expressions or subexpressions at compile time rather than at runtime. See "Constant Folding" on page 397 for more information.

Constant propagation
Constant propagation replaces a variable with a constant value if the compiler determines that the program assigned that constant to the variable earlier in the code. See "Constant Propagation" on page 400 for more information.

Dead code elimination
Dead code elimination removes the object code associated with a particular source code statement when the program will never use the result of that statement, or when a conditional block will never be `true`. See "Dead Code Elimination" on page 404 for more information.

Common subexpression elimination
Frequently, part of an expression will appear elsewhere in the current function; this is known as a *subexpression*. If the values of the variables in a subexpression haven't changed, the program does not need to recompute them everywhere the subexpression appears. The program can simply save the subexpression's value on the first evaluation and then use it for every other occurrence of the subexpression. See "Common Subexpression Elimination" on page 410 for more information.

Strength reduction
Often, the CPU can directly compute a value using a different operator than the source code specifies. For example, a `shift` instruction can implement multiplication or division by a constant that is a power of 2, and a bitwise `and` instruction can compute certain modulo (remainder) operations (the `shift` and `and` instructions generally execute much faster than the multiply and divide instructions). Most compiler optimizers

are good at recognizing such operations and replacing the more expensive computation with a less expensive sequence of machine instructions. See "Strength Reduction" on page 417 for more information.

Induction
In many expressions, particularly those appearing within a loop, the value of one variable in the expression is completely dependent upon some other variable. Frequently, the compiler can eliminate the computation of the new value or merge the two computations into one for the duration of that loop. See "Induction" on page 422 for more information.

Loop invariants
The optimizations so far have all been techniques a compiler can use to improve code that is already well written. Handling loop invariants, by contrast, is a compiler optimization for fixing bad code. A *loop invariant* is an expression that does not change on each iteration of some loop. An optimizer can compute the result of such a calculation just once, outside the loop, and then use the computed value within the loop's body. Many optimizers are smart enough to discover loop invariant calculations and can use *code motion* to move the invariant calculation outside the loop. See "Loop Invariants" on page 427 for more information.

Good compilers can perform many other optimizations, but these are the standard optimizations that any decent compiler should be able to do.

4.4.4.5 Compiler Optimization Control

By default, most compilers do very little or no optimization unless you explicitly tell them to. This might seem counterintuitive; after all, we generally want compilers to produce the best possible code for us. However, there are many definitions of "optimal," and no single compiler output is going to satisfy every possible one.

You might argue that some sort of optimization, even if it's not the particular type you're interested in, is better than none at all. However, there are a few reasons why no optimization is a compiler's default state:

- Optimization is a slow process. You get quicker turnaround times on compiles when you have the optimizer turned off. This can be a big help during rapid edit-compile-test cycles.
- Many debuggers don't work properly with optimized code, and you have to turn off optimization in order to use a debugger on your application (this also makes analyzing the compiler output much easier).
- Most compiler defects occur in the optimizer. By emitting unoptimized code, you're less likely to encounter defects in the compiler (then again, the compiler's author is less likely to be notified about defects in the compiler, too).

Most compilers provide command-line options that let you control the types of optimization the compiler performs. Early C compilers under Unix used command-line arguments like `-O`, `-O1`, and `-O2`. Many later compilers (C and otherwise) have adopted this strategy, if not exactly the same command-line options.

If you're wondering why a compiler might offer multiple options to control optimization rather than just a single option (optimization or no optimization), remember that "optimal" means different things to different people. Some people might want code that is optimized for space; others might want code that is optimized for speed (and those two optimizations could be mutually exclusive in a given situation). Some people might want to optimize their files but don't want the compiler to take forever to process them, so they'd be willing to compromise with a small set of fast optimizations. Others might want to control optimization for a specific member of a CPU family (such as the Core i9 processor in the 80x86 family). Furthermore, some optimizations are "safe" (that is, they always produce correct code) only if the program is written in a certain way. You certainly don't want to enable such optimizations unless the programmer guarantees that they've written their code accordingly. Finally, for programmers who are writing their HLL code carefully, some optimizations the compiler performs may actually produce *inferior* code, in which case the ability to specify optimizations is very handy. For these reasons and more, most modern compilers provide considerable flexibility over the types of optimizations they perform.

Consider the Microsoft Visual C++ compiler. It provides the following command-line options to control optimization:

```
                              -OPTIMIZATION-

/O1 minimize space
/O2 maximize speed
/Ob<n> inline expansion (default n=0)
/Od disable optimizations (default)
/Og enable global optimization
/Oi[-] enable intrinsic functions
/Os favor code space
/Ot favor code speed
/Ox maximum optimizations
/favor:<blend|AMD64|INTEL64|ATOM> select processor to optimize for, one of:
    blend - a combination of optimizations for several different x64 processors
    AMD64 - 64-bit AMD processors
    INTEL64 - Intel(R)64 architecture processors
    ATOM - Intel(R) Atom(TM) processors

                             -CODE GENERATION-

/Gw[-] separate global variables for linker
/GF enable read-only string pooling
/Gm[-] enable minimal rebuild
/Gy[-] separate functions for linker
/GS[-] enable security checks
```

```
/GR[-] enable C++ RTTI
/GX[-] enable C++ EH (same as /EHsc)
/guard:cf[-] enable CFG (control flow guard)
/EHs enable C++ EH (no SEH exceptions)
/EHa enable C++ EH (w/ SEH exceptions)
/EHc extern "C" defaults to nothrow
/EHr always generate noexcept runtime termination checks
/fp:<except[-]|fast|precise|strict> choose floating-point model:
    except[-] - consider floating-point exceptions when generating code
    fast - "fast" floating-point model; results are less predictable
    precise - "precise" floating-point model; results are predictable
    strict - "strict" floating-point model (implies /fp:except)
/Qfast_transcendentals generate inline FP intrinsics even with /fp:except
/Qspectre[-] enable mitigations for CVE 2017-5753
/Qpar[-] enable parallel code generation
/Qpar-report:1 auto-parallelizer diagnostic; indicate parallelized loops
/Qpar-report:2 auto-parallelizer diagnostic; indicate loops not parallelized
/Qvec-report:1 auto-vectorizer diagnostic; indicate vectorized loops
/Qvec-report:2 auto-vectorizer diagnostic; indicate loops not vectorized
/GL[-] enable link-time code generation
/volatile:<iso|ms> choose volatile model:
    iso - Acquire/release semantics not guaranteed on volatile accesses
    ms  - Acquire/release semantics guaranteed on volatile accesses
/GA optimize for Windows Application
/Ge force stack checking for all funcs
/Gs[num] control stack checking calls
/Gh enable _penter function call
/GH enable _pexit function call
/GT generate fiber-safe TLS accesses
/RTC1 Enable fast checks (/RTCsu)
/RTCc Convert to smaller type checks
/RTCs Stack Frame runtime checking
/RTCu Uninitialized local usage checks
/clr[:option] compile for common language runtime, where option is:
    pure - produce IL-only output file (no native executable code)
    safe - produce IL-only verifiable output file
    initialAppDomain - enable initial AppDomain behavior of Visual C++ 2002
    noAssembly - do not produce an assembly
    nostdlib - ignore the default \clr directory
/homeparams Force parameters passed in registers to be written to the stack
/GZ Enable stack checks (/RTCs)
/arch:AVX enable use of instructions available with AVX-enabled CPUs
/arch:AVX2 enable use of instructions available with AVX2-enabled CPUs
/Gv __vectorcall calling convention
```

GCC has a comparable—though much longer—list, which you can view by specifying `-v --help` on the GCC command line. Most of the individual optimization flags begin with `-f`. You can also use `-O`*n*, where *n* is a single digit integer value, to specify different levels of optimization. Take care when using `-O3` (or higher), as doing so may perform some unsafe optimizations in certain cases.

4.4.5 Compiler Benchmarking

One real-world constraint on our ability to produce great code is that different compilers provide a wildly varying set of optimizations. Even the same optimizations performed by two different compilers can differ greatly in effectiveness.

Fortunately, several websites have benchmarked various compilers. (One good example is Willus.com.) Simply search online for a topic like "compiler benchmarks" or "compiler comparisons" and have fun.

4.4.6 Native Code Generation

The native code generation phase is responsible for translating the intermediate code into machine code for the target CPU. An 80x86 native code generator, for example, might translate the intermediate code sequence given previously into something like the following:

```
mov( 5, eax ); // move the constant 5 into the EAX register.
mov( eax, k ); // Store the value in EAX (5) into k.
add( eax, m ); // Add the value in EAX to variable m.
```

The second optimization phase, which takes place after native code generation, handles machine idiosyncrasies that don't exist on all machines. For example, an optimizer for a Pentium II processor might replace an instruction of the form `add(1, eax);` with the instruction `inc(eax);`. Optimizers for later CPUs might do just the opposite. Optimizers for certain 80x86 processors might arrange the sequence of instructions one way to maximize parallel execution of the instructions in a superscalar CPU, while an optimizer targeting a different (80x86) CPU might arrange the instructions differently.

4.5 Compiler Output

The previous section stated that compilers typically produce machine code as their output. Strictly speaking, this is neither necessary nor even that common. Most compiler output is not code that a given CPU can directly execute. Some compilers emit assembly language source code, which requires further processing by an assembler prior to execution. Other compilers produce an object file, which is similar to executable code but is not directly executable. Still other compilers actually produce source code output that requires further processing by a different HLL compiler. I'll discuss these different output formats and their advantages and disadvantages in this section.

4.5.1 Emitting HLL Code as Compiler Output

Some compilers actually emit output that is source code for a different high-level programming language (see Figure 4-7). For example, many compilers (including the original C++ compiler) emit C code as their output. Indeed, compiler writers who emit some high-level source code from their compiler frequently choose the C programming language.

Emitting HLL code as compiler output offers several advantages. The output is human-readable and generally easy to verify. The HLL code emitted is often portable across various platforms; for example, if a compiler emits C code, you can usually compile that output on several different machines because C compilers exist for most platforms. Finally, by emitting HLL code, a translator can rely on the optimizer of the target language's compiler, thereby saving the effort of writing its own optimizer. In other words, emitting HLL code allows a compiler writer to create a less complex code generator module and rely on the robustness of some other compiler for the most complex part of the compilation process.

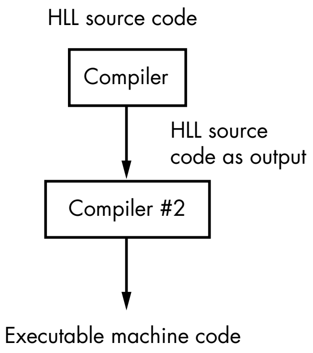


Figure 4-7: Emission of HLL code by a compiler

Emitting HLL code has several disadvantages, too. First and foremost, this approach usually takes more processing time than directly generating executable code. To produce an executable file, a second, otherwise unnecessary, compiler might be needed. Worse, the output of that second compiler might need to be processed further by another compiler or assembler, exacerbating the problem. Another disadvantage is that in HLL code it's difficult to embed debugging information that a debugger program can use. Perhaps the most fundamental problem with this approach, however, is that HLLs are usually an abstraction of the underlying machine. Therefore, it could be quite difficult for a compiler to emit statements in an HLL that efficiently map to low-level machine code.

Generally, compilers that emit HLL statements as their output are translating a *very high-level language (VHLL)* into a lower-level language. For example, C is often considered to be a fairly low-level HLL, which is one reason why it's a popular output format for many compilers. Projects that have attempted to create a special, portable, low-level language specifically for this purpose have never been enormously popular. Check out any of the "C- -" projects on the internet for examples of such systems.

If you want to write efficient code by analyzing compiler output, you'll probably find it more difficult to work with compilers that output HLL code. With a standard compiler, all you have to learn is the particular machine code statements that your compiler produces. However, when a compiler emits HLL statements as its output, learning to write great code with that compiler is more difficult. You need to understand both how the main language emits the HLL statements and how the second compiler translates the code into machine code.

Generally, compilers that produce HLL code as their output are either experimental compilers for VHLLs, or compilers that attempt to translate legacy code in an older language to a more modern language (for example, FORTRAN to C). As a result, expecting those compilers to emit efficient code is generally asking too much. Thus, you'd probably be wise to avoid a compiler that emits HLL statements. A compiler that directly generates machine code (or assembly language code) is more likely to produce smaller and faster-running executables.

4.5.2 Emitting Assembly Language as Compiler Output

Many compilers will emit human-readable assembly language source files rather than binary machine code files (see Figure 4-8). Probably the most famous example is the FSF/GNU GCC compiler suite, which emits assembly language output for the FSF/GNU assembler Gas. Like compilers that emit HLL source code, emitting assembly language has some advantages and disadvantages.

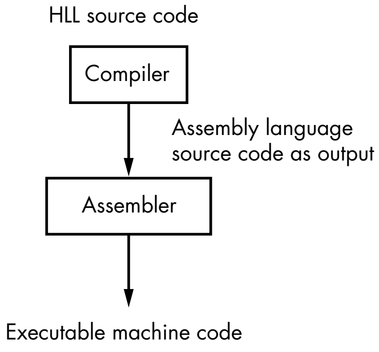


Figure 4-8: Emission of assembly code by a compiler

The principal disadvantages to emitting assembly code are similar to the downsides of emitting HLL source output. First, you have to run a second language translator (namely the assembler) to produce the actual object code for execution. Second, some assemblers may not allow the embedding of debugging metadata that allows a debugger to work with the original source code (though many assemblers do support this capability). These two disadvantages turn out to be minimal if a compiler emits code for an appropriate assembler. For example, Gas is very fast and supports the insertion of debug information for use by source-level debuggers. Therefore, the FSF/GNU compilers don't suffer as a result of emitting Gas output.

The advantage of assembly language output, particularly for our purposes, is that it's easy to read the compiler's output and determine which machine instructions the compiler emits. Indeed, this book uses this compiler facility to analyze compiler output. Emitting assembly code frees the compiler writer from having to worry about several different object code output formats—the underlying assembler handles that—which allows the compiler writer to create a more portable compiler. True, the assembler has to be capable of generating code for different operating systems, but you only need to repeat this exercise once for each object file format, rather than once for each format multiplied by the number of compilers you write. The FSF/GNU compiler suite has taken good advantage of the Unix philosophy of using small tools that chain together to accomplish larger, more complicated tasks—that is, minimize redundancy.

Another advantage of compilers that can emit assembly language output is that they generally allow you to embed *inline assembly language* statements in the HLL code. This lets you insert a few machine instructions directly into time-critical sections of your code without the hassle of having to create a separate assembly language program and link its output to your HLL program.

4.5.3 Emitting Object Files as Compiler Output

Most compilers translate the source language into an object file format, an intermediate file format that contains machine instructions and binary runtime data along with certain metadata. This metadata allows a linker/loader program to combine various object modules to produce a complete executable. This in turn allows programmers to link *library modules* and other object modules that they've written and compiled separately from their main application module.

The advantage of object file output is that you don't need a separate compiler or assembler to convert the compiler's output to object code form, which saves a little time during compilation. Note, however, that a linker program must still process the object file output, which consumes a little time after compilation. Nevertheless, linkers are generally quite fast, so it's usually more cost-effective to compile a single module and link it with several previously compiled modules than it is to compile all the modules together to form an executable file.

Object modules are binary files and do not contain human-readable data, so it's a bit more difficult to analyze compiler output in this format than in the others we've discussed. Fortunately, there are utility programs that will disassemble the output of an object module into a human-readable form. The result isn't as easy to read as straight assembly compiler output, but you can still do a reasonably good job.

Because object files are challenging to analyze, many compiler writers provide an option to emit assembly code instead of object code. This handy feature makes analysis much easier, so we'll use it with various compilers throughout this book.

NOTE *The section "Object File Formats" on page 71 provides a detailed look at the elements of an object file, focusing on COFF (Common Object File Format).*

4.5.4 Emitting Executable Files as Compiler Output

Some compilers directly emit an executable output file. These compilers are often very fast, producing almost instantaneous turnaround during the edit-compile-run-test-debug cycle. Unfortunately, their output is often the most difficult to read and analyze, requiring the use of a debugger or disassembler program and a lot of manual work. Nevertheless, the fast turnaround makes these compilers popular, so later in this book, we'll look at how to analyze the executable files they produce.

4.6 Object File Formats

As previously noted, object files are among the most popular forms of compiler output. Even though it is possible to create a proprietary object file format—one that only a single compiler and its associated tools can use—most compilers emit code using one or more standardized object file formats. This allows different compilers to share the same set of object file utilities, including linkers, librarians, dump utilities, and disassemblers. Examples of common object file formats include: OMF (Object Module Format), COFF (Common Object File Format), PE/COFF (Microsoft's Portable Executable variant on COFF), and ELF (Executable and Linkable Format). There are several variants of these file formats, as well as many altogether different object file formats.

Most programmers understand that object files represent the machine code that an application executes, but they often don't realize the impact of the object file's organization on their application's performance and size. Although you don't need to have detailed knowledge of an object file's internal representation to write great code, having a basic understanding will help you organize your source files to better take advantage of the way compilers and assemblers generate code for your applications.

An object file usually begins with a header that comprises the first few bytes of the file. This header contains certain *signature information* that identifies the file as a valid object file, along with several other values that define the location of various data structures in the file. Beyond the header, an object file is usually divided into several sections, each containing application data, machine instructions, symbol table entries, relocation data, and other metadata (data about the program). In some cases, the actual code and data represent only a small part of the entire object code file.

To get a feeling for how object files are structured, it's worthwhile to look at a specific object file format in detail. I'll use COFF in the following discussion because most object file formats (for example, ELF and PE/COFF) are based on, or very similar to, COFF. The basic layout of a COFF file is shown in Figure 4-9, after which I'll describe each section in turn.

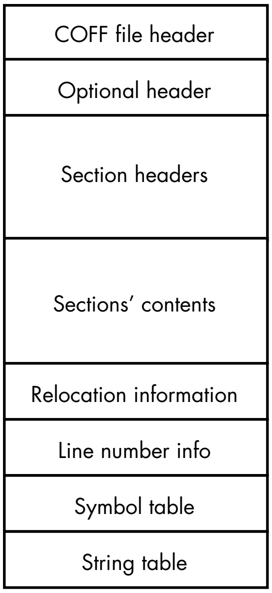


Figure 4-9: Layout of a COFF file

4.6.1 The COFF File Header

At the beginning of every COFF file is a *COFF file header*. Here are the definitions that Microsoft Windows and Linux use for the COFF header structure:

```
// Microsoft Windows winnt.h version:

typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;

// Linux coff.h version:

struct COFF_filehdr {
        char f_magic[2];        /* magic number */
        char f_nscns[2];        /* number of sections */
        char f_timdat[4];       /* time & date stamp */
        char f_symptr[4];       /* file pointer to symtab */
        char f_nsyms[4];        /* number of symtab entries */
        char f_opthdr[2];       /* sizeof(optional hdr) */
        char f_flags[2];        /* flags */
};
```

The Linux *coff.h* header file uses traditional Unix names for these fields; the Microsoft *winnt.h* header file uses (arguably) more readable names. Here's a summary of each field in the header, with Unix names to the left of the slash and Microsoft equivalents to the right:

`f_magic/Machine`

Identifies the system for which this COFF file was created. In the original Unix definition, this value identified the particular Unix port for which the code was created. Today's operating systems define this value somewhat differently, but the bottom line is that this value is a signature that specifies whether the COFF file contains data or machine instructions that are appropriate for the current operating system and CPU.

Table 4-1 provides the encodings for the `f_magic/Machine` field.

Table 4-1: `f_magic/Machine` Field Encoding

Value	Description
0x14c	Intel 386
0x8664	x86-64
0x162	MIPS R3000
0x168	MIPS R10000
0x169	MIPS little endian WCI v2
0x183	old Alpha AXP
0x184	Alpha AXP
0x1a2	Hitachi SH3
0x1a3	Hitachi SH3 DSP
0x1a6	Hitachi SH4
0x1a8	Hitachi SH5
0x1c0	ARM little endian
0x1c2	Thumb
0x1c4	ARMv7
0x1d3	Matsushita AM33
0x1f0	PowerPC little endian
0x1f1	PowerPC with floating-point support
0x200	Intel IA64
0x266	MIPS16
0x268	Motorola 68000 series
0x284	Alpha AXP 64-bit
0x366	MIPS with FPU
0x466	MIPS16 with FPU
0xebc	EFI bytecode
0x8664	AMD AMD64
0x9041	Mitsubishi M32R little endian
0xaa64	ARM64 little endian
0xc0ee	CLR pure MSIL

`f_nscns/NumberOfSections`
: Specifies how many segments (sections) are present in the COFF file. A linker program can iterate through a set of section headers (described a little later) using this value.

`f_timdat/TimeDateStamp`
: Contains a Unix-style timestamp (number of seconds since January 1, 1970) value specifying the file's creation date and time.

`f_symptr/PointerToSymbolTable`
: Contains a file offset value (that is, the number of bytes from the beginning of the file) that specifies where the *symbol table* begins in the file. The symbol table is a data structure that specifies the names and other information about all external, global, and other symbols used by the code in the COFF file. Linkers use the symbol table to resolve external references. This symbol table information may also appear in the final executable file for use by a symbolic debugger.

`f_nsyms/NumberOfSymbols`
: The number of entries in the symbol table.

`f_opthdr/SizeOfOptionalHeader`
: Specifies the size of the optional header that immediately follows the file header (that is, the first byte of the optional header immediately follows the `f_flags/Characteristics` field in the file header structure). A linker or other object code manipulation program would use the value in this field to determine where the optional header ends and the section headers begin in the file. The section headers immediately follow the optional header, but the optional header's size isn't fixed. Different implementations of a COFF file can have different optional header structures. If the optional header is not present in a COFF file, the `f_opthdr/SizeOfOptionalHeader` field will contain zero, and the first section header will immediately follow the file header.

`f_flags/Characteristics`
: A small bitmap that specifies certain Boolean flags, such as whether the file is executable, whether it contains symbol information, and whether it contains line number information (for use by debuggers).

4.6.2 The COFF Optional Header

The COFF optional header contains information pertinent to executable files. This header may not be present if the file contains object code that is not executable (because of unresolved references). Note, however, that this optional header is always present in Linux COFF and Microsoft PE/COFF files, even when the file is not executable. The Windows and Linux structures for this optional file header take the following forms in C.

```
// Microsoft PE/COFF Optional Header (from winnt.h)

typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;


// Linux/COFF Optional Header format (from coff.h)

typedef struct
{
  char  magic[2];  /* type of file */
  char  vstamp[2]; /* version stamp */
  char  tsize[4];  /* text size in bytes, padded to
                      FW bdry */
  char  dsize[4]; /* initialized   data "   " */
  char  bsize[4]; /* uninitialized data "   " */
  char  entry[4]; /* entry pt. */
  char  text_start[4]; /* base of text used for this file */
  char  data_start[4]; /* base of data used for this file */
} COFF_AOUTHDR;
```

The first thing to notice is that these structures are not identical. The Microsoft version has considerably more information than the Linux version. The `f_opthdr/SizeOfOptionalHeader` field exists in the file header to determine the actual size of the optional header.

`magic/Magic`

Provides yet another signature value for the COFF file. This signature value identifies the file type (that is, COFF) rather than the system under which it was created. Linkers use the value of this field to determine if they are truly operating on a COFF file (instead of some arbitrary file that would confuse the linker).

`vstamp/MajorLinkerVersion/MinorLinkerVersion`

Specifies the version number of the COFF format so that a linker written for an older version of the file format won't try to process files intended for newer linkers.

`tsize/SizeOfCode`

Attempts to specify the size of the code section found in the file. If the COFF file contains more than one code section, the value of this field is undefined, although it usually specifies the size of the first code/text section in the COFF file.

`dsize/SizeOfInitializedData`

Specifies the size of the data segment appearing in this COFF file. Once again, this field is undefined if there are two or more data sections in the file. Usually, this field specifies the size of the first data section if there are multiple data sections.

`bsize/SizeOfUninitializedData`

Specifies the size of the *block started by symbol (BSS)* section—the uninitialized data section—in the COFF file. As for the text and data sections, this field is undefined if there are two or more BSS sections; in such cases this field usually specifies the size of the first BSS section in the file.

NOTE *See "Pages, Segments, and File Size" on page 81 for more on BSS sections.*

`entry/AddressOfEntryPoint`

Contains the starting address of the executable program. Like other pointers in the COFF file header, this field is actually an offset into the file; it is not an actual memory address.

`text_start/BaseOfCode`

Specifies a file offset into the COFF file where the code section begins. If there are two or more code sections, this field is undefined, but it generally specifies the offset to the first code section in the COFF file.

`data_start/BaseOfData`

Specifies a file offset into the COFF file where the data section begins. If there are two or more data sections, this field is undefined, but it generally specifies the offset to the first data section in the COFF file.

There is no need for a `bss_start/StartOfUninitializedData` field. The COFF file format assumes that the operating system's program loader will automatically allocate storage for a BSS section when the program loads into memory. There is no need to consume space in the COFF file for uninitialized data (however, "Executable File Formats" on page 80 describes how some compilers actually merge BSS and DATA sections together for performance reasons).

The optional file header structure is actually a throwback to the *a.out* format, an older object file format used in Unix systems. This is why it doesn't handle multiple text/code and data sections, even though COFF allows them.

The remaining fields in the Windows variant of the optional header hold values that Windows linkers allow programmers to specify. While their purposes will likely be clear to anyone who has manually run Microsoft's linker from a command line, those are not important here. What is important is that COFF does not require a specific data structure for the optional header. Different implementations of COFF (such as Microsoft's) may freely extend the definition of the optional header.

4.6.3 COFF Section Headers

The section headers follow the optional header in a COFF file. Unlike the file and optional headers, a COFF file may contain multiple section headers. The `f_nscns/NumberOfSections` field in the file header specifies the exact number of section headers (and, therefore, sections) found in the COFF file. Keep in mind that the first section header does not begin at a fixed offset in the file. Because the optional header's size is variable (and, in fact, could even be 0 if it isn't present), you have to add the `f_opthdr/SizeOfOptionalHeader` field in the file header to the size of the file header to get the starting offset of the first section header. Section headers are a fixed size, so once you have the address of the first section header you can easily compute the address of any other by multiplying the desired section header number by the section header size and adding the result to the base offset of the first section header.

Here are the C struct definitions for Windows and Linux section headers:

```
// Windows section header structure (from winnt.h)

typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;


// Linux section header definition (from coff.h)

struct COFF_scnhdr
{
  char s_name[8]; /* section name */
  char s_paddr[4]; /* physical address, aliased s_nlib */
  char s_vaddr[4]; /* virtual address */
  char s_size[4]; /* section size */
  char s_scnptr[4]; /* file ptr to raw data */
  char s_relptr[4]; /* file ptr to relocation */
  char s_lnnoptr[4]; /* file ptr to line numbers */
  char s_nreloc[2]; /* number of relocation entries */
  char s_nlnno[2]; /* number of line number entries */
  char s_flags[4]; /* flags */
};
```

If you inspect these two structures closely, you'll find that they are roughly equivalent (the only structural difference is that Windows overloads the physical address field, which in Linux is always equivalent to the `VirtualAddress` field, to hold a `VirtualSize` field).

Here's a summary of each field:

`s_name/Name`

Specifies the name of the section. As is apparent in the Linux definition, this field is limited to eight characters and, accordingly, section names will be a maximum of eight characters long. (Usually, if a source file specifies a longer name, the compiler/assembler will truncate it to 8 characters when creating the COFF file.) If the section name is exactly eight characters long, those eight characters will consume all 8 bytes of this field and there will be no zero-terminating byte. If the section name is shorter than eight characters, a zero-terminating byte will follow the name. The value of this field is often something like `.text`, `CODE`, `.data`, or `DATA`. Note, however, that the name does not define the segment's type. You could create a code/text section and name it `DATA`; you could also create a data section and name it `.text` or `CODE`. The `s_flags/Characteristics` field determines the actual type of this section.

`s_paddr/PhysicalAddress/VirtualSize`
: Not used by most tools. Under Unix-like operating systems (such as Linux), this field is usually set to the same value as the `VirtualAddress` field. Different Windows tools set this field to different values (including zero); the linker/loader seems to ignore whatever value appears here.

`s_vaddr/VirtualAddress`
: Specifies the section's loading address in memory (that is, its virtual memory address). Note that this is a runtime memory address, not an offset into the file. The program loader uses this value to determine where to load the section into memory.

`s_size/SizeOfRawData`
: Specifies the size, in bytes, of the section.

`s_scnptr/PointerToRawData`
: Provides the file offset to the start of the section's data in the COFF file.

`s_relptr/PointerToRelocations`
: Provides a file offset to the relocation list for this particular section.

`s_lnnoptr/PointerToLinenumbers`
: Contains a file offset to the line number records for the current section.

`s_nreloc/NumberOfRelocations`
: Specifies the number of *relocation entries* found at that file offset. Relocation entries are small structures that provide file offsets to address data in the section's data area that must be patched when the file is loaded into memory. We won't discuss these relocation entries in this book, but if you're interested in more details, see the references at the end of this chapter.

`s_nlnno/NumberOfLinenumbers`
: Specifies how many line number records can be found at that offset. Line number information is used by debuggers and is beyond the scope of this chapter. Again, see the references at the end of this chapter if you're interested in more information about the line number entries.

`s_flags/Characteristics`
: A bitmap that specifies the characteristics of this section. In particular, this field will tell you whether the section requires relocation, whether it contains code, whether it is read-only, and so on.

4.6.4 COFF Sections

The section headers provide a directory that describes the actual data and code found in the object file. The `s_scnptr/PointerToRawData` field contains a file offset to where the raw binary data or code is sitting in the file, and the `s_size/SizeOfRawData` field specifies the length of the section's data. Due to relocation requirements, the data actually appearing in the section block may not be an exact representation of the data that the operating system loads into memory. This is because many instruction operand addresses and pointer values appearing in the section may need to be *patched* to relocate the file based on where the OS loads it into memory. The relocation list (which is separate from the section's data) contains offsets into the section where the OS must patch the relocatable addresses. The OS performs this patching when loading the section's data from disk.

Although the bytes in a COFF section may not be an exact representation of the data in memory at runtime, the COFF format requires that all of the bytes in the section *map* to the corresponding address in memory. This allows the loader to copy the section's data directly from the file into sequential memory locations. The relocation operation never inserts or deletes bytes in a section; it only changes the values of certain bytes in the section. This requirement helps simplify the system loader and improves application performance because the operating system doesn't have to move large blocks of memory around when loading the application into memory. The drawback to this scheme is that the COFF format misses the opportunity to compress redundant data appearing in the section's data area. The COFF designers felt it was more important to emphasize performance over space in their design.

4.6.5 The Relocation Section

The relocation section in the COFF file contains the offsets to the pointers in the COFF sections that must be relocated when the system loads those sections' code or data into memory.

4.6.6 Debugging and Symbolic Information

The last three sections shown in Figure 4-9 contain information that debuggers and linkers use. One section contains line number information that a debugger uses to correlate lines of source code with the executable machine code instructions. The symbol table and string table sections hold the public and external symbols for the COFF file. Linkers use this information to resolve external references between object modules; debuggers use this information to display symbolic variable and function names during debugging.

NOTE *This book doesn't provide a complete description of the COFF file format, but you'll definitely want to dig deeper into it and other object code formats (ELF, MACH-O, OMF, and so on) if you're interested in writing applications such as assemblers, compilers, and linkers. To study this area further, see the references at the end of this chapter.*

4.7 Executable File Formats

Most operating systems use a special file format for executable files. Often, the executable file format is similar to the object file format, the principal difference being that there are usually no unresolved external references in the executable file.

In addition to machine code and binary data, executable files contain other metadata, including debugging information, linkage information for dynamically linked libraries, and details about how the operating system should load different sections of the file into memory. Depending on the CPU and OS, the executable files may also contain relocation information so that the OS can patch absolute addresses when it loads the file into memory. Object code files contain the same information, so it's no surprise that the executable file formats used by many operating systems are similar to their object file formats.

The Executable and Linkable Format (ELF), employed by Linux, QNX, and other Unix-like operating systems, is very typical of a combined object file format and executable format. Indeed, the name of the format suggests its dual nature. As another example, Microsoft's PE file format is a straightforward variant of the COFF format. The similarity between the object and executable file formats allows the OS designer to share code between the loader (responsible for executing the program) and linker applications. Given this similarity, there's little reason to discuss the specific data structures found in an executable file, as doing so would largely repeat the information from the previous sections.

However, there's one very practical difference in the layout of these two types of files worth mentioning. Object files are usually designed to be as small as possible, while executable files are usually designed to load into memory as fast as possible, even if this means that they're larger than absolutely necessary. It may seem paradoxical that a larger file could load into memory faster than a smaller file; however, the OS might load only a small part of the executable file into memory at one time if it supports virtual memory. As we'll discuss next, a well-designed executable file format can take advantage of this fact by laying out the data and machine instructions in the file to reduce virtual memory overhead.

4.7.1 Pages, Segments, and File Size

Virtual memory subsystems and memory protection schemes generally operate on *pages* in memory. A page on a typical processor is usually between 1KB and 64KB in size. Whatever the size, a page is the smallest unit of memory to which you can apply discrete protection features (such as whether the data in that page is read-only, read/write, or executable). In particular, you cannot mix read-only/executable code with read/write data in the same page—the two must appear in separate pages in memory. Using the 80x86 CPU family as an example, pages in memory are 4KB each. Therefore, the minimum amount of code space and the minimum amount of data space we can allocate to a process is 8KB if we have read/write data and we want to place the machine instructions in read-only memory. In fact, most programs contain several segments or sections (as you saw previously with object files) to which we can apply individual protection rights, and each section will require a unique set of one or more pages in memory that are not shared with any of the other sections. A typical program has four or more sections in memory: code or text, static data, uninitialized data, and stack are the most common. In addition, many compilers also generate heap segments, linkage segments, read-only segments, constant data segments, and application-named data segments (see Figure 4-10).

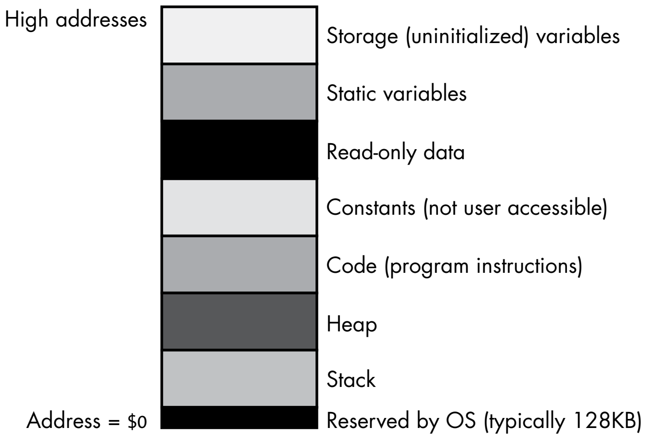

High addresses
Storage (uninitialized) variables
Static variables
Read-only data
Constants (not user accessible)
Code (program instructions)
Heap
Stack
Address = $0
Reserved by OS (typically 128KB)

Figure 4-10: Typical segments found in memory

Because operating systems map segments to pages, a segment will always require some number of bytes that is a multiple of the page size. For example, if a program has a segment that contains only a single byte of data, that segment will still consume 4,096 bytes on an 80x86 processor. Similarly, if an 80x86 application consists of six different segments, that application will consume at least 24KB in memory, regardless of the number of machine instructions and data bytes the program uses and regardless of the executable file's size.

Many executable file formats (such as ELF and PE/COFF) provide an option in memory for a BSS section where a programmer can place uninitialized static variables. Because the values are uninitialized, there is no need to clutter the executable file with random data values for each of these variables. Therefore, the BSS section in some executable file formats is just a small stub that tells the OS loader the size of the BSS section. This way, you can add new uninitialized static variables to your application without affecting the executable file's size. When you increase the amount of BSS data, the compiler simply adjusts a value to tell the loader how many bytes to reserve for the uninitialized variables. Were you to add those same variables to an initialized data section, the size of the executable file would grow with each byte of data that you added. Obviously, saving space on your mass storage device is a good thing to do, so using BSS sections to reduce your executable file sizes is a useful optimization.

The one thing that many people tend to forget, however, is that a BSS section still requires main memory at runtime. Even though the executable file size may be smaller, each byte of data you declare in your program translates to 1 byte of data in memory. Some programmers have the mistaken impression that the executable's file size is indicative of the amount of memory that the program consumes. This, however, isn't necessarily true, as our BSS example shows. A given application's executable file might consist of only 600 bytes, but if that program uses four different sections, with each section consuming a 4KB page in memory, the program will require 16,384 bytes of memory when the OS loads it into memory. This is because the underlying memory protection hardware requires the OS to allocate whole pages of memory to a given process.

4.7.2 Internal Fragmentation

Another reason an executable file might be smaller than an application's *execution memory footprint* (the amount of memory the application consumes at runtime) is *internal fragmentation*. Internal fragmentation occurs when you must allocate sections of memory in fixed-sized chunks even though you might need only a portion of each chunk (see Figure 4-11).

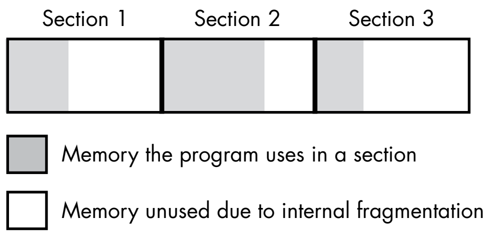

Section 1
Section 2
Section 3
Memory the program uses in a section
Memory unused due to internal fragmentation

Figure 4-11: Internal fragmentation

Remember that each section in memory consumes an integral number of pages, even if that section's data size is not a multiple of the page size. All bytes from the last data/code byte in a section to the end of the page holding that byte are wasted; this is internal fragmentation. Some executable file formats allow you to pack each section without padding it to some multiple of the page size. However, as you'll soon see, there may be a performance penalty for packing sections together in this fashion, so some executable formats don't do it.

Finally, don't forget that an executable file's size does not include any data (including data objects on the heap and values placed on the CPU's stack) allocated dynamically at runtime. As you can see, an application can actually consume much more memory than the executable file's size.

Programmers commonly compete to see who can write the smallest "Hello World" program using their favorite language. Assembly language programmers are especially guilty of bragging about how much smaller they can write this program in assembly than they can in C or some other HLL. This is a fun mental challenge, but whether the program's executable file is 600 or 16,000 bytes long, the chances are pretty good that the program will consume exactly the same amount of memory at runtime once the operating system allocates four or five pages for the program's different sections. While writing the world's shortest "Hello World" application might win bragging rights, in real-world terms such an application saves almost nothing at runtime due to internal fragmentation.

4.7.3 Reasons to Optimize for Space

This is not to suggest that optimizing for space isn't worthwhile. Programmers who write great code consider all the machine resources their application uses, and they avoid wasting those resources. However, attempting to take this process to an extreme is a waste of effort. Once you've gotten a given section below 4,096 bytes (on an 80x86 or other CPU with a 4KB page size), additional optimizations save you nothing. Remember, the *allocation granularity*—that is, the minimum allocation block size—is 4,096 bytes. If you have a section with 4,097 bytes of data, it's going to consume 8,192 bytes at runtime. In that case, it would behoove you to reduce that section by 1 byte (thereby saving 4,096 bytes at runtime). However, if you have a data section that consumes 16,380 bytes, attempting to reduce its size by 4,092 bytes in order to reduce the file size is going to be difficult unless the data organization was very bad to begin with.

Note that most operating systems allocate disk space in clusters (or blocks) that are often comparable to (or even larger than) the page size for the memory management unit in the CPU. Therefore, if you shrink an executable's file size down to 700 bytes in an attempt to save disk space (an admirable goal, even given the gargantuan size of modern disk drive subsystems), the savings won't be as great as you'd expect. That 700-byte application, for example, is still going to consume a minimum of one block on the disk's surface. All you achieve by reducing your application's code or data size is to waste that much more space in the disk file—subject, of course, to section/block allocation granularity.

For larger executable files, those larger than the disk block size, internal fragmentation has less impact with respect to wasted space. If an executable file packs the data and code sections without any wasted space between the sections, then internal fragmentation occurs only at the end of the file, in the very last disk block. Assuming that file sizes are random (even distribution), then internal fragmentation will waste approximately one-half of a disk block per file (that is, an average of 2KB per file when the disk block size is 4KB). For a very small file, one that is less than 4KB in size, this might represent a significant amount of the file's space. For larger applications, however, the wasted space becomes insignificant. So it would seem that as long as an executable file packs all the sections of the program sequentially in the file, the file will be as small as possible. But is this really desirable?

Assuming all things are equal, having smaller executable files is a good thing. However, all things aren't always equal, so sometimes creating the smallest possible executable file isn't really best. To understand why, recall the earlier discussion of the operating system's virtual memory subsystem. When an OS loads an application into memory for execution, it doesn't actually have to read the entire file. Instead, the operating system's paging system can load only those pages needed to start the application. This usually consists of the first page of executable code, a page of memory to hold stack-based data, and, possibly, some data pages. In theory, an application could begin execution with as few as two or three pages of memory and bring in the remaining pages of code and data *on demand* (as the application requests the data or code found in those pages). This is known as *demand-paged memory management*. In practice, most operating systems actually preload pages for efficiency reasons (maintaining a working set of pages in memory). However, operating systems generally don't load the entire executable file into memory; instead, they load various blocks as the application requires them. As a result, the effort needed to load a page of memory from a file can dramatically affect a program's performance. Is there some way, then, to organize the executable file to improve performance when the OS uses demand-paged memory management? Yes—if you make the file a little larger.

The trick to improving performance is to organize the executable file's blocks to match the memory page layout. This means that sections in memory should be aligned on page-sized boundaries in the executable file. It also means that disk blocks should be the size of, or a multiple of the size of, a disk sector or block. This being the case, the virtual memory management system can rapidly copy a single block on the disk into a single page of memory, update any necessary relocation values, and continue program execution. On the other hand, if a page of data is spread across two blocks on the disk and is not aligned on a disk block boundary, the OS has to read two blocks (rather than one) from disk into an internal buffer and then copy the page of data from that buffer to the destination page where it belongs. This extra work can be very time-consuming and hamper application performance.

For this reason, some compilers will actually pad the executable file to ensure that each section in the executable file begins on a block boundary that the virtual memory management subsystem can map directly to a page in memory. Compilers that employ this technique often produce much larger executable file sizes than those that don't. This is especially true if the executable file contains a large amount of BSS (uninitialized) data that a packed file format can represent very compactly.

Because some compilers produce packed files at the expense of execution time, while others produce expanded files that load and run faster, it's dangerous to compare the quality of compilers based on the size of the executable files they produce. The best way to determine the quality of a compiler's output is by directly analyzing that output, not by using a weak metric such as output file size.

NOTE *Analyzing compiler output is the subject of the very next chapter, so if you're interested in the topic, keep reading.*

4.8 Data and Code Alignment in an Object File

As I pointed out in *WGC1*, aligning data objects on an address boundary that is "natural" for that object's size can improve performance. It's also true that aligning the start of a procedure's code or the starting instruction of a loop on some nice boundary can improve performance. Compiler writers are well aware of this fact and will often emit *padding bytes* in the data or code stream to align data or code sequences on an appropriate boundary. However, note that the linker is free to move sections of code around when linking two object files to produce a single executable result.

Sections are generally aligned to a page boundary in memory. For a typical application, the text/code section will begin on a page boundary, the data section will begin on a different page boundary, the BSS section (if it exists) will begin on its own page boundary, and so on. However, this doesn't imply that each and every section associated with a section header in the object files starts on its own page in memory. The linker program will combine sections that have the same name into a single section in the executable file. So, for example, if two different object files both contain a `.text` segment, the linker will combine them into a single `.text` section in the final executable file. By combining sections that have the same name, the linker avoids wasting a large amount of memory to internal fragmentation.

How does the linker respect the alignment requirements of each of the sections it combines? The answer, of course, depends on exactly what object file format and OS you're using, but it's usually found in the object file format itself. For example, in a Windows PE/COFF file the `IMAGE_OPTIONAL_HEADER32` structure contains a field named `SectionAlignment`. This field specifies the address boundary that the linker and OS must respect when combining sections and loading the section into memory. Under Windows, the `SectionAlignment` field in the PE/COFF optional header will usually contain 32 or 4,096 bytes. The 4KB value, of course, will align a section to a 4KB page boundary in memory. The alignment value of 32 was probably chosen because this is a reasonable cache line value (see *WGC1* for a discussion of cache lines). Other values are certainly possible—an application programmer can usually specify section alignment values by using linker (or compiler) command-line parameters.

4.8.1 Choosing a Section Alignment Size

Within each section, a compiler, assembler, or other code-generation tool can guarantee any alignment that is a submultiple of the section's alignment. For example, if the section's alignment value is 32, then alignments of 1, 2, 4, 8, 16, and 32 are possible within that section. Larger alignment values are not possible. If a section's alignment value is 32 bytes, you cannot guarantee alignment within that section on a 64-byte boundary, because the OS or linker will respect only the section's alignment value and it can place that section on any boundary that is a multiple of 32 bytes. And about half of those won't be 64-byte boundaries.

Perhaps less obvious, but just as true, is the fact that you cannot align an object within a section on a boundary that is not a submultiple of the section's alignment. For example, a section with a 32-byte alignment value will not allow an alignment of 5 bytes. True, you could guarantee that the offset of some object within the section would be a multiple of 5; however, if the starting memory address of the section is not a multiple of 5, then the address of the object you attempted to align might not fall on a multiple of 5 bytes. The only solution is to pick a section alignment value that is some multiple of 5.

Because memory addresses are binary values, most language translators and linkers limit alignment values to a power of 2 that is less than or equal to some maximum value, usually the memory management unit's page size. Many languages restrict the alignment value to a small power of 2 (such as 32, 64, or 256).

4.8.2 Combining Sections

When a linker combines two sections, it has to respect the alignment values associated with each section because the application may depend on that alignment for correct operation. Therefore, a linker or other program that combines sections in object files can't simply concatenate the data for the two sections when building the combined section.

When combining two sections, a linker might have to add padding bytes between the sections if one or both of the lengths is not a multiple of the sections' alignment. For example, if two sections have an alignment value of 32, and one section is 37 bytes long and the other section is 50 bytes long, the linker will have to add 27 bytes of padding between the first and second sections, or it will have to add 14 bytes of padding between the second section and the first (the linker usually gets to choose in which order it places the sections in the combined file).

The situation gets a bit more complicated if the alignment values are not the same for the two sections. When a linker combines two sections, it has to ensure that the alignment requests are met for the data in both sections. If the alignment value of one section is a multiple of the other section's alignment value, then the linker can simply choose the larger of the two alignment values. For example, if the alignment values are always powers of 2 (as most linkers require), then the linker can simply choose the larger of the two alignment values for the combined section.

If one section's alignment value is not a multiple of the other's, then the only way to guarantee the alignment requirements of both sections when combining them is to use an alignment value that is a product of the two values (or, better yet, the *least common multiple* of the two values). For example, combining a section aligned on a 32-byte boundary with one aligned on a 5-byte boundary requires an alignment value of 160 bytes (5 × 32). Because of the complexities of combining two such sections, most linkers require section sizes to be small powers of 2, which guarantees that the larger segment alignment value is always a multiple of the smaller alignment value.

4.8.3 Controlling the Section Alignment

You typically use linker options to control the section alignment within your programs. For example, with the Microsoft *link.exe* program, the `/ALIGN:value` command-line parameter tells the linker to align all sections in the output file to the specified boundary (which must be a power of 2). GNU's *ld* linker program lets you specify a section alignment by using the `BLOCK(value)` option in a linker script file. The macOS linker (`ld`) provides a `-segalign value` command-line option you can use to specify section alignment. The exact command and possible values are specific to the linker;

however, almost every modern linker allows you to specify the section alignment properties. See your linker's documentation for details.

One word of caution about setting the section alignment: more often than not, a linker will require that all sections in a given file be aligned on the same boundary (a power of 2). Therefore, if you have different alignment requirements for all your sections, then you'll need to choose the largest alignment value for all the sections in your object file.

4.8.4 Aligning Sections Within Library Modules

Section alignment can have a very big impact on the size of your executable files if you use a lot of short library routines. Suppose, for example, that you've specified an alignment size of 16 bytes for the sections associated with the object files appearing in a library. Each library function that the linker processes will be placed on a 16-byte boundary. If the functions are small (fewer than 16 bytes in length), the space between the functions will be unused when the linker creates the final executable. This is another form of internal fragmentation.

To understand why you would want to align the code (or data) in a section on a given boundary, think about how cache lines work (see *WGC1* for a refresher). By aligning the start of a function on a cache line, you may be able to slightly increase the execution speed of that function, as it may generate fewer cache misses during execution. For this reason, many programmers like to align all their functions at the start of a cache line. Although the size of a cache line varies from CPU to CPU, a typical cache line is 16 to 64 bytes long, so many compilers, assemblers, and linkers will attempt to align code and data to one of these boundaries. On the 80x86 processor, there are some other benefits to 16-byte alignment, so many 80x86-based tools default to a 16-byte section alignment for object files.

Consider, for example, the following short HLA (High-Level Assembly) program, processed by Microsoft tools, that calls two relatively small library routines:

```
program t;
#include( "bits.hhf" )

begin t;

        bits.cnt( 5 );
        bits.reverse32( 10 );

end t;

Here is the source code to the bits.cnt library module:

unit bitsUnit;

#includeonce( "bits.hhf" );


    // bitCount-
```

```
//
//  Counts the number of "1" bits in a dword value.
//  This function returns the dword count value in EAX.

procedure bits.cnt( BitsToCnt:dword ); @nodisplay;

const
    EveryOtherBit       := $5555_5555;
    EveryAlternatePair  := $3333_3333;
    EvenNibbles         := $0f0f_0f0f;

begin cnt;

    push( edx );
    mov( BitsToCnt, eax );
    mov( eax, edx );

    // Compute sum of each pair of bits
    // in EAX. The algorithm treats
    // each pair of bits in EAX as a
    // 2-bit number and calculates the
    // number of bits as follows (description
    // is for bits 0 and 1, but it generalizes
    // to each pair):
    //
    //  EDX =   BIT1  BIT0
    //  EAX =      0  BIT1
    //
    //  EDX-EAX =   00 if both bits were 0.
    //              01 if Bit0 = 1 and Bit1 = 0.
    //              01 if Bit0 = 0 and Bit1 = 1.
    //              10 if Bit0 = 1 and Bit1 = 1.
    //
    // Note that the result is left in EDX.

    shr( 1, eax );
    and( EveryOtherBit, eax );
    sub( eax, edx );

    // Now sum up the groups of 2 bits to
    // produces sums of 4 bits. This works
    // as follows:
    //
    //  EDX = bits 2,3, 6,7, 10,11, 14,15, ..., 30,31
    //        in bit positions 0,1, 4,5, ..., 28,29 with
    //        0s in the other positions.
    //
    //  EAX = bits 0,1, 4,5, 8,9, ... 28,29 with 0s
    //        in the other positions.
    //
    //  EDX + EAX produces the sums of these pairs of bits.
    //  The sums consume bits 0,1,2, 4,5,6, 8,9,10, ...
    //                                             28,29,30
    //  in EAX with the remaining bits all containing 0.
```

```
        mov( edx, eax );
        shr( 2, edx );
        and( EveryAlternatePair, eax );
        and( EveryAlternatePair, edx );
        add( edx, eax );

        // Now compute the sums of the even and odd nibbles in
        // the number. Since bits 3, 7, 11, etc. in EAX all
        // contain 0 from the above calculation, we don't need
        // to AND anything first, just shift and add the two
        // values.
        // This computes the sum of the bits in the 4 bytes
        // as four separate values in EAX (AL contains number of
        // bits in original AL, AH contains number of bits in
        // original AH, etc.)

        mov( eax, edx );
        shr( 4, eax );
        add( edx, eax );
        and( EvenNibbles, eax );

        // Now for the tricky part.
        // We want to compute the sum of the 4 bytes
        // and return the result in EAX. The following
        // multiplication achieves this. It works
        // as follows:
        //  (1) the $01 component leaves bits 24..31
        //      in bits 24..31.
        //
        //  (2) the $100 component adds bits 17..23
        //      into bits 24..31.
        //
        //  (3) the $1_0000 component adds bits 8..15
        //      into bits 24..31.
        //
        //  (4) the $1000_0000 component adds bits 0..7
        //      into bits 24..31.
        //
        //  Bits 0..23 are filled with garbage, but bits
        //  24..31 contain the actual sum of the bits
        //  in EAX's original value. The SHR instruction
        //  moves this value into bits 0..7 and zeros
        //  out the HO bits of EAX.

        intmul( $0101_0101, eax );
        shr( 24, eax );

        pop( edx );

    end cnt;

end bitsUnit;
```

Here is the source code for the `bits.reverse32()` library function. Note that this source file also includes the `bits.reverse16()` and `bits.reverse8()` functions (to conserve space, the bodies of these functions do not appear below). Although their operation is not pertinent to our discussion, note that these functions swap the values in the HO (high-order) and LO (low-order) bit positions. Because these three functions appear in a single source file, any program that includes one of these functions will automatically include all three (because of the way compilers, assemblers, and linkers work).

```
unit bitsUnit;

#include( "bits.hhf" );


    procedure bits.reverse32( BitsToReverse:dword ); @nodisplay; @noframe;
    begin reverse32;

        push( ebx );
        mov( [esp+8], eax );

        // Swap the bytes in the numbers:

        bswap( eax );

        // Swap the nibbles in the numbers

        mov( $f0f0_f0f0, ebx );
        and( eax, ebx );
        and( $0f0f_0f0f, eax );
        shr( 4, ebx );
        shl( 4, eax );
        or( ebx, eax );

        // Swap each pair of 2 bits in the numbers:

        mov( eax, ebx );
        shr( 2, eax );
        shl( 2, ebx );
        and( $3333_3333, eax );
        and( $cccc_cccc, ebx );
        or( ebx, eax );

        // Swap every other bit in the number:

        lea( ebx, [eax + eax] );
        shr( 1, eax );
        and( $5555_5555, eax );
        and( $aaaa_aaaa, ebx );
        or( ebx, eax );
        pop( ebx );
        ret( 4 );
    end reverse32;


    procedure bits.reverse16( BitsToReverse:word );
        @nodisplay; @noframe;
    begin reverse16;

        // Uninteresting code that is very similar to
        // that appearing in reverse32 has been snipped...

    end reverse16;



    procedure bits.reverse8( BitsToReverse:byte );
        @nodisplay; @noframe;
    begin reverse8;

        // Uninteresting code snipped...

    end reverse8;


end bitsUnit;
```

The Microsoft *dumpbin.exe* tool allows you to examine the various fields of an *.obj* or *.exe* file. Running `dumpbin` with the `/headers` command-line option on the *bitcnt.obj* and *reverse.obj* files (produced for the HLA standard library) tells us that each of the sections is aligned to a 16-byte boundary. Therefore, when the linker combines the *bitcnt.obj* and *reverse.obj* data with the sample program given earlier, it will align the `bits.cnt()` function in the *bitcnt.obj* file on a 16-byte boundary, and the three functions in the *reverse.obj* file on a 16-byte boundary. (Note that it will not align each function in the file on a 16-byte boundary. That task is the responsibility of the tool that created the object file, if such alignment is desired.) By using the *dumpbin.exe* program with the `/disasm` command-line option on the executable file, you can see that the linker has honored these alignment requests (note that an address that is aligned on a 16-byte boundary will have a `0` in the LO hexadecimal digit):

```
  Address   opcodes            Assembly Instructions
  --------- ------------------ ----------------------------
  04001000: E9 EB 00 00 00     jmp         040010F0
  04001005: E9 57 01 00 00     jmp         04001161
  0400100A: E8 F1 00 00 00     call        04001100

; Here's where the main program starts.

  0400100F: 6A 00              push        0
  04001011: 8B EC              mov         ebp,esp
  04001013: 55                 push        ebp
  04001014: 6A 05              push        5
  04001016: E8 65 01 00 00     call        04001180
  0400101B: 6A 0A              push        0Ah
  0400101D: E8 0E 00 00 00     call        04001030
  04001022: 6A 00              push        0
  04001024: FF 15 00 20 00 04  call        dword ptr ds:[04002000h]

;The following INT3 instructions are used as padding in order
;to align the bits.reverse32 function (which immediately follows)
;to a 16-byte boundary:

  0400102A: CC                 int         3
  0400102B: CC                 int         3
  0400102C: CC                 int         3
  0400102D: CC                 int         3
  0400102E: CC                 int         3
  0400102F: CC                 int         3

; Here's where bits.reverse32 starts. Note that this address
; is rounded up to a 16-byte boundary.

  04001030: 53                 push        ebx
  04001031: 8B 44 24 08        mov         eax,dword ptr [esp+8]
  04001035: 0F C8              bswap       eax
  04001037: BB F0 F0 F0 F0     mov         ebx,0F0F0F0F0h
  0400103C: 23 D8              and         ebx,eax
  0400103E: 25 0F 0F 0F 0F     and         eax,0F0F0F0Fh
  04001043: C1 EB 04           shr         ebx,4
  04001046: C1 E0 04           shl         eax,4
  04001049: 0B C3              or          eax,ebx
  0400104B: 8B D8              mov         ebx,eax
  0400104D: C1 E8 02           shr         eax,2
  04001050: C1 E3 02           shl         ebx,2
  04001053: 25 33 33 33 33     and         eax,33333333h
  04001058: 81 E3 CC CC CC CC  and         ebx,0CCCCCCCCh
  0400105E: 0B C3              or          eax,ebx
  04001060: 8D 1C 00           lea         ebx,[eax+eax]
  04001063: D1 E8              shr         eax,1
  04001065: 25 55 55 55 55     and         eax,55555555h
  0400106A: 81 E3 AA AA AA AA  and         ebx,0AAAAAAAAh
  04001070: 0B C3              or          eax,ebx
  04001072: 5B                 pop         ebx
  04001073: C2 04 00           ret         4

; Here's where bits.reverse16 begins. As this function appeared
; in the same file as bits.reverse32, and no alignment option
; was specified in the source file, HLA and the linker won't
; bother aligning this to any particular boundary. Instead, the
; code immediately follows the bits.reverse32 function
; in memory.

  04001076: 53                 push        ebx
  04001077: 50                 push        eax
  04001078: 8B 44 24 0C        mov         eax,dword ptr [esp+0Ch]

        .
        .    ; uninteresting code for bits.reverse16 and
        .    ; bits.reverse8 was snipped

; end of bits.reverse8 code

  040010E6: 88 04 24          mov        byte ptr [esp],al
  040010E9: 58                pop        eax
  040010EA: C2 04 00          ret        4

; More padding bytes to align the following function (used by
; HLA exception handling) to a 16-byte boundary:

  040010ED: CC                int        3
  040010EE: CC                int        3
  040010EF: CC                int        3

; Default exception return function (automatically generated
; by HLA):

  040010F0: B8 01 00 00 00    mov        eax,1
  040010F5: C3                ret

; More padding bytes to align the internal HLA BuildExcepts
; function to a 16-byte boundary:

  040010F6: CC                int        3
  040010F7: CC                int        3
  040010F8: CC                int        3
  040010F9: CC                int        3
  040010FA: CC                int        3
  040010FB: CC                int        3
  040010FC: CC                int        3
  040010FD: CC                int        3
  040010FE: CC                int        3
  040010FF: CC                int        3

; HLA BuildExcepts code (automatically generated by the
; compiler):

  04001100: 58                pop        eax
  04001101: 68 05 10 00 04    push       4001005h
  04001106: 55                push       ebp

        .
        .    ; Remainder of BuildExcepts code goes here
        .    ; along with some other code and data
        .

; Padding bytes to ensure that bits.cnt is aligned
; on a 16-byte boundary:

  0400117D: CC                int        3
  0400117E: CC                int        3
  0400117F: CC                 int        3

; Here's the low-level machine code for the bits.cnt function:

  04001180: 55                 push       ebp
  04001181: 8B EC              mov        ebp,esp
  04001183: 83 E4 FC           and        esp,0FFFFFFFCh
  04001186: 52                 push       edx
  04001187: 8B 45 08           mov        eax,dword ptr [ebp+8]
  0400118A: 8B D0              mov        edx,eax
  0400118C: D1 E8              shr        eax,1
  0400118E: 25 55 55 55 55     and        eax,55555555h
  04001193: 2B D0              sub        edx,eax
  04001195: 8B C2              mov        eax,edx
  04001197: C1 EA 02           shr        edx,2
  0400119A: 25 33 33 33 33     and        eax,33333333h
  0400119F: 81 E2 33 33 33 33  and        edx,33333333h
  040011A5: 03 C2              add        eax,edx
  040011A7: 8B D0              mov        edx,eax
  040011A9: C1 E8 04           shr        eax,4
  040011AC: 03 C2              add        eax,edx
  040011AE: 25 0F 0F 0F 0F     and        eax,0F0F0F0Fh
  040011B3: 69 C0 01 01 01 01  imul       eax,eax,1010101h
  040011B9: C1 E8 18           shr        eax,18h
  040011BC: 5A                 pop        edx
  040011BD: 8B E5              mov        esp,ebp
  040011BF: 5D                 pop        ebp
  040011C0: C2 04 00           ret        4
```

The exact operation of this program really isn't important (after all, it doesn't actually do anything useful). The takeaway is how the linker inserts extra bytes (`$cc`, the `int 3` instruction) before a group of one or more functions appearing in a source file to ensure that they are aligned on the specified boundary.

In this particular example, the `bits.cnt()` function is actually 64 bytes long, and the linker inserted only 3 bytes in order to align it to a 16-byte boundary. This percentage of waste—the number of padding bytes compared to the size of the function—is quite low. However, if you have a large number of small functions, the wasted space can become significant (as with the default exception handler in this example that has only two instructions). When creating your own library modules, you'll need to weigh the inefficiencies of extra space for padding against the small performance gains you'll obtain by using aligned code.

Object code dump utilities (like *dumpbin.exe*) are quite useful for analyzing object code and executable files in order to determine attributes such as section size and alignment. Linux (and most Unix-like systems) provides the comparable `objdump` utility. I'll discuss these tools in the next chapter, as they are great for analyzing compiler output.

4.9 How Linkers Affect Code

The limitations of object file formats such as COFF and ELF have a big impact on the quality of code that compilers can generate. Because of how object file formats are designed, linkers and compilers often have to inject extra code into an executable file that wouldn't be necessary otherwise. In this section we'll explore some of the problems that generic object code formats like COFF and ELF inflict on the executable code.

One problem with generic object file formats like COFF and ELF is that they were not designed to produce efficient executable files for specific CPUs. Instead, they were created to support a wide variety of CPUs and to make it easy to link together object modules. Unfortunately, their versatility often prevents them from creating the best possible object files.

Perhaps the biggest problem with the COFF and ELF formats is that relocation values in the object file must apply to 32- and 64-bit pointers in the object code. This creates problems, for example, when an instruction encodes a displacement or address value with less than 32 (64) bits. On some processors, such as the 80x86, displacements smaller than 32 bits are so small (for example, the 80x86's 8-bit displacement) that you would never use them to refer to code outside the current object module. However, on some RISC processors, such as the PowerPC or ARM, displacements are much larger (26 bits in the case of the PowerPC branch instruction). This can lead to code kludges like the function stub generation that GCC produces for external function calls. Consider the following C program and the PowerPC code that GCC emits for it:

```
#include <stdio.h>
int main( int argc )
{
      .
      .
      .
    printf
    (
        "%d %d %d %d %d ",
        .
        .
        .
    );
    return( 0 );
}

; PowerPC assembly output from GCC:

            .
            .
            .
        ;The following sets up the
        ; call to printf and calls printf:

        addis r3,r31,ha16(LC0-L1$pb)
        la r3,lo16(LC0-L1$pb)(r3)
        lwz r4,64(r30)
        lwz r5,80(r30)
        lwz r6,1104(r30)
        lwz r7,1120(r30)
        lis r0,0x400
        ori r0,r0,1120
        lwzx r8,r30,r0
        bl L_printf$stub ; Call to printf "stub" routine.

        ;Return from main program:

        li r0,0
        mr r3,r0
        lwz r1,0(r1)
        lwz r0,8(r1)
        mtlr r0
        lmw r30,-8(r1)
        blr

; Stub, to call the external printf function.
; This code does an indirect jump to the printf
; function using the 32-bit L_printf$lazy_ptr
; pointer that the linker can modify.

        .data
        .picsymbol_stub
L_printf$stub:
        .indirect_symbol _printf
        mflr r0
        bcl 20,31,L0$_printf
L0$_printf:
        mflr r11
        addis r11,r11,ha16(L_printf$lazy_ptr-L0$_printf)
        mtlr r0
        lwz r12,lo16(L_printf$lazy_ptr-L0$_printf)(r11)
        mtctr r12
        addi r11,r11,lo16(L_printf$lazy_ptr-L0$_printf)
        bctr
.data
.lazy_symbol_pointer
L_printf$lazy_ptr:
        .indirect_symbol _printf

; The following is where the compiler places a 32-bit
; pointer that the linker can fill in with the address
; of the actual printf function:

        .long dyld_stub_binding_helper
```

The compiler must generate the `L_printf$stub` stub because it doesn't know how far away the actual `printf()` routine will be when the linker adds it to the final executable file. It's unlikely that `printf()` would be sitting outside the ±32MB range that the PowerPC's 24-bit branch displacement supports (extended to 26 bits), but it's not guaranteed. If `printf()` is part of a shared library that is dynamically linked in at runtime, it very well could be outside this range. Therefore, the compiler has to make the safe choice and use a 32-bit displacement for the address of the `printf()` function. Unfortunately, PowerPC instructions don't support a 32-bit displacement, because all PowerPC instructions are 32 bits long. A 32-bit displacement would leave no room for the instruction's opcode. Therefore, the compiler has to store a 32-bit pointer to the `printf()` routine in a variable and jump indirect through that variable. Accessing a 32-bit memory pointer on the PowerPC takes quite a bit of code if you don't already have the pointer's address in a register, hence all the extra code following the `L_printf$stub` label.

If the linker were able to adjust 26-bit displacements rather than just 32-bit values, there would be no need for the `L_printf$stub` routine or the `L_printf$lazy_ptr` pointer variable. Instead, the `bl L_printf$stub` instruction would be able to branch directly to the `printf()` routine (assuming it's not more than ±32MB away). Because single program files generally don't contain more than 32MB of machine instructions, there would rarely be the need to go through the gymnastics this code does in order to call an external routine.

Unfortunately, there is nothing you can do about the object file format; you're stuck with whatever format the OS specifies (which is usually a variant of COFF or ELF on modern 32-bit and 64-bit machines). However, you can work within those limitations.

If you expect your code to run on a CPU like the PowerPC or ARM (or some other RISC processor) that cannot encode 32-bit displacements directly within instructions, you can optimize by avoiding cross-module calls as much as possible. While it's not good programming practice to create monolithic applications, where all the source code appears in one source file (or is processed by a single compilation), there's really no need to place all of your own functions in separate source modules and compile each one separately from the others—particularly if these routines make calls to one another. By placing a set of common routines your code uses into a single compilation unit (source file), you allow the compiler to optimize the calls among these functions and avoid all the stub generation on processors like the PowerPC. This is not a suggestion to simply move all of your external functions into a single source file. The code is better only if the functions in a module call one another or share other global objects. If the functions are completely independent of one another and are called only by code external to the compilation unit, then you've saved nothing because the compiler may still need to generate stub routines in the external code.

4.10 For More Information

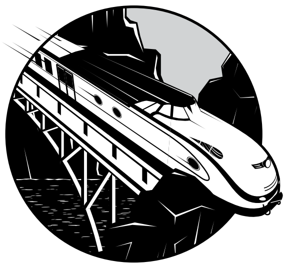
Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* 2nd ed. Essex, UK: Pearson Education Limited, 1986.

Gircys, Gintaras. *Understanding and Using COFF.* Sebastopol, CA: O'Reilly Media, 1988.

Levine, John R. *Linkers and Loaders.* San Diego: Academic Press, 2000.

5

TOOLS FOR ANALYZING COMPILER OUTPUT



In order to write great code, you must be able to recognize the difference between programming language sequences that do their job more or less adequately and those that are great. In the context of our discussion, great code sequences use fewer instructions, fewer machine cycles, or less memory than mediocre code sequences. If you're working in assembly language, the CPU manufacturers' data sheets and a bit of experimentation are all it takes to determine which code sequences are great and which are not. When working with HLLs, however, you need some way of mapping the high-level language statements in a program to the corresponding machine code, so that you can determine the quality of those HLL statements. In this chapter, we'll discuss how to:

- View and analyze a compiler's machine language output so you can use that information to write better HLL code
- Tell certain compilers to produce a human-readable assembly language output file

- Analyze binary object output files using tools such as `dumpbin` and `objdump`
- Use a disassembler to examine the machine code output a compiler produces
- Use a debugger to analyze compiler output
- Compare two different assembly language listings for the same HLL source file to determine which version is better

Analyzing compiler output is one of the principal skills you'll need in order to distinguish great machine code from merely sufficient machine code. To analyze compiler output, you'll need to learn a couple of things. First, you'll need to learn enough assembly language programming so that you can effectively read compiler output.[1] Second, you'll need to learn how to tell a compiler (or some other tool) to produce human-readable assembly language output. Finally, you'll have to learn how to correlate the assembly instructions with the HLL code. Chapters 3 and 4 gave you the foundation you need to read some basic assembly code. This chapter discusses how to translate compiler output into a human-readable form. And the rest of this book deals with analyzing that assembly code so you can generate better machine code by wisely choosing your HLL statements.

Let's begin with some background on compiler output and things to keep in mind for optimization.

5.1 Background

As Chapter 4 discussed, most compilers available today emit object code output that a linker program reads and processes in order to produce an executable program. Because the object code file generally consists of binary data that is not human-readable, many compilers also provide an option to produce an assembly language version of the output code. By activating this option, you can analyze the compiler's output and, if necessary, refine your HLL source code accordingly. Indeed, with a specific compiler and a thorough knowledge of its optimizations, you can write HLL source code that compiles to machine code that's almost as good as the best handwritten assembly language code. Although you can't expect such optimizations to work with every compiler, this trick enables you to write good code with one compiler that will still be able to run (though possibly less efficiently) on other processors. This is an excellent solution for code that needs to run as efficiently as possible on a certain class of machines but still needs to run on other CPUs.

NOTE *Keep in mind that examining compiler output may lead you to implement nonportable optimizations. That is, when you examine your compiler's output you might decide to modify your HLL source code to produce better output; however, those optimizations might not carry over to a different compiler.*

1. This is perhaps the most practical reason for the typical programmer to learn assembly.

The ability to emit assembly language output is compiler specific. Some compilers do so by default. GCC, for example, always emits an assembly language file (though it typically deletes that file after the compilation). Most compilers, however, must be explicitly told to produce an assembly language listing. Some compilers produce an assembly listing that can be run through an assembler to produce object code. Some compilers may only produce assembly annotation in a listing file, and that "assembly code" is not syntax-compatible with any existing assembler. For your purposes, it doesn't matter if a real-world assembler is capable of processing the compiler's assembly output; you're only going to read that output to determine how to tweak the HLL code to produce better object code.

For those compilers that can produce assembly language output, the readability of the assembly code varies considerably. Some compilers insert the original HLL source code into the assembly output as comments, which makes it easy to correlate the assembly instructions with the HLL code. Other compilers (such as GCC) emit pure assembly language code; so, unless you're well versed in the particular CPU's assembly language, analyzing the output can be difficult.

Another problem that may affect the readability of the compiler output is the optimization level you choose. If you disable all optimizations, it is often easier to determine which assembly instructions correspond to the HLL statements. Unfortunately, with the optimizations turned off, most compilers generate low-quality code. If the purpose of viewing assembly output from a compiler is to choose better HLL sequences, then you must specify the same optimization level that you will use for the production version of your application. You should never tweak your high-level code to produce better assembly code at one optimization level and then change the optimization level for your production code. If you do, you may wind up doing extra work that the optimizer would normally do for you. Worse, those manual optimizations could actually prevent the compiler from doing a decent job when you increase its optimization level.

When you specify a higher level of optimization for a compiler, the compiler will often move code around in the assembly output file, eliminate code entirely, and do other code transformations that obfuscate the correspondence between the high-level code and the assembly output. Still, with a bit of practice, you can determine which machine instructions correspond to a given statement in the HLL code.

5.2 Telling a Compiler to Produce Assembly Output

How you tell a compiler to emit an assembly language output file is specific to the compiler. For that information, you'll need to consult the documentation for your particular compiler. This section will look at two commonly used C/C++ compilers: GCC and Microsoft's Visual C++.

5.2.1 Assembly Output from GNU Compilers

To emit assembly output with the GCC compiler, you specify the `-S` option on the command line when invoking the compiler. Here is a sample command line for GCC:

```
gcc -O2 -S t1.c     # -O2 option is for optimization
```

When supplied to GCC, the `-S` option doesn't actually tell the compiler to produce an assembly output file. GCC *always* produces an assembly output file. The `-S` simply tells GCC to stop all processing after it has produced an assembly file. GCC will produce an assembly output file whose root name is the same as the original C file (*t1* in these examples) with a *.s* suffix (during normal compilation, GCC deletes the *.s* file after assembling it).

5.2.2 Assembly Output from Visual C++

The Visual C++ compiler (VC++) uses the `-FA` command-line option to specify MASM-compatible assembly language output. The following is a typical command line to VC++ to tell it to produce an assembly listing:

```
cl -O2 -FA t1.c
```

5.2.3 Example Assembly Language Output

As an example of producing assembly language output from a compiler, consider the following C program:

```
#include <stdio.h>
int main( int argc, char **argv )
{
    int i;
    int j;

    i = argc;
    j = **argv;


    if( i == 2 )
    {
        ++j;
    }
    else
    {
        --j;
    }

    printf( "i=%d, j=%d\n", i, j );
    return 0;
}
```

The following subsections provide the compiler output for Visual C++ and GCC from this code sequence in order to highlight the differences between their respective assembly language listings.

5.2.3.1 Visual C++ Assembly Language Output

Compiling this file with VC++ using the command line

```
cl -Fa -O1 t1.c
```

produces the following (MASM) assembly language output.

NOTE *The exact meaning of each assembly language statement appearing in this output isn't important—yet! What's important is seeing the difference between the syntax in this listing and the listings for Visual C++ and Gas that appear in the following sections.*

```
; Listing generated by Microsoft (R) Optimizing
; Compiler Version 19.00.24234.1
; This listing is manually annotated for readability.

include listing.inc

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

PUBLIC  __local_stdio_printf_options
PUBLIC  _vfprintf_l
PUBLIC  printf
PUBLIC  main
PUBLIC  ??_C@_0M@MJLDLLNK@i?$DN?$CFd?0?5j?$DN?$CFd?6?$AA@ ; `string'
EXTRN   __acrt_iob_func:PROC
EXTRN   __stdio_common_vfprintf:PROC
_DATA   SEGMENT
COMM    ?_OptionsStorage@?1??__local_stdio_printf_options@@9@9:QWORD
; `__local_stdio_printf_options'::`2'::_OptionsStorage
_DATA   ENDS
;       COMDAT pdata
pdata   SEGMENT
    .
    .
    .
;       COMDAT main
_TEXT   SEGMENT
argc$ = 48
argv$ = 56
main    PROC                                        ; COMDAT

$LN6:
        sub     rsp, 40                             ; 00000028H

; if( i == 2 )
;{
;   ++j;
;}
;else
;{
;    --j
;}

        mov     rax, QWORD PTR [rdx]   ; rax (i) = *argc
        cmp     ecx, 2
        movsx   edx, BYTE PTR [rax]    ; rdx(j) = **argv

        lea     eax, DWORD PTR [rdx-1] ; rax = ++j
        lea     r8d, DWORD PTR [rdx+1] ; r8d = --j;

        mov     edx, ecx               ; edx = argc (argc was passed in rcx)
        cmovne  r8d, eax               ; eax = --j if i != 2

; printf( "i=%d, j+5d\n", i, j ); (i in edx, j in eax)

        lea     rcx, OFFSET FLAT:??_C@_0M@MJLDLLNK@i?$DN?$CFd?0?5j?$DN?$CFd?6?$AA@
        call    printf

; return 0;

        xor     eax, eax

        add     rsp, 40                                  ; 00000028H
        ret     0
main    ENDP
_TEXT   ENDS
; Function compile flags: /Ogtpy
; File c:\program files (x86)\windows kits\10\include\10.0.17134.0\ucrt\stdio.h
;       COMDAT printf
_TEXT   SEGMENT
    .
    .
    .
    END
```

5.2.3.2 GCC Assembly Language Output (PowerPC)

Like Visual C++, GCC doesn't insert the C source code into the assembly output file. In GCC's case, it's somewhat understandable: producing assembly output is something it always does (rather than something it does because of a user request). By not inserting the C source code into the output file, GCC can cut down compilation times by a small amount (because the compiler won't have to write the C source code data and the assembler won't have to read this data). Here's the output of GCC for a PowerPC processor when using the command line `gcc -O1 -S t1.c`:

```
gcc -O1 -S t1.c

.data
.cstring
```

```
        .align 2
LC0:
        .ascii "i=%d, j=%d\12\0"
.text
        .align 2
        .globl _main
_main:
LFB1:
        mflr r0
        stw r31,-4(r1)
LCFI0:
        stw r0,8(r1)
LCFI1:
        stwu r1,-80(r1)
LCFI2:
        bcl 20,31,L1$pb
L1$pb:
        mflr r31
        mr r11,r3
        lwz r9,0(r4)
        lbz r0,0(r9)
        extsb r5,r0
        cmpwi cr0,r3,2
        bne+ cr0,L2
        addi r5,r5,1
        b L3
L2:
        addi r5,r5,-1
L3:
        addis r3,r31,ha16(LC0-L1$pb)
        la r3,lo16(LC0-L1$pb)(r3)
        mr r4,r11
        bl L_printf$stub
        li r3,0
        lwz r0,88(r1)
        addi r1,r1,80
        mtlr r0
        lwz r31,-4(r1)
        blr
LFE1:
.data
.picsymbol_stub
L_printf$stub:
        .indirect_symbol _printf
        mflr r0
        bcl 20,31,L0$_printf
L0$_printf:
        mflr r11
        addis r11,r11,ha16(L_printf$lazy_ptr-L0$_printf)
        mtlr r0
        lwz r12,lo16(L_printf$lazy_ptr-L0$_printf)(r11)
        mtctr r12
        addi r11,r11,lo16(L_printf$lazy_ptr-L0$_printf)
        bctr
.data
```

```
.lazy_symbol_pointer
L_printf$lazy_ptr:
        .indirect_symbol _printf
        .long dyld_stub_binding_helper
.data
.constructor
.data
.destructor
        .align 1
```

As you can see, the output of GCC is quite sparse. Of course, as this is PowerPC assembly language, it's not really practical to compare this assembly output to the 80x86 output from the Visual C++ compiler.

5.2.3.3 GCC Assembly Language Output (80x86)

The following code provides the GCC compilation to x86-64 assembly code for the *t1.c* source file:

```
      .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 13
    .globl  _main                   ## -- Begin function main
    .p2align    4, 0x90
_main:                              ## @main
    .cfi_startproc
## BB#0:
    pushq   %rbp
Lcfi0:
    .cfi_def_cfa_offset 16
Lcfi1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
Lcfi2:
    .cfi_def_cfa_register %rbp
    movl    %edi, %ecx
    movq    (%rsi), %rax
    movsbl  (%rax), %eax
    cmpl    $2, %ecx
    movl    $1, %esi
    movl    $-1, %edx
    cmovel  %esi, %edx
    addl    %eax, %edx
    leaq    L_.str(%rip), %rdi
    xorl    %eax, %eax
    movl    %ecx, %esi
    callq   _printf
    xorl    %eax, %eax
    popq    %rbp
    retq
    .cfi_endproc
                                        ## -- End function
    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
```

```
    .asciz  "i=%d, j=%d\n"

.subsections_via_symbols
```

This example should help demonstrate that the massive amount of code that GCC emitted for the PowerPC is more a function of the machine's architecture than of the compiler. If you compare this to the code that other compilers emit, you'll discover that it is roughly equivalent.

5.2.3.4 GCC Assembly Language Output (ARMv7)

The following code provides the GCC compilation to ARMv6 assembly code for the *t1.c* source file as compiled on a Raspberry Pi (running 32-bit Raspian):

```
..arch armv6
    .eabi_attribute 27, 3
    .eabi_attribute 28, 1
    .fpu vfp
    .eabi_attribute 20, 1
    .eabi_attribute 21, 1
    .eabi_attribute 23, 3
    .eabi_attribute 24, 1
    .eabi_attribute 25, 1
    .eabi_attribute 26, 2
    .eabi_attribute 30, 2
    .eabi_attribute 34, 1
    .eabi_attribute 18, 4
    .file   "t1.c"
    .section    .text.startup,"ax",%progbits
    .align  2
    .global main
    .type   main, %function
main:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 0, uses_anonymous_args = 0
    stmfd   sp!, {r3, lr}
    cmp r0, #2
    ldr r3, [r1]
    mov r1, r0
    ldr r0, .L5
    ldrb    r2, [r3]    @ zero_extendqisi2
    addeq   r2, r2, #1
    subne   r2, r2, #1
    bl  printf
    mov r0, #0
    ldmfd   sp!, {r3, pc}
.L6:
    .align  2
.L5:
    .word   .LC0
    .size   main, .-main
    .section    .rodata.str1.4,"aMS",%progbits,1
    .align  2
```

```
.LC0:
    .ascii  "i=%d, j=%d\012\000"
    .ident  "GCC: (Raspbian 4.9.2-10) 4.9.2"
    .section    .note.GNU-stack,"",%progbits
```

Note that the `@` denotes a comment in this source code; Gas ignores everything from the `@` to the end of the line.

5.2.3.5 Swift Assembly Language Output (x86-64)

Given a Swift source file *main.swift*, you can request an assembly language output file from the macOS Swift compiler using the following command:

```
swiftc -O -emit-assembly main.swift -o result.asm
```

This will produce the *result.asm* output assembly language file. Consider the following Swift source code:

```
import Foundation

var i:Int = 0;
var j:Int = 1;


    if( i == 2 )
    {
        i = i + 1
    }
    else
    {
        i = i - 1
    }

    print( "i=\(i), j=\(j)" )
```

Compiling this with the previous command line produces a rather long assembly language output file; here is the main procedure from that code:

```
_main:
.cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %rbx
    pushq   %rax
    .cfi_offset %rbx, -56
    .cfi_offset %r12, -48
    .cfi_offset %r13, -40
    .cfi_offset %r14, -32
    .cfi_offset %r15, -24
    movq    $1, _$S6result1jSivp(%rip)
    movq    $-1, _$S6result1iSivp(%rip)
    movq    _$Ss23_ContiguousArrayStorageCyypGML(%rip), %rdi
    testq   %rdi, %rdi
    jne LBB0_3
    movq    _$SypN@GOTPCREL(%rip), %rsi
    addq    $8, %rsi
    xorl    %edi, %edi
    callq   _$Ss23_ContiguousArrayStorageCMa
    movq    %rax, %rdi
    testq   %rdx, %rdx
    jne LBB0_3
    movq    %rdi, _$Ss23_ContiguousArrayStorageCyypGML(%rip)
LBB0_3:
    movabsq $8589934584, %r12
    movl    48(%rdi), %esi
    movzwl  52(%rdi), %edx
    addq    $7, %rsi
    andq    %r12, %rsi
    addq    $32, %rsi
    orq $7, %rdx
    callq   _swift_allocObject
    movq    %rax, %r14
    movq    _$Ss27_ContiguousArrayStorageBaseC16countAndCapacitys01_B4BodyVvpWvd@GOTPCREL(%rip), %rbx
    movq    (%rbx), %r15
    movaps  LCPI0_0(%rip), %xmm0
    movups  %xmm0, (%r14,%r15)
    movq    _$SSSN@GOTPCREL(%rip), %rax
    movq    %rax, 56(%r14)
    movq    _$Ss23_ContiguousArrayStorageCySSGML(%rip), %rdi
    testq   %rdi, %rdi
    jne LBB0_6
    movq    _$SSSN@GOTPCREL(%rip), %rsi
    xorl    %edi, %edi
    callq   _$Ss23_ContiguousArrayStorageCMa
    movq    %rax, %rdi
    testq   %rdx, %rdx
    jne LBB0_6
    movq    %rdi, _$Ss23_ContiguousArrayStorageCySSGML(%rip)
    movq    (%rbx), %r15
LBB0_6:
    movl    48(%rdi), %esi
    movzwl  52(%rdi), %edx
    addq    $7, %rsi
    andq    %r12, %rsi
    addq    $80, %rsi
    orq $7, %rdx
    callq   _swift_allocObject
    movq    %rax, %rbx
    movaps  LCPI0_1(%rip), %xmm0
    movups  %xmm0, (%rbx,%r15)
    movabsq $-2161727821137838080, %r15
```

```
movq    %r15, %rdi
callq   _swift_bridgeObjectRetain
movl    $15721, %esi
movq    %r15, %rdi
callq   _$Ss27_toStringReadOnlyStreamableySSxs010TextOutputE0RzlFSS_Tg5Tf4x_n
movq    %rax, %r12
movq    %rdx, %r13
movq    %r15, %rdi
callq   _swift_bridgeObjectRelease
movq    %r12, 32(%rbx)
movq    %r13, 40(%rbx)
movq    _$S6result1iSivp(%rip), %rdi
callq   _$Ss26_toStringReadOnlyPrintableySSxs06CustomB11ConvertibleRzlFSi_Tg5
movq    %rax, 48(%rbx)
movq    %rdx, 56(%rbx)
movabsq $-2017612633061982208, %r15
movq    %r15, %rdi
callq   _swift_bridgeObjectRetain
movl    $1030365228, %esi
movq    %r15, %rdi
callq   _$Ss27_toStringReadOnlyStreamableySSxs010TextOutputE0RzlFSS_Tg5Tf4x_n
movq    %rax, %r12
movq    %rdx, %r13
movq    %r15, %rdi
callq   _swift_bridgeObjectRelease
movq    %r12, 64(%rbx)
movq    %r13, 72(%rbx)
movq    _$S6result1jSivp(%rip), %rdi
callq   _$Ss26_toStringReadOnlyPrintableySSxs06CustomB11ConvertibleRzlFSi_Tg5
movq    %rax, 80(%rbx)
movq    %rdx, 88(%rbx)
movabsq $-2305843009213693952, %r15
movq    %r15, %rdi
callq   _swift_bridgeObjectRetain
xorl    %esi, %esi
movq    %r15, %rdi
callq   _$Ss27_toStringReadOnlyStreamableySSxs010TextOutputE0RzlFSS_Tg5Tf4x_n
movq    %rax, %r12
movq    %rdx, %r13
movq    %r15, %rdi
callq   _swift_bridgeObjectRelease
movq    %r12, 96(%rbx)
movq    %r13, 104(%rbx)
movq    %rbx, %rdi
callq   _$SSS19stringInterpolationS2Sd_tcfCTf4nd_n
movq    %rax, 32(%r14)
movq    %rdx, 40(%r14)
callq   _$Ss5print_9separator10terminatoryypd_S2StFfA0_
movq    %rax, %r12
movq    %rdx, %r15
callq   _$Ss5print_9separator10terminatoryypd_S2StFfA1_
movq    %rax, %rbx
movq    %rdx, %rax
movq    %r14, %rdi
movq    %r12, %rsi
movq    %r15, %rdx
movq    %rbx, %rcx
movq    %rax, %r8
callq   _$Ss5print_9separator10terminatoryypd_S2StF
movq    %r14, %rdi
callq   _swift_release
movq    %r12, %rdi
callq   _swift_bridgeObjectRelease
movq    %rbx, %rdi
callq   _swift_bridgeObjectRelease
xorl    %eax, %eax
addq    $8, %rsp
popq    %rbx
popq    %r12
popq    %r13
popq    %r14
popq    %r15
popq    %rbp
retq
.cfi_endproc
```

As you can see, Swift doesn't generate very optimal code compared to C++. In fact, hundreds of additional lines of code have been omitted from this listing to save space.

5.2.4 Assembly Output Analysis

Unless you're well versed in assembly language programming, analyzing assembly output can be tricky. If you're not an assembly language programmer, about the best you can do is count instructions and assume that if a compiler option (or reorganization of your HLL source code) produces fewer instructions, the result is better. In reality, this assumption isn't always correct. Some machine instructions (particularly on CISC processors such as the 80x86) require substantially more time to execute than others. A sequence of three or more instructions on a processor such as the 80x86 could execute faster than a single instruction that does the same operation. Fortunately, a compiler is not likely to produce both of these sequences based on a reorganization of your high-level source code. Therefore, you don't usually have to worry about such issues when examining the assembly output.

Note that some compilers will produce two different sequences if you change the optimization level. This is because certain optimization settings tell the compiler to favor shorter programs, while other optimization settings tell the compiler to favor faster execution. The optimization setting that favors smaller executable files will probably pick the single instruction over the three instructions that do the same work (assuming those three instructions compile into more code); the optimization setting that favors speed will probably pick the faster instruction sequence.

This section uses various C/C++ compilers in its examples, but you should remember that compilers for other languages also provide the ability to emit assembly code. You'll have to check your compiler's documentation to see if this is possible and what options you can use to produce the assembly output. Some compilers (Visual C++, for example) provide an integrated development environment (IDE) that you may use in place of a command-line tool. Even though most compilers that work through an IDE also work from the command line, you can usually specify assembly output from within an IDE as well as from the command line. Once again, see your compiler vendor's documentation for details.

5.3 Using Object Code Utilities to Analyze Compiler Output

Although many compilers provide an option to emit assembly language rather than object code, a large number of compilers do not; they can only emit binary machine code to an object code file. Analyzing this kind of compiler output will be a bit more work, and it's going to require some specialized tools. If your compiler emits object code files (such as PE/COFF or ELF files) to be fed into a linker, you can probably find an "object code dump" utility that will prove quite useful for analyzing the compiler's output. For example, Microsoft provides the *dumpbin.exe* program, and the FSF/GNU *dumpobj* program has similar capabilities for ELF files under Linux and other operating systems. In the following subsections, we'll take a look at using these two tools when analyzing compiler output.

One nice feature of working with object files is that they usually contain symbolic information. That is, in addition to binary machine code, the object file contains strings specifying identifier names that appear in the source file (such information does not normally appear in an executable file). Object code utilities can usually display these symbolic names within the machine instructions that reference the memory locations associated with these symbols. Though these object code utilities can't automatically correlate the HLL source code with the machine code, having the symbolic information available can help you when you're studying their output, because it's much easier to read names like `JumpTable` than memory addresses like `$401_1000`.

5.3.1 The Microsoft dumpbin.exe Utility

Microsoft's `dumpbin` command-line tool allows you to examine the contents of a Microsoft PE/COFF file.[2] You run the program as follows:

```
dumpbin options filename
```

The *`filename`* parameter is the name of the *.obj* file that you wish to examine, and the *`options`* parameter is a set of optional command-line arguments that specify the type of information you want to display. These options each begin with a forward slash (`/`). We'll take a look at each of the

2. Actually, *dumpbin.exe* is just a *wrapper* program for *link.exe*; that is, it processes its own command-line parameters, builds a *link.exe* command line, and runs the linker.

possible options in a moment. First, here is a listing of the possible objects (obtained via the /? command-line option):

```
Microsoft (R) COFF/PE Dumper Version 14.00.24234.1
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: dumpbin options files

   options:

      /ALL
      /ARCHIVEMEMBERS
      /CLRHEADER
      /DEPENDENTS
      /DIRECTIVES
      /DISASM[:{BYTES|NOBYTES}]
      /ERRORREPORT:{NONE|PROMPT|QUEUE|SEND}
      /EXPORTS
      /FPO
      /HEADERS
      /IMPORTS[:filename]
      /LINENUMBERS
      /LINKERMEMBER[:{1|2}]
      /LOADCONFIG
      /NOLOGO
      /OUT:filename
      /PDATA
      /PDBPATH[:VERBOSE]
      /RANGE:vaMin[,vaMax]
      /RAWDATA[:{NONE|1|2|4|8}[,#]]
      /RELOCATIONS
      /SECTION:name
      /SUMMARY
      /SYMBOLS
      /TLS
      /UNWINDINFO
```

Though the primary use of `dumpbin` is to look at the object code a compiler produces, it also displays a considerable amount of interesting information about a PE/COFF file. For more information on the meaning of many of the `dumpbin` command-line options, review "Object File Formats" on page 71 or "Executable File Formats" on page 80.

The following subsections describe several of the possible `dumpbin` command-line options and provide example output for a simple "Hello World" program written in C:

```
#include <stdio.h>

int main( int argc, char **argv)
{
    printf( "Hello World\n" );
}
```

5.3.1.1 /all

The /all command-line option instructs dumpbin to display all the information it can except for a disassembly of the code found in the object file. The problem with this approach is that an *.exe* file contains all the routines from the language's standard library (such as the C Standard Library) that the linker has merged into the application. When analyzing compiler output in order to improve your application's code, wading through all this extra information about code outside your program can be tedious. Fortunately, there's an easy way to pare down the unnecessary information—run dumpbin on your object (*.obj*) files rather than your executable (*.exe*) files. Here is the (shortened) output that dumpbin produces for the "Hello World" example:

```
G:\>dumpbin /all hw.obj
Microsoft (R) COFF/PE Dumper Version 14.00.24234.1
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file hw.obj

File Type: COFF OBJECT

FILE HEADER VALUES
            8664 machine (x64)
               D number of sections
        5B2C175F time date stamp Thu Jun 21 14:23:43 2018
             466 file pointer to symbol table
              2D number of symbols
               0 size of optional header
               0 characteristics

SECTION HEADER #1
.drectve name
       0 physical address
       0 virtual address
      2F size of raw data
     21C file pointer to raw data (0000021C to 0000024A)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
  100A00 flags
         Info
         Remove
         1 byte align

Hundreds of lines deleted...

  Summary

           D .data
          70 .debug$S
          2F .drectve
          24 .pdata
          C2 .text$mn
          18 .xdata
```

This example deletes the bulk of the output from this command (to spare you having to read a dozen or so extra pages). Try executing the `/all` command yourself to see the quantity of output you get. In general, though, use this option with care.

5.3.1.2 /disasm

The `/disasm` command-line option is the one of greatest interest to us. It produces a disassembled listing of the object file. As with the `/all` option, you shouldn't try to disassemble an *.exe* file using `dumpbin`. The disassembled listing you'll get will be quite long, and the vast majority of the code will probably be the listings of all the library routines your application calls. For example, the simple "Hello World" application generates over 5,000 lines of disassembled code. All but a small handful of those statements correspond to library routines. Wading through that amount of code will prove overwhelming to most people.

However, if you disassemble the *hw.obj* file rather than the executable file, here's the output you typically get:

```
Microsoft (R) COFF/PE Dumper Version 14.00.24234.1
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file hw.obj

File Type: COFF OBJECT

main:
  0000000000000000: 48 89 54 24 10     mov         qword ptr [rsp+10h],rdx
  0000000000000005: 89 4C 24 08        mov         dword ptr [rsp+8],ecx
  0000000000000009: 48 83 EC 28        sub         rsp,28h
  000000000000000D: 48 8D 0D 00 00 00  lea         rcx,[$SG4247]
                    00
  0000000000000014: E8 00 00 00 00     call        printf
  0000000000000019: 33 C0              xor         eax,eax
  000000000000001B: 48 83 C4 28        add         rsp,28h
  000000000000001F: C3                 ret

// Uninterested code emitted by dumpbin.exe left out...

  Summary

           D .data
          70 .debug$S
          2F .drectve
          24 .pdata
          C2 .text$mn
          28 .xdata
```

If you look closely at this disassembled code, you'll find the major problem with disassembling object files rather than executable files—most addresses in the code are relocatable addresses, which appear as `$00000000` in the object code listing. As a result, you'll probably have a hard time figuring out what the various assembly statements are doing. For example, in the *hw.obj*'s disassembled listing you see the following two statements:

```
000000000000000D: 48 8D 0D 00 00 00  lea        rcx,[$SG4247]
                  00
0000000000000014: E8 00 00 00 00     call       printf
```

The `lea` instruction opcode is the 3-byte sequence `48 8D 0D` (which includes an REX opcode prefix byte). The address of the `"Hello World"` string is not `00 00 00 00` (the 4 bytes following the opcode); instead, it is a relocatable address that the linker/system fills in later. If you run `dumpbin` on *hw.obj* with the `/all` command-line option, you'll notice that this file has two relocation entries:

```
RELOCATIONS #4
                                                    Symbol    Symbol
 Offset    Type              Applied To             Index     Name
 --------  ----------------  -----------------  --------  ------
 00000010  REL32                      00000000         8  $SG4247
 00000015  REL32                      00000000        15  printf
```

The Offset column tells you the byte offset into the file where the relocations are to be applied. In the preceding disassembly, note that the `lea` instruction starts at offset `$d`, so the actual displacement is at offset `$10`. Similarly, the `call` instruction begins at offset `$14`, so the address of the actual routine that needs to be patched is 1 byte later, at offset `$15`. From the relocation information that `dumpbin` outputs, you can discern the symbols associated with these relocations. (`$SG4247` is an internal symbol that the C compiler generates for the `"Hello World"` string. And `printf` is, obviously, the name associated with the C `printf()` function.)

Cross-referencing every call and memory reference against the relocation list may seem like a pain, but at least you get symbolic names when you do this.

Consider the first few lines of the disassembled code when you apply the `/disasm` option to the *hw.exe* file:

```
0000000140001009: 48 83 EC 28        sub        rsp,28h
000000014000100D: 48 8D 0D EC DF 01  lea        rcx,[000000014001F000h]
                  00
0000000140001014: E8 67 00 00 00     call       0000000140001080
0000000140001019: 33 C0              xor        eax,eax
000000014000101B: 48 83 C4 28        add        rsp,28h
000000014000101F: C3                 ret
                             .
                             .
                             .
```

Notice that the linker has filled in the addresses (relative to the load address for the file) of the offset $SG4247 and print labels. This may seem somewhat convenient; however, note that these labels (especially the printf label) are no longer present in the file. When you are reading the disassembled output, the absence of these labels can make it very difficult to figure out which machine instructions correspond to HLL statements. This is yet another reason why you should use object files rather than executable files when running dumpbin.

If you think it's going to be a major pain to read the disassembled output of the dumpbin utility, don't worry: for optimization purposes, you're often more interested in the code differences between two versions of an HLL program than in figuring out what each machine instruction does. Therefore, you can easily determine which machine instructions are affected by a change in your code by running dumpbin on two versions of your object files (one before the change to the HLL code and one created afterward). For example, consider the following modification to the "Hello World" program:

```
#include <stdio.h>

int main( int argc, char **argv)
{
        char *hwstr = "Hello World\n";

        printf( hwstr );
}
```

Here's the disassembly output that dumpbin produces:

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
  0000000140001000: 48 89 54 24 10     mov        qword ptr [rsp+10h],rdx
  0000000140001005: 89 4C 24 08        mov        dword ptr [rsp+8],ecx
  0000000140001009: 48 83 EC 28        sub        rsp,28h
  000000014000100D: 48 8D 0D EC DF 01  lea        rcx,[000000014001F000h]
                    00
  0000000140001014: E8 67 00 00 00     call       0000000140001080
  0000000140001019: 33 C0              xor        eax,eax
  000000014000101B: 48 83 C4 28        add        rsp,28h
  000000014000101F: C3                 ret
```

By comparing this output with the previous assembly output (either manually or by running one of the programs based on the Unix diff utility), you can see the effect of the change to your HLL source code on the emitted machine code.

NOTE *The section "Comparing Output from Two Compilations" on page 137 discusses the merits of both comparison methods (manual and* diff*-based).*

5.3.1.3 /headers

The /headers option instructs dumpbin to display the COFF header files and section header files. The /all option also prints this information, but /header displays *only* the header information without all the other output. Here's the sample output for the "Hello World" executable file:

```
G:\WGC>dumpbin /headers hw.exe
Microsoft (R) COFF/PE Dumper Version 14.00.24234.1
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file hw.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
            8664 machine (x64)
               6 number of sections
        5B2C1A9F time date stamp Thu Jun 21 14:37:35 2018
               0 file pointer to symbol table
               0 number of symbols
              F0 size of optional header
              22 characteristics
                   Executable
                   Application can handle large (>2GB) addresses

OPTIONAL HEADER VALUES
             20B magic # (PE32+)
           14.00 linker version
           13400 size of code
            D600 size of initialized data
               0 size of uninitialized data
            1348 entry point (0000000140001348)
            1000 base of code
       140000000 image base (0000000140000000 to 0000000140024FFF)
            1000 section alignment
             200 file alignment
            6.00 operating system version
            0.00 image version
            6.00 subsystem version
               0 Win32 version
           25000 size of image
             400 size of headers
               0 checksum
               3 subsystem (Windows CUI)
            8160 DLL characteristics
                   High Entropy Virtual Addresses
                   Dynamic base
                   NX compatible
                   Terminal Server Aware
          100000 size of stack reserve
```

```
            1000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
               0 loader flags
              10 number of directories
               0 [       0] RVA [size] of Export Directory
           1E324 [      28] RVA [size] of Import Directory
               0 [       0] RVA [size] of Resource Directory
           21000 [    126C] RVA [size] of Exception Directory
               0 [       0] RVA [size] of Certificates Directory
           24000 [     620] RVA [size] of Base Relocation Directory
           1CDA0 [      1C] RVA [size] of Debug Directory
               0 [       0] RVA [size] of Architecture Directory
               0 [       0] RVA [size] of Global Pointer Directory
               0 [       0] RVA [size] of Thread Storage Directory
           1CDC0 [      94] RVA [size] of Load Configuration Directory
               0 [       0] RVA [size] of Bound Import Directory
           15000 [     230] RVA [size] of Import Address Table Directory
               0 [       0] RVA [size] of Delay Import Directory
               0 [       0] RVA [size] of COM Descriptor Directory
               0 [       0] RVA [size] of Reserved Directory


SECTION HEADER #1
   .text name
   1329A virtual size
    1000 virtual address (0000000140001000 to 0000000140014299)
   13400 size of raw data
     400 file pointer to raw data (00000400 to 000137FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read

SECTION HEADER #2
  .rdata name
    9A9A virtual size
   15000 virtual address (0000000140015000 to 000000014001EA99)
    9C00 size of raw data
   13800 file pointer to raw data (00013800 to 0001D3FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only

  Debug Directories
```

```
        Time Type        Size      RVA  Pointer
    -------- ------- -------- -------- --------
    5B2C1A9F coffgrp      2CC 0001CFC4    1B7C4

SECTION HEADER #3
   .data name
    1BA8 virtual size
   1F000 virtual address (000000014001F000 to 0000000140020BA7)
     A00 size of raw data
   1D400 file pointer to raw data (0001D400 to 0001DDFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write

SECTION HEADER #4
  .pdata name
    126C virtual size
   21000 virtual address (0000000140021000 to 000000014002226B)
    1400 size of raw data
   1DE00 file pointer to raw data (0001DE00 to 0001F1FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only

SECTION HEADER #5
  .gfids name
      D4 virtual size
   23000 virtual address (0000000140023000 to 00000001400230D3)
     200 size of raw data
   1F200 file pointer to raw data (0001F200 to 0001F3FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only

SECTION HEADER #6
  .reloc name
     620 virtual size
   24000 virtual address (0000000140024000 to 000000014002461F)
     800 size of raw data
   1F400 file pointer to raw data (0001F400 to 0001FBFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only

  Summary

        2000 .data
        1000 .gfids
        2000 .pdata
        A000 .rdata
        1000 .reloc
       14000 .text
```

Review the discussion of object file formats in Chapter 4 (see "Object File Formats" on page 71) to make sense of the information that `dumpbin` outputs when you specify the `/headers` option.

5.3.1.4 /imports

The `/imports` option lists all of the dynamic-link symbols that the operating system must supply when the program loads into memory. This information isn't particularly useful for analyzing code emitted for HLL statements, so this chapter won't mention this option further.

5.3.1.5 /relocations

The `/relocations` option displays all the relocation objects in the file. This command is quite useful because it provides a list of all the symbols for the program and the offsets of their use in the disassembly listing. Of course, the `/all` option also presents this information, but `/relocations` provides just this information without anything else.

5.3.1.6 Other dumpbin.exe Command-Line Options

The `dumpbin` utility supports many more command-line options beyond those this chapter describes. As noted earlier, you can get a list of all possible options by specifying `/?` on the command line when running `dumpbin`. You can also read more online at *https://docs.microsoft.com/en-us/cpp/build/reference/dumpbin-reference?view=vs-2019/.*

5.3.2 The FSF/GNU objdump Utility

If you're running the GNU toolset on your operating system (for example, under Linux, Mac, or BSD), then you can use the FSF/GNU `objdump` utility to examine the object files produced by GCC and other GNU-compliant tools. Here are the command-line options it supports:

```
Usage: objdump <option(s)> <file(s)>
Usage: objdump <option(s)> <file(s)>
Display information from object <file(s)>.
```

```
At least one of the following switches must be given:
 -a, --archive-headers    Display archive header information
 -f, --file-headers       Display the contents of the overall file header
 -p, --private-headers    Display object format specific file header contents
 -P, --private=OPT,OPT... Display object format specific contents
 -h, --[section-]headers  Display the contents of the section headers
 -x, --all-headers        Display the contents of all headers
 -d, --disassemble        Display assembler contents of executable sections
 -D, --disassemble-all    Display assembler contents of all sections
 -S, --source             Intermix source code with disassembly
 -s, --full-contents      Display the full contents of all sections requested
 -g, --debugging          Display debug information in object file
 -e, --debugging-tags     Display debug information using ctags style
 -G, --stabs              Display (in raw form) any STABS info in the file
 -W[lLiaprmfFsoRt] or
 --dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
         =frames-interp,=str,=loc,=Ranges,=pubtypes,
         =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
         =addr,=cu_index]
                          Display DWARF info in the file
 -t, --syms               Display the contents of the symbol table(s)
 -T, --dynamic-syms       Display the contents of the dynamic symbol table
 -r, --reloc              Display the relocation entries in the file
 -R, --dynamic-reloc      Display the dynamic relocation entries in the file
 @<file>                  Read options from <file>
 -v, --version            Display this program's version number
 -i, --info               List object formats and architectures supported
 -H, --help               Display this information

The following switches are optional:
 -b, --target=BFDNAME           Specify the target object format as BFDNAME
 -m, --architecture=MACHINE     Specify the target architecture as MACHINE
 -j, --section=NAME             Only display information for section NAME
 -M, --disassembler-options=OPT Pass text OPT on to the disassembler
 -EB --endian=big               Assume big endian format when disassembling
 -EL --endian=little            Assume little endian format when disassembling
     --file-start-context       Include context from start of file (with -S)
 -I, --include=DIR              Add DIR to search list for source files
 -l, --line-numbers             Include line numbers and filenames in output
 -F, --file-offsets             Include file offsets when displaying information
 -C, --demangle[=STYLE]         Decode mangled/processed symbol names
                                The STYLE, if specified, can be `auto', `gnu',
                                 `lucid', `arm', `hp', `edg', `gnu-v3', `java'
                                 or `gnat'
 -w, --wide                     Format output for more than 80 columns
 -z, --disassemble-zeroes       Do not skip blocks of zeroes when disassembling
     --start-address=ADDR       Only process data whose address is >= ADDR
     --stop-address=ADDR        Only process data whose address is <= ADDR
     --prefix-addresses         Print complete address alongside disassembly
     --[no-]show-raw-insn       Display hex alongside symbolic disassembly
     --insn-width=WIDTH         Display WIDTH bytes on a single line for -d
     --adjust-vma=OFFSET        Add OFFSET to all displayed section addresses
     --special-syms             Include special symbols in symbol dumps
```

```
      --prefix=PREFIX            Add PREFIX to absolute paths for -S
      --prefix-strip=LEVEL       Strip initial directory names for -S
      --dwarf-depth=N            Do not display DIEs at depth N or greater
      --dwarf-start=N            Display DIEs starting with N, at the same depth
                                 or deeper
      --dwarf-check              Make additional dwarf internal consistency checks.

objdump: supported targets: elf64-x86-64 elf32-i386 elf32-iamcu elf32-x86-64 a.out-i386-linux
pei-i386 pei-x86-64 elf64-l1om elf64-k1om elf64-little elf64-big elf32-little elf32-big pe-x86-
64 pe-bigobj-x86-64 pe-i386 plugin srec symbolsrec verilog tekhex binary ihex
objdump: supported architectures: i386 i386:x86-64 i386:x64-32 i8086 i386:intel i386:x86-
64:intel i386:x64-32:intel i386:nacl i386:x86-64:nacl i386:x64-32:nacl iamcu iamcu:intel l1om
l1om:intel k1om k1om:intel plugin
The following i386/x86-64 specific disassembler options are supported for use
with the -M switch (multiple options should be separated by commas):
  x86-64          Disassemble in 64bit mode
  i386            Disassemble in 32bit mode
  i8086           Disassemble in 16bit mode
  att             Display instruction in AT&T syntax
  intel           Display instruction in Intel syntax
  att-mnemonic    Display instruction in AT&T mnemonic
  intel-mnemonic  Display instruction in Intel mnemonic
  addr64          Assume 64bit address size
  addr32          Assume 32bit address size
  addr16          Assume 16bit address size
  data32          Assume 32bit data size
  data16          Assume 16bit data size
  suffix          Always display instruction suffix in AT&T syntax
  amd64           Display instruction in AMD64 ISA
  intel64         Display instruction in Intel64 ISA
Report bugs to <http://www.sourceware.org/bugzilla/>.
```

Given the following *m.hla* source code fragment:

```
begin t;

        // test mem.alloc and mem.free:

        for( mov( 0, ebx ); ebx < 16; inc( ebx )) do

                // Allocate lots of storage:

                for( mov( 0, ecx ); ecx < 65536; inc( ecx )) do

                        rand.range( 1, 256 );
                        malloc( eax );
                        mov( eax, ptrs[ ecx*4 ] );

                endfor;
                   .
                   .
                   .
```

Here is some sample output produced on the 80x86, created with the Linux command line `objdump -S m`:

```
        objdump -S m


 0804807e <_HLAMain>:
 804807e:   89 e0                   mov    %esp,%eax

        .
        . // Some deleted code here,
        . // that HLA automatically generated.
        .

 80480ae:   bb 00 00 00 00          mov    $0x0,%ebx
 80480b3:   eb 2a                   jmp    80480df <StartFor__hla_2124>

080480b5 <for__hla_2124>:
 80480b5:   b9 00 00 00 00          mov    $0x0,%ecx
 80480ba:   eb 1a                   jmp    80480d6 <StartFor__hla_2125>

080480bc <for__hla_2125>:
 80480bc:   6a 01                   push   $0x1
 80480be:   68 00 01 00 00          push   $0x100
 80480c3:   e8 64 13 00 00          call   804942c <RAND_RANGE>
 80480c8:   50                      push   %eax
 80480c9:   e8 6f 00 00 00          call   804813d <MEM_ALLOC1>
 80480ce:   89 04 8d 68 c9 04 08    mov    %eax,0x804c968(,%ecx,4)

080480d5 <continue__hla_2125>:
 80480d5:   41                      inc    %ecx

080480d6 <StartFor__hla_2125>:
 80480d6:   81 f9 00 00 01 00       cmp    $0x10000,%ecx
 80480dc:   72 de                   jb     80480bc <for__hla_2125>

080480de <continue__hla_2124>:
 80480de:   43                      inc    %ebx

080480df <StartFor__hla_2124>:
 80480df:   83 fb 10                cmp    $0x10,%ebx
 80480e2:   72 d1                   jb     80480b5 <for__hla_2124>

080480e4 <QuitMain__hla_>:
 80480e4:   b8 01 00 00 00          mov    $0x1,%eax
 80480e9:   31 db                   xor    %ebx,%ebx
 80480eb:   cd 80                   int    $0x80
 8048274:   bb 00 00 00 00          mov    $0x0,%ebx
 8048279:   e9 d5 00 00 00          jmp    8048353 <L1021_StartFor__hla_>
```

```
0804827e <L1021_for__hla_>:
 804827e:   b9 00 00 00 00          mov    $0x0,%ecx
 8048283:   eb 1a                   jmp    804829f <L1022_StartFor__hla_>

08048285 <L1022_for__hla_>:
 8048285:   6a 01                   push   $0x1
 8048287:   68 00 01 00 00          push   $0x100
 804828c:   e8 db 15 00 00          call   804986c <RAND_RANGE>
 8048291:   50                      push   %eax
 8048292:   e8 63 0f 00 00          call   80491fa <MEM_ALLOC>
 8048297:   89 04 8d 60 ae 04 08    mov    %eax,0x804ae60(,%ecx,4)

0804829e <L1022_continue__hla_>:
 804829e:   41                      inc    %ecx

0804829f <L1022_StartFor__hla_>:
 804829f:   81 f9 00 00 01 00       cmp    $0x10000,%ecx
 80482a5:   72 de                   jb     8048285 <L1022_for__hla_>

080482a7 <L1022_exitloop__hla_>:
 80482a7:   b9 00 00 00 00          mov    $0x0,%ecx
 80482ac:   eb 0d                   jmp    80482bb <L1023_StartFor__hla_>
```

These listings are only a fragment of the total code (which is why certain labels are absent). Nevertheless, you can see how the `objdump` utility can be useful for analyzing compiler output by allowing you to disassemble the object code for a certain code fragment.

Like `dumpbin`, `objdump` can display additional information beyond the machine code disassembly that may prove useful when you're analyzing compiler output. For most purposes, however, the GCC `-S` (assembly output) option is the most useful. Here's an example of a disassembly of some C code using the `objdump` utility. First, the original C code:

```
// Original C code:

#include <stdio.h>
int main( int argc, char **argv )
{
    int i,j,k;

    j = **argv;
    k = argc;
    i = j && k;
    printf( "%d\n", i );
    return 0;
}
```

Here's the Gas output (x86-64) from GCC for the C code:

```
    .file   "t.c"
    .section    .rodata
.LC0:
    .string "%d\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movl    %edi, -20(%rbp)
    movq    %rsi, -32(%rbp)
    movq    -32(%rbp), %rax
    movq    (%rax), %rax
    movzbl  (%rax), %eax
    movsbl  %al, %eax
    movl    %eax, -12(%rbp)
    movl    -20(%rbp), %eax
    movl    %eax, -8(%rbp)
    cmpl    $0, -12(%rbp)
    je  .L2
    cmpl    $0, -8(%rbp)
    je  .L2
    movl    $1, %eax
    jmp .L3
.L2:
    movl    $0, %eax
.L3:
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"
    .section    .note.GNU-stack,"",@progbits
```

Now here's the `objdump` disassembly of the main function:

```
.file   "t.c"

0000000000400526 <main>:
  400526:   55                      push   %rbp
  400527:   48 89 e5                mov    %rsp,%rbp
  40052a:   48 83 ec 20             sub    $0x20,%rsp
  40052e:   89 7d ec                mov    %edi,-0x14(%rbp)
  400531:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
  400535:   48 8b 45 e0             mov    -0x20(%rbp),%rax
  400539:   48 8b 00                mov    (%rax),%rax
  40053c:   0f b6 00                movzbl (%rax),%eax
  40053f:   0f be c0                movsbl %al,%eax
  400542:   89 45 f4                mov    %eax,-0xc(%rbp)
  400545:   8b 45 ec                mov    -0x14(%rbp),%eax
  400548:   89 45 f8                mov    %eax,-0x8(%rbp)
  40054b:   83 7d f4 00             cmpl   $0x0,-0xc(%rbp)
  40054f:   74 0d                   je     40055e <main+0x38>
  400551:   83 7d f8 00             cmpl   $0x0,-0x8(%rbp)
  400555:   74 07                   je     40055e <main+0x38>
  400557:   b8 01 00 00 00          mov    $0x1,%eax
  40055c:   eb 05                   jmp    400563 <main+0x3d>
  40055e:   b8 00 00 00 00          mov    $0x0,%eax
  400563:   89 45 fc                mov    %eax,-0x4(%rbp)
  400566:   8b 45 fc                mov    -0x4(%rbp),%eax
  400569:   89 c6                   mov    %eax,%esi
  40056b:   bf 14 06 40 00          mov    $0x400614,%edi
  400570:   b8 00 00 00 00          mov    $0x0,%eax
  400575:   e8 86 fe ff ff          callq  400400 <printf@plt>
  40057a:   b8 00 00 00 00          mov    $0x0,%eax
  40057f:   c9                      leaveq
  400580:   c3                      retq
```

As you can see, the assembly code output is somewhat easier to read than `objdump`'s output.

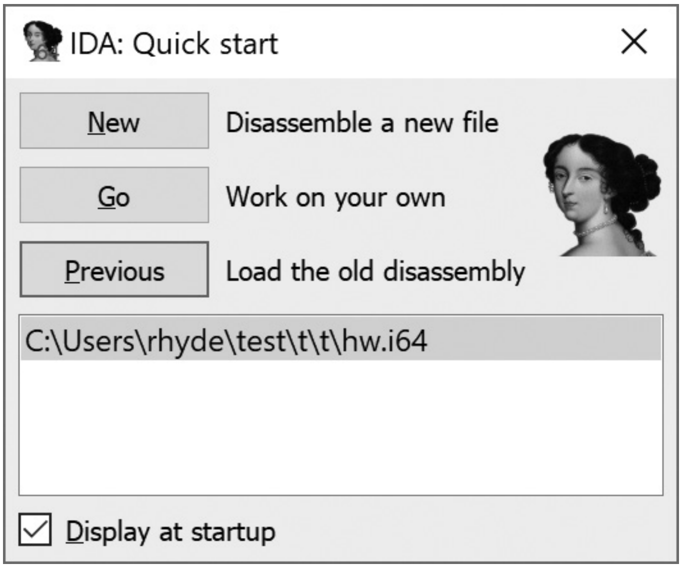
5.4 Using a Disassembler to Analyze Compiler Output

Although using an object code "dump" tool is one way to analyze compiler output, another possible solution is to run a *disassembler* on the executable file. A disassembler is a utility that translates binary machine code into human-readable assembly language statements ("human-readable" is debatable, but that's the idea, anyway). As such, it's another tool you can use to analyze compiler output.

There is a subtle, but important, difference between an object code dump utility (which contains a simple disassembler) and a sophisticated disassembler program. Object code dump utilities are automatic, but they can get easily confused if the object code contains tricky constructs (such as buried data in the instruction stream). An *automatic disassembler* is very convenient to use, requiring little expertise on the user's part, but rarely disassembles the machine code correctly. A full-blown *interactive disassembler*, on the other hand, requires more training to use properly, but is capable of disassembling tricky machine code sequences with a little help from its user. Therefore, decent disassemblers will work in situations where a simplistic object code dump utility will fail. Fortunately, most compilers do not always emit the kind of tricky code that confuses object code dump utilities, so you can sometimes get by without having to learn how to use a full-blown disassembler program. Nevertheless, having a disassembler handy can be useful in situations where a simplistic approach doesn't work.

Several "free" disassemblers are available. The one we'll cover in this chapter is IDA7. IDA is the freeware version of IDA Pro, a very capable and powerful commercial disassembler system (*https://www.hex-rays.com/products/ida/*).

When you first run IDA, it opens the window shown in Figure 5-1.


IDA: Quick start

New Disassemble a new file

Go Work on your own

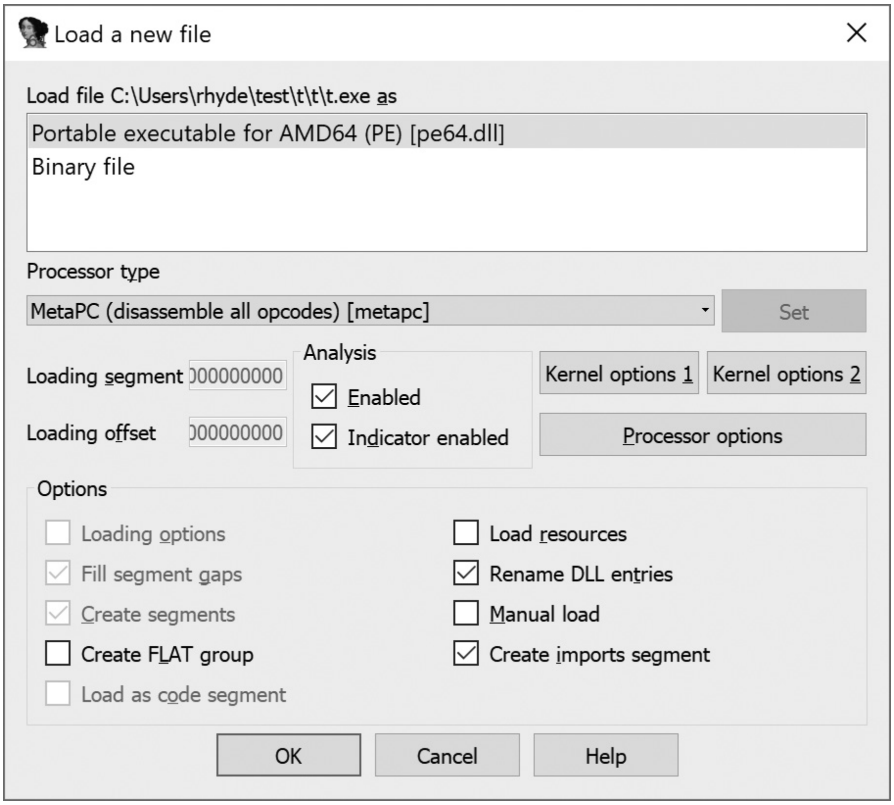
Previous Load the old disassembly

C:\Users\rhyde\test\t\t\hw.i64

☑ Display at startup

Figure 5-1: IDA opening window

Click the **New** button and type in the name of the *.exe* or *.obj* file you wish to disassemble. Once you enter an executable filename, IDA brings up the Format dialog shown in Figure 5-2. In this dialog, you can select the binary file type (for example, PE/COFF, PE64 executable file, or pure binary) and the options to use when disassembling the file. IDA does a good job of choosing reasonable default values for these options, so most of the time you'll just accept the defaults unless you're working with some weird binary files.



Load a new file

Load file C:\Users\rhyde\test\t\t\t.exe as

Portable executable for AMD64 (PE) [pe64.dll]
Binary file

Processor type: MetaPC (disassemble all opcodes) [metapc]

Loading segment: 0000000000
Loading offset: 0000000000

Analysis
- [✓] Enabled
- [✓] Indicator enabled

Options
- [ ] Loading options
- [✓] Fill segment gaps
- [✓] Create segments
- [ ] Create FLAT group
- [ ] Load as code segment
- [ ] Load resources
- [✓] Rename DLL entries
- [ ] Manual load
- [✓] Create imports segment

Figure 5-2: IDA executable file format dialog

Generally, IDA will figure out the appropriate file type information for a standard disassembly, then do an "automatic" disassembly of the object code file. To produce an assembly language output file, click **OK**. Here are the first few lines of the disassembly of the *t1.c* file given in "Example Assembly Language Output" on page 102:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main    proc    near
        sub     rsp, 28h
        mov     rax, [rdx]
        cmp     ecx, 2
        movsx   edx, byte ptr [rax]
        lea     eax, [rdx-1]
        lea     r8d, [rdx+1]
        mov     edx, ecx
        cmovnz  r8d, eax
        lea     rcx, aIDJD      ; "i=%d, j=%d\n"
        call    sub_140001040
        xor     eax, eax
        add     rsp, 28h
        retn
main endp
```

IDA is an *interactive* disassembler. This means that it provides lots of complex features that you can use to guide the disassembly to produce a more reasonable assembly language output file. However, its "automatic" mode of operation is generally all you'll need in order to examine compiler output files to assess their quality. For more details on IDA (freeware) or IDA Pro, see its documentation (*https://www.hex-rays.com/products/ida/support/*).

5.5 Using the Java Bytecode Disassembler to Analyze Java Output

Most Java compilers (particularly those from Oracle, Inc.) do not generate machine code directly. Instead, they generate Java bytecode (JBC), which computer systems then execute using a JBC interpreter. To improve performance, some Java interpreters run a just-in-time (JIT) compiler that translates JBC into native machine code during interpretation to improve performance (though the result is rarely as good as the machine code an optimizing compiler generates). Unfortunately, because the Java interpreter does this translation at runtime, it is difficult to analyze the machine code output from the Java compiler. It is, however, possible to analyze the JBC it produces. This can give you a better picture of what the compiler is doing with your Java code than simply guessing. Consider the following (relatively trivial) Java program:

```
public class Welcome
{
    public static void main( String[] args )
    {
        switch(5)
        {
            case 0:
                System.out.println("0");
                break;
            case 1:
                System.out.println("1");
                break;
            case 2:
            case 5:
                System.out.println("5");
                break;
            default:
                System.out.println("default" );
        }
        System.out.println( "Hello World" );
    }
}
```

Typically, you can compile this program (*Welcome.java*) using a command line of the form:

```
javac Welcome.java
```

This command produces the *Welcome.class* JBC file. You can use the following command to disassemble this file (to the standard output):

```
javap -c Welcome
```

Note that you do not include the *.class* file extension on the command line; the javap command automatically supplies it.

The javap command produces a bytecode disassembly listing similar to the following:

```
Compiled from "Welcome.java"
public class Welcome extends java.lang.Object{
public Welcome();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   iconst_5
   1:   tableswitch{ //0 to 5
        0: 40;
        1: 51;
        2: 62;
        3: 73;
        4: 73;
        5: 62;
        default: 73 }
   40:  getstatic   #2;     //Field java/lang/System.out:Ljava/io/PrintStream;
   43:  ldc #3;             //String 0
   45:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   48:  goto    81
   51:  getstatic   #2;     //Field java/lang/System.out:Ljava/io/PrintStream;
   54:  ldc #5;             //String 1
   56:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   59:  goto    81
   62:  getstatic   #2;     //Field java/lang/System.out:Ljava/io/PrintStream;
   65:  ldc #6;             //String 5
   67:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   70:  goto    81
   73:  getstatic   #2;     //Field java/lang/System.out:Ljava/io/PrintStream;
   76:  ldc #7;             //String default
   78:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   81:  getstatic   #2;     //Field java/lang/System.out:Ljava/io/PrintStream;
   84:  ldc #8;             //String Hello World
   86:  invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   89:  return

}
```

You can find documentation for the JBC mnemonics and the `javap` Java class file disassembler on Oracle.com (search the site for "javap" and "Java bytecode disassembler"). Also, the online chapters (specifically, Appendix D) accompanying this book discuss the Java VM bytecode assembly language.

5.6 Using the IL Disassembler to Analyze Microsoft C# and Visual Basic Output

Microsoft's .NET language compilers do not directly emit native machine code. Instead, they emit a special IL (intermediate language) code. This is quite similar, in principle, to Java bytecode or UCSD p-machine code. The .NET runtime system will compile IL executable files and run them using a JIT compiler.

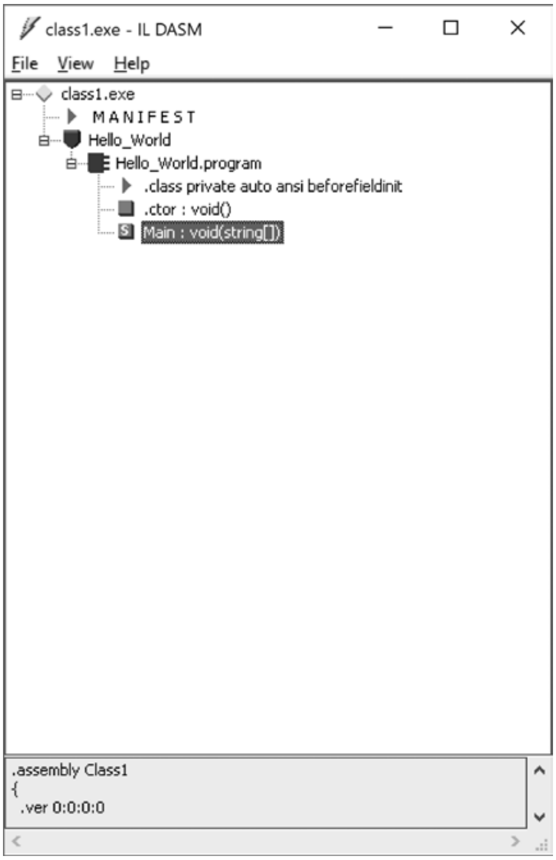
The Microsoft C# compiler is a good example of a .NET language that works in this fashion. Compiling a simple C# program will produce a Microsoft *.exe* file that you can examine with `dumpbin`. Unfortunately, you can't use `dumpbin` to look at the object code (IL or otherwise). Fortunately, Microsoft supplies a utility, *ildasm.exe*, that you can use to disassemble the IL byte/assembly code.

Consider the following small C# example program (*Class1.cs*, a slight tweak of the ubiquitous "Hello World!" program):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Hello_World
{
    class program
    {
        static void Main( string[] args)
        {
            int i = 5;
            int j = 6;
            int k = i + j;
            Console.WriteLine("Hello World! k={0}", k);
        }
    }
}
```

Typing `ildasm class1.exe` from a command prompt brings up the window shown in Figure 5-3.



```
.assembly Class1
{
  .ver 0:0:0:0
```

Figure 5-3: IL disassembler window

To view the code disassembly, double-click the S icon (next to the Main entry). This opens a window containing the following text (comments added for clarity):

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       25 (0x19)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
; push constant 5 on stack

  IL_0000:  ldc.i4.5

; pop stack and store into i

  IL_0001:  stloc.0
; push constant 6 on stack

  IL_0002:  ldc.i4.6

; pop stack and store in j

  IL_0003:  stloc.1

; Push i and j onto stack:

  IL_0004:  ldloc.0
  IL_0005:  ldloc.1

; Add two items on stack, leave result on stack

  IL_0006:  add

; Store sum into k

  IL_0007:  stloc.2

; Load string onto stack (pointer to string)

  IL_0008:  ldstr      "Hello World! k={0}"

; Push k's value onto stack:

  IL_000d:  ldloc.2
  IL_000e:  box        [mscorlib]System.Int32

; call writeline routine:

  IL_0013:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object)
  IL_0018:  ret
} // end of method program::Main
```

You can use the IL disassembler program for any .NET language (such as Visual Basic and F#). See Appendix E online for details on Microsoft's IL assembly language.

5.7 Using a Debugger to Analyze Compiler Output

Another option you can use to analyze compiler output is a debugger program, which usually incorporates a disassembler that you can use to view machine instructions. Depending on the debugger you use, viewing your compiler output this way can be either a headache or a breeze. Typically, if you use a stand-alone debugger, you'll find that it takes considerably more effort to analyze your compiler output than if you use a debugger built into a compiler's IDE. This section looks at both approaches.

5.7.1 Using an IDE's Debugger

The Microsoft Visual C++ environment provides excellent tools for analyzing the code produced by a compilation (of course, the compiler also produces assembly output, but we'll ignore that fact here). To view the output using the Visual Studio debugger, first compile your C/C++ program to an executable file and then select **Debug ▸ Step Into** from the Visual Studio Debug menu. When the program pauses execution, select **Debug ▸ Windows ▸ Disassembly** from the debug menu. For the *t1.c* program (see "Example Assembly Language Output" on page 102), you should see a disassembly like the following (assuming you're producing 32-bit code):

```
--- c:\users\rhyde\test\t\t\t.cpp --------------------------------------------
#include "stdafx.h"
#include <stdio.h>
int main(int argc, char **argv)
{
00F61000  push        ebp
00F61001  mov         ebp,esp
00F61003  sub         esp,8
    int i;
    int j;

    i = argc;
00F61006  mov         eax,dword ptr [argc]
00F61009  mov         dword ptr [i],eax
    j = **argv;
00F6100C  mov         ecx,dword ptr [argv]
00F6100F  mov         edx,dword ptr [ecx]
00F61011  movsx       eax,byte ptr [edx]
00F61014  mov         dword ptr [j],eax


    if (i == 2)
00F61017  cmp         dword ptr [i],2
00F6101B  jne         main+28h (0F61028h)
    {
        ++j;
00F6101D  mov         ecx,dword ptr [j]
00F61020  add         ecx,1
00F61023  mov         dword ptr [j],ecx
    }
    else
00F61026  jmp         main+31h (0F61031h)
    {
        --j;
00F61028  mov         edx,dword ptr [j]
00F6102B  sub         edx,1
00F6102E  mov         dword ptr [j],edx
    }

    printf("i=%d, j=%d\n", i, j);
00F61031  mov         eax,dword ptr [j]
00F61034  push        eax
00F61035  mov         ecx,dword ptr [i]
00F61038  push        ecx
00F61039  push        0F620F8h
00F6103E  call        printf (0F61090h)
00F61043  add         esp,0Ch
    return 0;
00F61046  xor         eax,eax
}
00F61048  mov         esp,ebp
00F6104A  pop         ebp
00F6104B  ret
```

Of course, because Microsoft's Visual C++ package is already capable of outputting an assembly language file during compilation, using the Visual Studio integrated debugger in this manner isn't necessary.[3] However, some compilers do not provide assembly output, and debugger output may be the easiest way to view the machine code the compiler produces. For example, Embarcadero's Delphi compiler does not provide an option to produce assembly language output. Given the massive amount of class library code that Delphi links into an application, attempting to view the code for a small section of your program by using a disassembler would be like trying to find a needle in a haystack. A better solution is to use the debugger built into the Delphi environment.

5.7.2 Using a Stand-Alone Debugger

If your compiler doesn't provide its own debugger as part of an IDE, another alternative is to use a separate debugger such as OllyDbg, DDD, or GDB to disassemble your compiler's output. Simply load the executable file into the debugger for normal debugging operations.

Most debuggers that are not associated with a particular programming language are machine-level debuggers that disassemble the binary machine code into machine instructions for viewing during the debugging operation. One problem with using machine-level debuggers is that locating a particular section of code to disassemble can be difficult. Remember, when you load the entire executable file into a debugger, you load in all the statically linked library routines and other runtime support code that doesn't normally appear in the application's source file. Searching through all this extraneous code to find out how the compiler translates a particular sequence of statements to machine code can be time-consuming. Some serious code sleuthing may be necessary. Fortunately, most linkers collect all the library routines together and place them either at the beginning or end of the executable file. Therefore, that's generally also where you'll find the code associated with your application.

Debuggers come in one of three different flavors: pure machine-level debuggers, symbolic debuggers, and source-level debuggers. Symbolic

3. The Visual C++ debugger output injects the C/C++ source code into the disassembly output, an advantage over the assembly output produced by the compiler.

debuggers and source-level debuggers require executable files to contain special debugging information and, therefore, the compiler must specifically include this extra information.

Pure machine-level debuggers have no access to the original source code or symbols in the application. A pure machine-level debugger simply disassembles the application's machine code and displays the listing using literal numeric constants and machine addresses. Reading through such code is difficult, but if you understand how compilers generate code for the HLL statements (as this book will teach you), then locating the machine code is easier. Nevertheless, without any symbolic information to provide a "root point" in the code, analysis can be difficult.

Symbolic debuggers use special symbol table information found in the executable file (or a separate debugging file, in some instances) to associate labels with functions and, possibly, variable names in your source file. This feature makes locating sections of code within the disassembly listing much easier. When symbolic labels identify calls to functions, it's much easier to see the correspondence between the disassembled code and your original HLL source code. One thing to keep in mind, however, is that symbolic information is available only if the application was compiled with debugging mode enabled. Check your compiler's documentation to determine how to activate this feature for use with your debugger.

Source-level debuggers actually display the original source code associated with the file the debugger is processing. In order to see the machine code the compiler produced, you often have to activate a special machine-level view of the program. As with symbolic debuggers, your compiler must produce special executable files (or auxiliary files) containing debug information that a source-level debugger can use. Clearly, source-level debuggers are much easier to work with because they show the correspondence between the original HLL source code and the disassembled machine code.

5.8 Comparing Output from Two Compilations

If you are an expert assembly language programmer and you're well versed in compiler design, it should be pretty easy for you to determine what changes you'll need to make to your HLL source code to improve the quality of the output machine code. However, most programmers (especially those who do not have considerable experience studying compiler output) can't just read a compiler's assembly language output. They have to compare the two sets of outputs (before and after a change) to determine which code is better. After all, not every change you make to your HLL source files will result in better code. Some changes will leave the machine code unaffected (in which case, you should use the more readable and maintainable version of the HLL source code). In other cases, you could actually make the output machine code worse. Therefore, unless you know exactly what a compiler is going to do when you make changes to your HLL source file, you should do a before-and-after comparison of the compiler's output machine code before accepting any modifications you make.

5.8.1 Before-and-After Comparisons with diff

Of course, the first reaction from any experienced software developer is, "Well, if we have to compare files, we'll just use `diff`!" As it turns out, a typical `diff` (compute file differences) program will be useful for certain purposes, but it won't be universally applicable when you're comparing two different output files from a compiler. The problem with a program like `diff` is that it works great when there are only a few differences between two files, but it's not so useful when the files are wildly different. For example, consider the following C program (*t.c*) and two different outputs produced by the Microsoft VC++ compiler:

```
extern void f( void );
int main( int argc, char **argv )
{
    int boolResult;

    switch( argc )
    {
        case 1:
            f();
            break;

        case 10:
            f();
            break;

        case 100:
            f();
            break;

        case 1000:
            f();
            break;

        case 10000:
            f();
            break;

        case 100000:
            f();
            break;

        case 1000000:
            f();
            break;

        case 10000000:
            f();
            break;

        case 100000000:
            f();
            break;
            case 1000000000:
                f();
                break;

        }
        return 0;
}
```

Here's the assembly language output MSVC++ produces when using the command line `cl /Fa t.c` (that is, when compiling without optimization):

```
  ; Listing generated by Microsoft (R) Optimizing Compiler Version 19.00.24234.1

include listing.inc

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

PUBLIC  main
EXTRN   f:PROC
pdata   SEGMENT
$pdata$main DD  imagerel $LN16
        DD      imagerel $LN16+201
        DD      imagerel $unwind$main
pdata   ENDS
xdata   SEGMENT
$unwind$main DD 010d01H
        DD      0620dH
xdata   ENDS
; Function compile flags: /Odtp
_TEXT   SEGMENT
tv64 = 32
argc$ = 64
argv$ = 72
main    PROC
; File c:\users\rhyde\test\t\t\t.cpp
; Line 4
$LN16:
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 56                                 ; 00000038H
; Line 7
        mov     eax, DWORD PTR argc$[rsp]
        mov     DWORD PTR tv64[rsp], eax
        cmp     DWORD PTR tv64[rsp], 100000             ; 000186a0H
        jg      SHORT $LN15@main
        cmp     DWORD PTR tv64[rsp], 100000             ; 000186a0H
        je      SHORT $LN9@main
        cmp     DWORD PTR tv64[rsp], 1
        je      SHORT $LN4@main
        cmp     DWORD PTR tv64[rsp], 10
        je      SHORT $LN5@main
        cmp     DWORD PTR tv64[rsp], 100                ; 00000064H
        je      SHORT $LN6@main
        cmp     DWORD PTR tv64[rsp], 1000                ; 000003e8H
        je      SHORT $LN7@main
        cmp     DWORD PTR tv64[rsp], 10000               ; 00002710H
        je      SHORT $LN8@main
        jmp     SHORT $LN2@main
$LN15@main:
        cmp     DWORD PTR tv64[rsp], 1000000             ; 000f4240H
        je      SHORT $LN10@main
        cmp     DWORD PTR tv64[rsp], 10000000            ; 00989680H
        je      SHORT $LN11@main
        cmp     DWORD PTR tv64[rsp], 100000000           ; 05f5e100H
        je      SHORT $LN12@main
        cmp     DWORD PTR tv64[rsp], 1000000000          ; 3b9aca00H
        je      SHORT $LN13@main
        jmp     SHORT $LN2@main
$LN4@main:
; Line 10
        call    f
; Line 11
        jmp     SHORT $LN2@main
$LN5@main:
; Line 14
        call    f
; Line 15
        jmp     SHORT $LN2@main
$LN6@main:
; Line 18
        call    f
; Line 19
        jmp     SHORT $LN2@main
$LN7@main:
; Line 22
        call    f
; Line 23
        jmp     SHORT $LN2@main
$LN8@main:
; Line 26
        call    f
; Line 27
        jmp     SHORT $LN2@main
$LN9@main:
; Line 30
        call    f
; Line 31
        jmp     SHORT $LN2@main
$LN10@main:
; Line 34
        call    f
; Line 35
        jmp     SHORT $LN2@main
$LN11@main:
; Line 38
        call    f
; Line 39
        jmp     SHORT $LN2@main
```

```
$LN12@main:
; Line 42
        call    f
; Line 43
        jmp     SHORT $LN2@main
$LN13@main:
; Line 46
        call    f
$LN2@main:
; Line 50
        xor     eax, eax
; Line 51
        add     rsp, 56                                 ; 00000038H
        ret     0
main    ENDP
_TEXT   ENDS
END
```

Here's the assembly listing we get when we compile the C program with the command line `cl /Ox /Fa t.c` (`/Ox` enables maximum optimization for speed in Visual C++):

```
    ; Listing generated by Microsoft (R) Optimizing Compiler Version 19.00.24234.1

include listing.inc

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

PUBLIC  main
EXTRN   f:PROC
pdata   SEGMENT
$pdata$main DD  imagerel $LN18
        DD      imagerel $LN18+89
        DD      imagerel $unwind$main
pdata   ENDS
xdata   SEGMENT
$unwind$main DD 010401H
        DD      04204H
xdata   ENDS
; Function compile flags: /Ogtpy
_TEXT   SEGMENT
argc$ = 48
argv$ = 56
main    PROC
; File c:\users\rhyde\test\t\t\t.cpp
; Line 4
$LN18:
        sub     rsp, 40                                 ; 00000028H
; Line 7
        cmp     ecx, 100000                             ; 000186a0H
        jg      SHORT $LN15@main
        je      SHORT $LN10@main
        sub     ecx, 1
        je      SHORT $LN10@main
        sub     ecx, 9
        je      SHORT $LN10@main
        sub     ecx, 90                         ; 0000005aH
        je      SHORT $LN10@main
        sub     ecx, 900                        ; 00000384H
        je      SHORT $LN10@main
        cmp     ecx, 9000                       ; 00002328H
; Line 27
        jmp     SHORT $LN16@main
$LN15@main:
; Line 7
        cmp     ecx, 1000000                    ; 000f4240H
        je      SHORT $LN10@main
        cmp     ecx, 10000000                   ; 00989680H
        je      SHORT $LN10@main
        cmp     ecx, 100000000                  ; 05f5e100H
        je      SHORT $LN10@main
        cmp     ecx, 1000000000                 ; 3b9aca00H
$LN16@main:
        jne     SHORT $LN2@main
$LN10@main:
; Line 34
        call    f
$LN2@main:
; Line 50
        xor     eax, eax
; Line 51
        add     rsp, 40                         ; 00000028H
        ret     0
main    ENDP
_TEXT   ENDS
        END
```

It doesn't take a very sharp eye to notice that the two assembly language output files are radically different. Running these two files through `diff` simply produces a lot of noise; the output from `diff` is more difficult to interpret than manually comparing the two assembly language output files.

A differencing program like `diff` (or better yet, the differencing facility built into many advanced programming editors) works best for comparing two different outputs for a given HLL source file to which you've made a small change. In the current example, had we changed the statement `case 1000:` to `case 1001:`, then a `diff` of the resulting assembly file against the original produces the following output:

```
50c50
< cmp eax, 1000

---
> cmp eax, 1001
```

As long as you're comfortable reading `diff` output, this isn't too bad. However, a better solution is to use some commercially available file comparison programs. Two excellent options are Beyond Compare (*https://www.scootersoftware.com/*) and Araxis Merge (*https://www.araxis.com/merge/*).

Of course, another way to compare compiler output is manually. Set two listings side by side (either on paper or on your monitor) and start analyzing them. In the current C example, if we compare the two different outputs from the C compiler (without optimization and with the `/Ox` optimization option), we'll discover that both versions use a binary search algorithm to compare the `switch` value against a list of widely varying constants. The main difference between the optimized and unoptimized versions has to do with code duplication.

In order to properly compare two assembly listings that a compiler produces, you'll need to learn how to interpret the machine language output from your compilers and connect certain assembly language sequences with the statements in your HLL code. That's the purpose of many of the chapters to come.

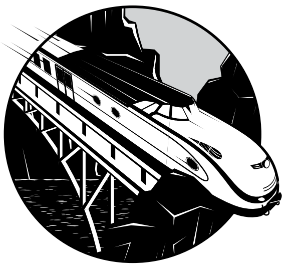
5.9 For More Information

Your compiler's manual is the first place to look when you're trying to figure out how to view the machine code the compiler produces. Many compilers produce assembly language output as an option, and that's the best way to view code output. If your compiler does not provide this option, a debugging tool built into the compiler's IDE (if available) is another good choice. See the documentation for your IDE or compiler for details.

Tools like `objdump` and `dumpbin` are also useful for examining compiler output. Check the Microsoft, FSF/GNU, or Apple LLVM documentation for details on using these programs. If you decide to use an external debugger, such as OllyDbg or GDB, check out the software's user documentation, or visit the author's support web page (for example, *http://www.ollydbg.de/* for the OllyDbg debugger).

6

CONSTANTS AND HIGH-LEVEL LANGUAGES



Some programmers may not realize it, but many CPUs do not treat constant and variable data identically at the machine code level. Most CPUs provide a special *immediate addressing mode* that allows a language translator to embed a constant value directly into a machine instruction rather than storing it in a memory location and accessing it as a variable. However, the CPU's ability to represent constant data efficiently varies by CPU and, in fact, by the type of the data. By understanding how a CPU treats constant data at the machine code level, you can choose appropriate ways to represent constants in your HLL source code to produce smaller and faster executable programs. To that end, this chapter discusses the following topics:

- How to use literal constants properly to improve the efficiency of your programs
- The difference between a literal constant and a manifest constant
- How compilers process compile-time constant expressions to reduce program size and avoid runtime calculations

- The difference between a compile-time constant and read-only data kept in memory
- How compilers represent noninteger constants, such as enumerated data types, Boolean data types, floating-point constants, and string constants
- How compilers represent composite data type constants, such as array constants and record/struct constants

By the time you finish this chapter, you should have a clear understanding of how various constants can affect the efficiency of the machine code your compiler produces.

NOTE *If you've already read* WGC1*, you may just want to skim through this chapter, which for the sake of completeness repeats some of the information from Chapters 6 and 7 of that volume.*

6.1 Literal Constants and Program Efficiency

High-level programming languages and most modern CPUs allow you to specify constant values just about anywhere you can legally read the value of a memory variable. Consider the following Visual Basic and HLA statements, which assign the constant `1000` to the variable `i`:

```
i = 1000

mov( 1000, i );
```

The 80x86, like most CPUs, actually encodes the constant representation for 1,000 directly into the machine instruction. This provides a compact and efficient way to work with constants at the machine level. Therefore, statements that use literal constants in this manner are often more efficient that those that assign constant values to a variable and then reference that variable later in the code. Consider the following Visual Basic code sequence:

```
oneThousand = 1000
    .
    .
    .
x = x + oneThousand 'Using "oneThousand" rather than
                    ' a literal constant.
y = y + 1000        'Using a literal constant.
```

Now consider the 80x86 assembly code you would probably write for these last two statements. For the first statement, we must use two instructions because we can't add the value of one memory location directly to another:

```
mov( oneThousand, eax ); // x = x + oneThousand
add( eax, x );
```

But we can add a constant to a memory location, so the second Visual Basic statement translates to a single machine instruction:

```
add( 1000, y ); // y = y + 1000
```

As you can see, using a literal constant, rather than a variable, is more efficient. This is not to suggest, however, that every processor operates more efficiently using literal constants, or that every CPU operates more efficiently no matter the value of the constant. Some very old CPUs don't provide the ability to embed literal constants within a machine instruction at all; and many RISC processors, such as the ARM, do so only for smaller 8-, 12-, or 16-bit constants.[1] Even those CPUs that allow you to load any integer constant may not support literal floating-point constants—the ubiquitous 80x86 processor being one example. Few CPUs provide the ability to encode large data structures (such as an array, record, or string) as part of a machine instruction. For example, consider the following C code:

```
#include <stdlib.h>
#include <stdio.h>
int main( int argc, char **argv, char **envp )
{
  int i,j,k;

  i = 1;
  j = 16000;
  k = 100000;
  printf( "%d, %d, %d\n", i, j, k );

}
```

Its compilation to PowerPC assembly by the GCC compiler looks like this (edited to remove the nonrelevant code):

```
L1$pb:
    mflr r31
    stw r3,120(r30)
    stw r4,124(r30)
    stw r5,128(r30)

; The following two instructions copy the value 1 into the variable "i"

    li r0,1
    stw r0,64(r30)

; The following two instructions copy the value 16,000 into the variable "j"

    li r0,16000
    stw r0,68(r30)
```

1. Even the 80x86 limits immediate constants to 32 bits.

```
; It takes three instructions to copy the value 100,000 into variable "k"

    lis r0,0x1
    ori r0,r0,34464
    stw r0,72(r30)

; The following code sets up and calls the printf function:

    addis r3,r31,ha16(LC0-L1$pb)
    la r3,lo16(LC0-L1$pb)(r3)
    lwz r4,64(r30)
    lwz r5,68(r30)
    lwz r6,72(r30)
    bl L_printf$stub
    mr r3,r0
    lwz r1,0(r1)
    lwz r0,8(r1)
    mtlr r0
    lmw r30,-8(r1)
    blr
```

The PowerPC CPU allows only 16-bit immediate constants in a single instruction. In order to load a larger value into a register, the program has to first use the `lis` instruction to load the higher-order (HO) 16 bits of a 32-bit register and then use the `ori` instruction to merge in the lower-order (LO) 16 bits. The exact operation of these instructions isn't too important. What's notable is that the compiler emits three instructions for large constants, and only two for smaller constants. Therefore, using 16-bit constant values on the PowerPC produces shorter and faster machine code.

The compilation of this C code to ARMv7 assembly by the GCC compiler looks like this (edited to remove the nonrelevant code):

```
.LC0:
    .ascii  "i=%d, j=%d, k=%d\012\000"
    .text
    .align  2
    .global main
    .type   main, %function
main:
    @ args = 0, pretend = 0, frame = 24
    @ frame_needed = 1, uses_anonymous_args = 0
    stmfd   sp!, {fp, lr}
    add fp, sp, #4
    sub sp, sp, #24
    str r0, [fp, #-24]
    str r1, [fp, #-28]

; Store 1 into 'i' variable:

    mov r3, #1
    str r3, [fp, #-8]
```

```
@ Store 16000 into 'j' variable:

    mov r3, #16000
    str r3, [fp, #-12]

@ Store 100,000 (constant appears in memory) into 'k' variable:

    ldr r3, .L3
    str r3, [fp, #-16]

@ Fetch the values and print them:

    ldr r0, .L3+4
    ldr r1, [fp, #-8]
    ldr r2, [fp, #-12]
    ldr r3, [fp, #-16]
    bl  printf
    mov r3, #0
    mov r0, r3
    sub sp, fp, #4
    @ sp needed
    ldmfd   sp!, {fp, pc}
.L4:

@ constant value for k appears in memory:

    .align  2
.L3:
    .word    100000
    .word   .LC0
```

The ARM CPU allows only 16-bit immediate constants in a single instruction. In order to load a larger value into a register, the compiler places the constant into a memory location and loads the constant from memory.

Even though CISC processors like the 80x86 can usually encode any integer constant (up to 32 bits) in a single instruction, this doesn't mean that the program's efficiency is independent of the sizes of the constants you use in your programs. CISC processors often use different encodings for machine instructions that have large or small immediate operands, allowing the program to use less memory for smaller constants. For example, consider the following two 80x86/HLA machine instructions:

```
add( 5, ebx );
add( 500_000, ebx );
```

On the 80x86 an assembler can encode the first instruction in 3 bytes: 2 bytes for the opcode and addressing mode information, and 1 byte to hold the small immediate constant `5`. The second instruction, on the other hand, requires 6 bytes to encode: 2 bytes for the opcode and addressing mode information, and 4 bytes to hold the constant `500_000`. Certainly the second instruction is larger, and in some cases it may even run a little slower.

6.2 Binding Times

What exactly is a constant? Obviously, from an HLL perspective, a constant is some sort of entity whose value doesn't change (that is, remains constant). However, there is more to the definition. For example, consider the following Pascal constant declaration:

```
const someConstant:integer = 5;
```

In the code following this declaration,[2] you can use the name `someConstant` in place of the value `5`. But what about before this declaration? How about outside the scope to which this declaration belongs? Clearly the value of `someConstant` can change upon the compiler processing this declaration. So the notion that a constant's "value doesn't change" doesn't exactly apply here.

The real concern here isn't *where* the program associates a value with `someConstant` but *when*. *Binding* is the technical name for creating associations between attributes (such as the name, value, and scope) of some object. For example, the earlier Pascal example binds the value `5` to the name `someConstant`. *Binding time*—when the binding (association) occurs—can happen at several different points:

- *At language definition time.* This refers to when the language designer(s) define the language. The constants `true` and `false` in many languages are good examples.
- *During compilation.* The Pascal `someConstant` declaration in this section is a good example.
- *During the linking phase.* An example of this might be a constant that specifies the size of the object code (machine instructions) in a program. The program cannot compute this size any earlier than during the link phase, when the linker pulls in all the object code modules and combines them together.
- *During program loading (into memory).* A good example of load time binding would be associating the address of an object in memory (such as a variable or machine instruction) with some pointer constant. On many systems, the operating system relocates the code when it loads it into memory, so the program can only determine absolute memory addresses after loading.
- *During program execution.* Some bindings can occur only while the program is running. For example, when you assign the value of some (computed) arithmetic expression to a variable, the binding of the value to the variable occurs during execution.

Dynamic bindings are those that occur during program execution. *Static bindings* are those that occur at any other time. Chapter 7 will take another look at binding (see "What Is a Variable?" on page 180).

2.Specifically, in the code within the scope of this declaration.

6.3 Literal Constants vs. Manifest Constants

A *manifest constant* is a constant value associated with—that is, bound to—a symbolic name. A language translator can directly substitute the value everywhere the name appears within the source code, producing easy-to-read and easily maintained programs. The proper use of manifest constants is a good indication of professionally written code.

Declaring manifest constants is simple in many programming languages:

- Pascal programmers use the const section.
- HLA programmers can use the const or the val declaration sections.
- C/C++ programmers can use the #define macro facility.

This Pascal code fragment demonstrates an appropriate use of manifest constants in a program:

```
const
    maxIndex = 9;

var
    a :array[0..maxIndex] of integer;
        .
        .
        .
    for i := 0 to maxIndex do
        a[i] := 0;
```

This code is much easier to read and maintain than code that uses literal constants. By changing a single statement in this program (the maxIndex constant declaration) and recompiling the source file, you can easily set the number of elements and the program will continue to function properly.

Because the compiler substitutes the literal numeric constant in place of the symbolic name for the manifest constant, there is no performance penalty for using manifest constants. Given that they improve the readability of your programs without any loss in efficiency, manifest constants are an important component of great code. Use them.

6.4 Constant Expressions

Many compilers support the use of *constant expressions*, which are expressions that can be evaluated during compilation. The component values of a constant expression are all known at compile time, so the compiler can evaluate the expression and substitute its value during compilation rather than computing it at runtime. As with manifest constants, constant expressions enable you to write more easily readable and maintainable code, without any runtime efficiency loss.

For example, consider the following C code:

```
#define smArraySize 128
#define bigArraySize (smArraySize*8)
      .
      .
      .
char name[ smArraySize ];
int  values[ bigArraySize ];
```

These two array declarations expand to the following:

```
char name[ 128 ];
int  values[ (smArraySize * 8) ];
```

The C preprocessor further expands this to:

```
char name[ 128 ];
int  values[ (128 * 8) ];
```

Although the C language definition supports constant expressions, this feature is not available in every language, so you'll need to check the language reference manual for your particular compiler. The Pascal language definition, for example, says nothing about constant expressions. Some Pascal implementations support them, but others do not.

Modern optimizing compilers are capable of computing constant subexpressions within arithmetic expressions at compile time (known as *constant folding*; see "Common Compiler Optimizations" on page 63), thereby saving the expense of computing fixed values at runtime. Consider the following Pascal code:

```
var
    i   :integer;
            .
            .
            .
    i := j + (5*2-3);
```

Any decent Pascal implementation will recognize that the subexpression `5*2-3` is a constant expression, compute the value for this expression (`7`) during compilation, and substitute that value at compile time. In other words, a good Pascal compiler generally emits machine code that is equivalent to the following statement:

```
i := j + 7;
```

If your particular compiler fully supports constant expressions, you can use this feature to write better source code. It may seem paradoxical, but writing out a full expression at some point in your program can sometimes make that particular piece of code easier to read and understand; someone reading your code can see exactly how you calculated a value, rather than having to figure out how you arrived at some "magic" number. For example, in the context of an invoicing or timesheet routine, the expression 5*2-3 might describe the computation "two persons working for five hours, minus three person-hours provided for the job" better than the literal constant 7.

The following sample C code, and the PowerPC output produced by the GCC compiler, demonstrates constant expression optimization in action:

```
#include <stdio.h>
int main( int argc, char **argv, char **envp )
{
  int j;

  j = argc+2*5+1;
  printf( "%d %d\n", j, argc );

}
```

Here's the GCC output (PowerPC assembly language):

```
_main:
    mflr r0
    mr r4,r3            // Register r3 holds the ARGC value upon entry
    bcl 20,31,L1$pb
L1$pb:
    mr r5,r4            // R5 now contains the ARGC value.
    mflr r10
    addi r4,r4,11       // R4 contains argc+ 2*5+1
                        // (i.e., argc+11)
    mtlr r0             // Code that calls the printf function.
    addis r3,r10,ha16(LC0-L1$pb)
    la r3,lo16(LC0-L1$pb)(r3)
    b L_printf$stub
```

As you can see, GCC has replaced the constant expression 2*5+1 with the constant 11.

Making your code more readable is definitely a good thing to do and a major part of writing great code. Keep in mind, however, that some compilers may not support the use of constant expressions, instead emitting code to compute the constant value at runtime. Obviously, this will affect the size and execution speed of your resulting program. Knowing what your compiler can do will help you decide whether to use constant expressions or precompute expressions to increase efficiency at the cost of readability.

6.5 Manifest Constants vs. Read-Only Memory Objects

C/C++ programmers may have noticed that the previous section did not discuss the use of the C/C++ const declaration. This is because symbolic names (hereafter *symbols*) you declare in a C/C++ const statement aren't necessarily manifest constants. That is, C/C++ does not always substitute the value for a symbol wherever it appears in a source file. Instead, a C/C++ compiler might store that const value in memory and then reference the const object as it would a static (read-only) variable. The only difference, then, between that const object and a static variable is that the C/C++ compiler doesn't allow you to assign a value to const at runtime.

C/C++ sometimes treats constants you declare in const statements as static variables for a very good reason—it allows you to create within a function local constants whose value can change each time that function executes (although while the function is executing, the value remains fixed). This is why you can't always use such a "constant" within a const in C/C++ and expect the C/C++ compiler to precompute its value.

Most C++ compilers will accept this:

```
const int arraySize = 128;
      .
      .
      .
int anArray[ arraySize ];
```

They will not, however, accept this sequence:

```
const int arraySizes[2] = {128,256}; // This is legal
const int arraySize = arraySizes[0]; // This is also legal

int array[ arraySize ]; // This is not legal
```

arraySize and arraySizes are both constants. Yet the C++ compiler won't allow you to use the arraySizes constant, or anything based on it, as an array bound. This is because arraySizes[0] is actually a runtime memory location and, therefore, arraySize must also be a runtime memory location. In theory, you'd think the compiler would be smart enough to figure out that arraySize is computable at compile time and just substitute that value (128). The C++ language, however, doesn't allow this.

6.6 Swift let Statements

In the Swift programming language, you can create constants using the let statement. For example:

```
let someConstant = 5
```

However, the value is bound to the constant's name at runtime (that is, this is a dynamic binding). The expression on the right-hand side of the assignment operator (=) doesn't have to be a constant expression; it can be an arbitrary expression involving variables and other nonconstant components. Every time the program executes this statement (such as in a loop), the program could bind a different value to *someConstant*.

The Swift let statement doesn't truly define constants in the traditional sense; rather, it lets you create "write-once" variables. In other words, within the scope of the symbol you define using the let statement, you can initialize the name with a value only once. Note that if you leave and re-enter the name's scope, the value is destroyed (upon exiting the scope) and you can bind a new (possibly different) value to the name upon re-entering the scope. Unlike, say, the const int declaration in C++, let statements do not allow you to allocate storage for the object in read-only memory.

6.7 Enumerated Types

Well-written programs often use a set of names to represent real-world quantities that don't have an explicit numeric representation. An example of such a set of names might be various display technologies, like crt, lcd, led, and plasma. Even though the real world doesn't associate numeric values with these concepts, you must encode the values numerically if you're going to efficiently represent them in a computer system. The internal representation for each symbol is generally arbitrary, as long as the value we assign is unique. Many computer languages provide an *enumerated data type* that automatically associates a unique value with each name in a list. By using enumerated data types in your programs, you can assign meaningful names to your data rather than using "magic" numbers such as 0, 1, 2, and so on.

For example, in early versions of the C language, you would create a sequence of identifiers, each with a unique value, as follows:

```
/*
   Define a set of symbols representing the
   different display technologies
*/

#define crt 0
#define lcd (crt+1)
#define led (lcd+1)
#define plasma (led+1)
```

By assigning values that are consecutive, you ensure that each is unique. Another advantage to this approach is that it orders the values. That is, crt < lcd < led < plasma. Unfortunately, creating manifest constants this way is laborious and error-prone.

Fortunately, in most languages enumerated constants can solve this problem. To "enumerate" means to number, and this is exactly what the compiler does—it numbers each constant, thereby handling the bookkeeping details of assigning values to enumerated constants.

Most modern programming languages provide support for declaring enumerated types and constants. Here are some examples from C/C++, Pascal, Swift, and HLA:

```
enum displays {crt, lcd, led, plasma, oled };       // C++
type displays = (crt, lcd, led, plasma, oled );     // Pascal
type displays :enum{crt, lcd, led, plasma, oled };  // HLA
```

```
// Swift example:
enum Displays
{
    case crt
    case lcd
    case led
    case plasma
    case oled
}
```

These four examples internally associate 0 with `crt`, 1 with `lcd`, 2 with `led`, 3 with `plasma`, and 4 with `oled`. Again, the exact internal representation is irrelevant (as long as each value is unique) because the value's only purpose is to differentiate the enumerated objects.

Most languages assign *monotonically increasing* values (that is, each successive value is greater than all previous values) to symbols in an enumerated list. Therefore, these examples have the following relationships:

```
crt < lcd < led < plasma < oled
```

Don't let this give you the impression that all enumerated constants appearing in a single program have a unique internal representation, though. Most compilers assign a value of `0` to the first item in an enumeration list you create, a value of `1` to the second, and so on. For example, consider the following Pascal type declarations:

```
type
    colors = (red, green, blue);
    fasteners = (bolt, nut, screw, rivet );
```

Most Pascal compilers would use the value `0` as the internal representation for both `red` and `bolt`; `1` for `green` and `nut`; and so on. In languages (like Pascal and Swift) that enforce type checking, you generally can't use symbols of type `colors` and `fasteners` in the same expression. Therefore, the fact that these symbols share the same internal representation isn't an issue because the compiler's type-checking facilities preclude any possible confusion. Some languages, like C/C++ and assembly, do not provide strong type checking, however, and so this kind of confusion can occur. In those languages, it is the programmer's responsibility to avoid mixing different types of enumeration constants.

Most compilers allocate the smallest unit of memory the CPU can efficiently access in order to represent an enumerated type. Because most enumerated type declarations define fewer than 256 symbols, compilers on machines that can efficiently access byte data will usually allocate a byte for any variable with an enumerated data type. Compilers on many RISC machines can allocate a 32-bit word (or more) simply because it's faster to access such blocks of data. The exact representation is language and compiler/implementation dependent, so check your compiler's reference manual for the details.

6.8 Boolean Constants

Many high-level programming languages provide *Boolean*, or *logical*, constants to represent the values `true` and `false`. Because there are only two possible Boolean values, their representation requires only a single bit. However, because most CPUs do not allow you to allocate a single bit of storage, most programming languages use a whole byte or even a larger object to represent a Boolean value. What happens to any leftover bits in a Boolean object? Unfortunately, the answer varies by language.

Many languages treat the Boolean data type as an enumerated type. For example, in Pascal, the Boolean type is defined this way:

```
type
    boolean = (false, true);
```

This declaration associates the internal value `0` with `false` and `1` with `true`. This association has a couple of desirable attributes:

- Most of the Boolean functions and operators behave as expected—for example, (`true and true`) = `true`, (`true and false`) = `false`, and so on.
- When you compare the two values, `false` is less than `true`—an intuitive result.

Unfortunately, associating `0` with `false` and `1` with `true` isn't always the best solution. Here are some reasons why:

- Certain Boolean operations, applied to a bit string, do not produce expected results. For example, you might expect (not `false`) to be equal to `true`. However, if you store a Boolean variable in an 8-bit object, then (not `false`) is equal to `$FF`, which is not equal to `true (1)`.
- Many CPUs provide instructions that easily test for `0` or nonzero after an operation; few CPUs provide an implicit test for `1`.

Many languages, such as C, C++, C#, and Java, treat `0` as `false` and anything else as `true`. This has a couple of advantages:

- CPUs that provide easy checks for `0` and nonzero can easily test a Boolean result.
- The `0`/nonzero representation is valid regardless of the size of the object holding a Boolean variable.

Unfortunately, this scheme also has some drawbacks:

- Many bitwise logical operations produce incorrect results when applied to `0` and nonzero Boolean values. For example `$A5` (`true`/nonzero) AND `$5A` (`true`/nonzero) is equal to `0` (`false`). Logically ANDing `true` and `true` should not produce `false`. Similarly, (NOT `$A5`) produces `$5A`. Generally, you'd expect (NOT `true`) to produce `false` rather than `true` (`$5A`).

- When a bit string is treated as a two's-complement signed-integer value, it's possible for certain values of true to be less than zero (for example, the 8-bit value $FF is equivalent to -1). So, in some cases, the intuitive result that false is less than true may not be correct.

Unless you are working in assembly language (where you get to define the values for true and false), you'll have to live with whatever scheme your HLL uses to represent Boolean values, as explained in its language reference manual.

Knowing how your language represents true and false can help you write high-level source code that produces better machine code. For example, suppose you are writing C/C++ code. In these languages, false is 0 and true is anything else. Consider the following statement in C:

```
int i, j, k;
      .
      .
      .
    i = j && k;
```

The machine code produced for this assignment statement by many compilers is absolutely horrid. It often looks like the following (Visual C++ output):

```
; Line 8
        cmp     DWORD PTR j$[rsp], 0
        je      SHORT $LN3@main
        cmp     DWORD PTR k$[rsp], 0
        je      SHORT $LN3@main
        mov     DWORD PTR tv74[rsp], 1
        jmp     SHORT $LN4@main
$LN3@main:
        mov     DWORD PTR tv74[rsp], 0
$LN4@main:
        mov     eax, DWORD PTR tv74[rsp]
        mov     DWORD PTR i$[rsp], eax
;
```

Now, suppose that you always ensure that you use the values 0 for false and 1 for true (with no possibility of any other value). Under these conditions, you could write the previous statement this way:

```
i = j & k;  /* Notice the bitwise AND operator */
```

Here's the code that Visual C++ generates for the preceding statement:

```
; Line 8
        mov     eax, DWORD PTR k$[rsp]
        mov     ecx, DWORD PTR j$[rsp]
        and     ecx, eax
        mov     DWORD PTR i$[rsp], ecx
```

As you can see, this code is significantly better. Provided that you always use 1 for true and 0 for false, you can get away with using the bitwise AND (&) and OR (|) operators in place of the logical operators.[3] As noted earlier, you can't get consistent results using the bitwise NOT operator; you can, however, do the following to produce correct results for a logical NOT operation:

```
i = ~j & 1; /* "~" is C's bitwise not operator */
```

This short sequence inverts all the bits in j and then clears all bits except bit 0.

The bottom line is that you should be intimately aware of how your particular compiler represents Boolean constants. If you're given a choice (such as any nonzero value), then you can pick appropriate values for true and false to help your compiler emit better code.

6.9 Floating-Point Constants

Floating-point constants are special cases on most computer architectures. Because floating-point representations can consume a large number of bits, few CPUs provide an immediate addressing mode to load an arbitrary constant into a floating-point register. This is true even for small (32-bit) floating-point constants. It is even true on many CISC processors such as the 80x86. Therefore, compilers often have to place floating-point constants in memory and then have the program read them from memory, just as though they were variables. Consider, for example, the following C program:

```
#include <stdlib.h>
#include <stdio.h>
int main( int argc, char **argv, char **envp )
{
  static int j;
  static double i = 1.0;
  static double a[8] = {0,1,2,3,4,5,6,7};

  j = 0;
  a[j] = i+1.0;


}
```

Now consider the PowerPC code that GCC generates for this program with the -O2 option:

```
.lcomm _j.0,4,2
.data
```

3. You can't always get away with using the bitwise operators; any logic that depends on short-circuit evaluation, which the bitwise operators don't support, will have to use the standard && and || operators.

```
// This is the variable i.
// As it is a static object, GCC emits the data directly
// for the variable in memory. Note that "1072693248" is
// the HO 32-bits of the double-precision floating-point
// value 1.0, 0 is the LO 32-bits of this value (in integer
// form).

    .align 3
_i.1:
    .long       1072693248
    .long       0

// Here is the "a" array. Each pair of double words below
// holds one element of the array. The funny integer values
// are the integer (bitwise) representation of the values
// 0.0, 1.0, 2.0, 3.0, ..., 7.0.

    .align 3
_a.2:
    .long       0
    .long       0
    .long       1072693248
    .long       0
    .long       1073741824
    .long       0
    .long       1074266112
    .long       0
    .long       1074790400
    .long       0
    .long       1075052544
    .long       0
    .long       1075314688
    .long       0
    .long       1075576832
    .long       0

// The following is a memory location that GCC uses to represent
// the literal constant 1.0. Note that these 64 bits match the
// same value as a[1] in the _a.2 array. GCC uses this memory
// location whenever it needs the constant 1.0 in the program.

.literal8
    .align 3
LC0:
    .long       1072693248
    .long       0

// Here's the start of the main program:

.text
    .align 2
    .globl _main
_main:
```

```
// This code sets up the static pointer register (R10), used to
// access the static variables in this program.

    mflr r0
    bcl 20,31,L1$pb
L1$pb:
    mflr r10
    mtlr r0

    // Load floating-point register F13 with the value
    // in variable "i":

    addis r9,r10,ha16(_i.1-L1$pb)  // Point R9 at i
    li r0,0
    lfd f13,lo16(_i.1-L1$pb)(r9)   // Load F13 with i's value.

    // Load floating-point register F0 with the constant 1.0
    // (which is held in "variable" LC0:

    addis r9,r10,ha16(LC0-L1$pb)   // Load R9 with the
                                   //  address of LC0
    lfd f0,lo16(LC0-L1$pb)(r9)     // Load F0 with the value
                                   //  of LC0 (1.0).

    addis r9,r10,ha16(_j.0-L1$pb)  // Load R9 with j's address
    stw r0,lo16(_j.0-L1$pb)(r9)    // Store a zero into j.

    addis r9,r10,ha16(_a.2-L1$pb)  // Load a[j]'s address into R9

    fadd f13,f13,f0                // Compute i+1.0

    stfd f13,lo16(_a.2-L1$pb)(r9)  // Store sum into a[j]
    blr                            // Return to caller
```

Because the PowerPC processor is a RISC CPU, the code that GCC generates for this simple sequence is rather convoluted. For comparison with a CISC equivalent, consider the following HLA code for the 80x86; it is a line-by-line translation of the C code:

```
program main;
static
    j:int32;
    i:real64 := 1.0;
    a:real64[8] := [0,1,2,3,4,5,6,7];

readonly
    OnePointZero : real64 := 1.0;

begin main;

    mov( 0, j );  // j=0;

    // push i onto the floating-point stack
    fld( i );

    // push the value 1.0 onto the floating-point stack

    fld( OnePointZero );

    // pop i and 1.0, add them, push sum onto the FP stack

    fadd();

    // use j as an index

    mov( j, ebx );

    // Pop item off FP stack and store into a[j].

    fstp( a[ ebx*8 ] );

end main;
```

This code is much easier to follow than the PowerPC code (this is one advantage of CISC code over RISC code). Note that like the PowerPC, the 80x86 does not support an immediate addressing mode for most floating-point operands. Therefore, as on the PowerPC, you have to place a copy of the constant `1.0` in some memory location and access that memory location whenever you want to work with the value `1.0`.[4]

Because most modern CPUs do not support an immediate addressing mode for all floating-point constants, using such constants in your programs is equivalent to accessing variables initialized with those constants. Don't forget that accessing memory can be very slow if the locations you're referencing are not in the data cache. Accordingly, using floating-point constants can be very slow compared with accessing integer or other constant values that fit within a register.

Note that some CPUs do allow you to encode certain floating-point immediate constants as part of the instruction's opcode. The 80x86, for example, has a special "load zero" instruction that loads `0.0` onto the floating-point stack. The ARM processor also provides an instruction that allows you to load certain floating-point constants into a CPU floating-point register (see "The `vmov` Instructions" in Appendix C online).

On 32-bit processors, a CPU can often do simple 32-bit floating-point operations using integer registers and the immediate addressing mode. For example, you can easily assign a 32-bit single-precision floating-point value to a variable by loading a 32-bit integer register with the bit pattern for that

4. Actually, HLA does allow you to specify an instruction like `fld( 1.0 );`. However, this is not a real CPU instruction. HLA will simply create a constant for you in the read-only data section and load a copy of that value from memory when you execute the `fld` instruction. Also note that 0.0 and 1.0 are special cases on the x86; you can use the `fldz` (`0.0`) and `fld1` instructions to load these common immediate constants.

number and then storing the integer register into the floating-point variable. Consider the following code:

```
#include <stdlib.h>
#include <stdio.h>
int main( int argc, char **argv, char **envp )
{

  static float i;

  i = 1.0;


}
```

Here's the PowerPC code that GCC generates for this sequence:

```
.lcomm _i.0,4,2 // Allocate storage for float variable i

.text
    .align 2
    .globl _main
_main:

    // Set up the static data pointer in R10:

    mflr r0
    bcl 20,31,L1$pb
L1$pb:
    mflr r10
    mtlr r0

    // Load the address of i into R9:

    addis r9,r10,ha16(_i.0-L1$pb)

    // Load R0 with the floating-point representation of 1.0
    // (note that 1.0 is equal to 0x3f800000):

    lis r0,0x3f80 // Puts 0x3f80 in HO 16 bits, 0 in LO bits

    // Store 1.0 into variable i:

    stw r0,lo16(_i.0-L1$pb)(r9)

    // Return to whomever called this code:

    blr
```

The 80x86, being a CISC processor, makes this task trivial in assembly language. Here's the HLA code that does the same job:

```
program main;
static
    i:real32;
begin main;

    mov( $3f800_0000, i ); // i = 1.0;

end main;
```

Simple assignments of single-precision floating-point constants to floating-point variables can often use a CPU's immediate addressing mode, sparing the program the expense of accessing memory (whose data might not be in the cache). Unfortunately, compilers don't always take advantage of this trick for assigning a floating-point constant to a double-precision variable. GCC on the PowerPC or ARM, for example, reverts to keeping a copy of the constant in memory and copying that memory location's value when assigning the constant to a floating-point variable.

Most optimizing compilers are smart enough to maintain a table of constants they've created in memory. Therefore, if you reference the constant `2.0` (or any other floating-point constant) multiple times in your source file, the compiler will allocate only one memory object for that constant. Keep in mind, however, that this optimization works only within the same source file. If you reference the same constant value but in different source files, the compiler will probably create multiple copies of that constant.

It's certainly true that having multiple copies of the data wastes storage, but given the amount of memory in most modern systems, that's a minor concern. A bigger problem is that the program usually accesses these constants in a random fashion, so they're rarely sitting in cache and, in fact, they often evict some other more frequently used data from cache.

One solution to this problem is to manage the floating-point "constants" yourself. Because these constants are effectively variables as far as the program is concerned, you can take charge of this process and place the floating-point constants you'll need in initialized static variables. For example:

```
#include <stdlib.h>
#include <stdio.h>

static double OnePointZero_c = 1.0;

int main( int argc, char **argv, char **envp )
{
  static double i;

  i = OnePointZero_c;
}
```

In this example, of course you gain absolutely nothing by treating the floating-point constants as static variables. However, in more complex situations where you have several floating-point constants, you can analyze your program to determine which constants you access often and place the variables for those constants at adjacent memory locations. Because of the way most CPUs handle spatial locality of reference (see *WGC1*), when you access one of these constant objects, the cache line will be filled with the values of the adjacent objects as well. Therefore, when you access those other objects within a short period of time, it's likely that their values will be in the cache. Another advantage to managing these constants yourself is that you can create a global set of constants that you can reference from different compilation units (source files), so the program accesses only a single memory object for a given constant rather the multiple memory objects (one for each compilation unit). Compilers generally aren't smart enough to make decisions like this concerning your data.

6.10 String Constants

Like floating-point constants, string constants cannot be processed efficiently by most compilers (even if they are literal or manifest constants). Understanding when you should use manifest constants and when you should replace them with memory references can help you guide the compiler to produce better machine code. For example, most CPUs are not capable of encoding a string constant as part of an instruction. Using a manifest string constant may actually make your program less efficient. Consider the following C code:

```
#define strConst "A string constant"
        .
        .
        .
    printf( "string: %s\n", strConst );
        .
        .
        .
    sptr = strConst;
        .
        .
        .
    result = strcmp( s, strConst );
        .
        .
        .
```

The compiler (actually, the C preprocessor) expands the macro `strConst` to the string literal `"A string constant"` everywhere the identifier `strConst` appears in the source file, so this code is actually equivalent to:

```
    .
    .
    .
printf( "string: %s\n", "A string constant" );
    .
    .
    .
sptr = "A string constant";
    .
    .
    .
result = strcmp( s, "A string constant" );
```

The problem with this code is that the same string constant appears at different places throughout the program. In C/C++, the compiler places the string constant in memory and substitutes a pointer to the string. A nonoptimizing compiler might wind up making three separate copies of the string in memory, which wastes space because the data is exactly the same in all three cases. (Remember that we're talking about *constant* strings here.)

Compiler writers discovered this problem a few decades ago and modified their compilers to keep track of the strings in a given source file. If a program used the same string literal two or more times, the compiler wouldn't allocate storage for a second copy of the string. Instead, it would simply use the address of the earlier string. This optimization (constant folding) could reduce the size of the code if the same string appeared throughout a source file.

Unfortunately, constant folding doesn't always work properly. One problem is that many older C programs assign a string literal constant to a character pointer variable and then proceed to change the characters in that literal string. For example:

```
sptr = "A String Constant";
    .
    .
    .
*(sptr+2) = 's';
    .
    .
    .
/* The following displays "string: 'a string Constant'" */

printf( "string: '%s'\n", sptr );
    .
    .
    .
/* This prints "a string Constant"! */

printf( "A String Constant" );
```

Compilers that reuse the same string constant fail if the user stores data into the string object, as this code demonstrates. Although this is bad programming practice, it occurred frequently enough in older C programs that compiler vendors couldn't use the same storage for multiple copies of the same string literal. Even if the compiler vendor were to place the string literal constant into write-protected memory to prevent this problem, there are other semantic issues that this optimization raises. Consider the following C/C++ code:

```
sptr1 = "A String Constant";
sptr2 = "A String Constant";
s1EQs2 = sptr1 == sptr2;
```

Will `s1EQs2` contain `true (1)` or `false (0)` after executing this instruction sequence? In programs written before C compilers had good optimizers available, this sequence of statements would leave `false` in `s1EQs2`. This was because the compiler created two different copies of the same string data and placed those strings at different addresses in memory (so the addresses the program assigns to `sptr1` and `sptr2` would be different). In a later compiler that kept only a single copy of the string data in memory, this code sequence would leave `true` sitting in `s1EQs2` because both `sptr1` and `sptr2` would be pointing at the same memory address. This difference exists regardless of whether or not the string data appears in write-protected memory.

To solve this dilemma, many compiler vendors provide a compiler option to enable programmers to determine whether the compiler should emit a single copy of each string or one copy for each occurrence of the string. If you don't write data into string literal constants or compare their addresses, you can select this option to reduce the size of your programs. If you have old code that requires separate copies of the string data (hopefully, you won't write new code that requires this), you can enable this option.

Unfortunately, many programmers are completely unaware of this option, and the default condition on some compilers is generally to make multiple copies of the string data. If you're using C/C++ or some other language that manipulates strings via pointers to the character data, investigate whether the compiler provides an option to merge identical strings and, if so, activate that feature in your compiler.

If your C/C++ compiler does not offer this string-merging optimization, you can implement it manually. To do so, just create a `char` array variable in your program and initialize it with the address of the string. Then use the name of that array variable exactly as you would a manifest constant throughout your program. For example:

```
char strconst[] = "A String Constant";
        .
        .
        .
    sptr = strconst;
        .
        .
        .
    printf( strconst );
        .
        .
        .
    if( strcmp( string, strconst ) == 0 )
    {
        .
        .
        .
    }
```

This code will maintain only a single copy of the string literal constant in memory, even if the compiler doesn't directly support the optimization. Actually, even if your compiler does directly support this optimization, there are several good reasons why you should use this trick rather than relying on your compiler to do the work for you.

- In the future you might have to port your code to a different compiler that doesn't support this optimization.
- By handling the optimization manually, you don't have to worry about it.
- By using a pointer variable rather than a string literal constant, you have the option of easily changing the string whose address this pointer contains under program control.
- In the future you might want to modify the program to switch (natural) languages under program control.
- You can easily share the string between multiple files.

This string optimization discussion assumes that your programming language manipulates strings by reference (that is, by using a pointer to the actual string data). Although this is certainly true for C/C++ programs, it is not true of all languages. Pascal implementations that support strings (such as Free Pascal) typically manipulate them by value rather than by reference. Any time you assign a string value to a string variable, the compiler makes a copy of the string data and places that copy in the storage reserved for the string variable. This copying process can be expensive and is unnecessary if your program never changes the data in the string variable. Worse still, if the (Pascal) program assigns a string literal to a string variable, the program will have two copies of the string floating around (the string literal constant in memory and the copy that the program made for the string variable). If the program never again changes the string (which is not at all uncommon), it will waste memory by maintaining two copies of the string when one would suffice. These reasons (space and speed) are probably why Borland went to a much more sophisticated string format when they created Delphi 4.0, abandoning the string format in earlier versions of Delphi.[5]

5. "Abandoning" is probably too strong a word here. Borland (Delphi's originator) continued to support the old format by using a different name for the `short` string data type.

Swift also treats strings as value objects. This means that, in the worst case, it will make a copy of a string literal whenever you assign that string literal to a string variable. However, Swift implements an optimization known as *copy-on-write*. Whenever you assign one string object to another, Swift just copies a pointer. Therefore, if multiple strings have been assigned the same value, Swift will use the same string data in memory for all the copies. When you modify some portion of the string, Swift will make a copy of the string prior to the modification (hence the name "copy-on-write") so that other string objects referencing the original string data are not affected by the change.

6.11 Composite Data Type Constants

Many languages support other composite constant types (such as arrays, structures/records, and sets) in addition to strings. Usually, the languages use these constants to statically initialize variables prior to the program's execution. For example, consider the following C/C++ code:

```
static int arrayOfInts[8] = {1,2,3,4,5,6,7,8};
```

Note that `arrayOfInts` is not a constant. Rather, it is the initializer that constitutes the array constant—that is, `{1,2,3,4,5,6,7,8}`. In the executable file, most C compilers simply overlay the eight integers at the address associated with `arrayOfInts` with these eight numeric values.

For example, here's what GCC emits for this variable:

```
LC0:          // LC0 is the internal label associated
              //  with arrayOfInts
    .long       1
    .long       2
    .long       3
    .long       4
    .long       5
    .long       6
    .long       7
    .long       8
```

There is no extra space consumed to hold the constant data, assuming that `arrayOfInts` is a static object in C.

The rules change, however, if the variable you're initializing is not a statically allocated object. Consider the following short C sequence:

```
int f()
{
  int arrayOfInts[8] = {1,2,3,4,5,6,7,8};
    .
    .
    .
} // end f
```

In this example, arrayOfInts is an *automatic* variable, meaning that the program allocates storage on the stack for the variable each time the program calls function f(). For this reason, the compiler cannot simply initialize the array with the constant data when the program loads into memory. The arrayOfInts object could actually lie at a different address on each activation of the function. To obey the semantics of the C programming language, the compiler will have to make a copy of the array constant and then physically copy that constant data into the arrayOfInts variable whenever the program calls the function. Using an array constant this way consumes extra space (to hold a copy of the array constant) and extra time (to copy the data). Sometimes the semantics of your algorithm requires a fresh copy of the data upon each new activation of the function f(). However, you need to recognize when this is necessary (and when the extra space and time are warranted) rather than blowing memory and CPU cycles.

If your program doesn't modify the array's data, you can use a static object that the compiler can initialize once when it loads the program into memory:

```
int f()
{
  static int arrayOfInts[8] = {1,2,3,4,5,6,7,8};
    .
    .
    .
} // end f
```

The C/C++ languages also support struct constants. The same space and speed considerations we've seen for arrays when initializing automatic variables also apply to struct constants.

Embarcadero's Delphi programming language also supports structured constants, though the term "constant" is a bit misleading here. Embarcadero calls them *typed constants,* and you declare them in the Delphi const section like this:

```
const
    ary: array[0..7] of integer = (1,2,3,4,5,6,7,8);
```

Although the declaration appears in a Delphi const section, Delphi actually treats it as a variable declaration. It's an unfortunate design choice, but for a programmer who wants to create structured constants, this mechanism works fine. As with the C/C++ examples in this section, it's important to remember that the constant in this example is actually the (1,2,3,4,5,6,7,8) object, not the ary variable.

Delphi (along with most modern Pascals, such as Free Pascal) supports several other composite constant types as well. Set constants are good examples. Whenever you create a set of objects, the Pascal compiler generally initializes some memory location with a powerset (bitmap) representation of the set's data. Wherever you refer to that set constant in your program, the Pascal compiler generates a memory reference to the set's constant data in memory.

Swift also supports composite data type constants for arrays, tuples, dictionaries, structs/classes, and other data types. For example, the following `let` statement creates an array constant with eight elements:

```
let someArray = [1,2,3,4,11,12,13,14]
```

6.12 Constants Don't Change

In theory, values bound to a constant don't change (Swift's `let` statement being the obvious exception). In modern systems, compilers that place constants in memory often put them in write-protected memory regions to force an exception if an inadvertent write occurs. Of course, few programs can be written using only read-only (or write-once) objects. Most programs require the ability to change the values of objects (*variables*) they manipulate. That is the subject of the next chapter.

6.13 For More Information

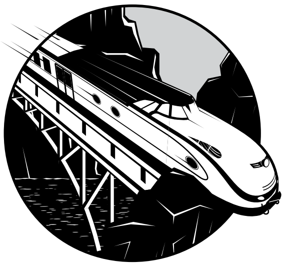
Duntemann, Jeff. *Assembly Language Step-by-Step.* 3rd ed. Indianapolis: Wiley, 2009.

Hyde, Randall. *The Art of Assembly Language.* 2nd ed. San Francisco: No Starch Press, 2010.

———. *Write Great Code, Volume 1: Understanding the Machine.* 2nd ed. San Francisco: No Starch Press, 2020.

7

VARIABLES IN A HIGH-LEVEL LANGUAGE



This chapter explores the low-level implementation of variables found in high-level languages. Although assembly language programmers usually have a good feel for the connection between variables and memory locations, HLLs add sufficient abstraction to obscure this relationship. We'll cover the following topics:

- The runtime memory organization typical for most compilers
- How the compiler breaks up memory into different sections and places variables into each
- The attributes that differentiate variables from other objects
- The difference between static, automatic, and dynamic variables
- How compilers organize automatic variables in a stack frame
- The primitive data types that hardware provides for variables
- How machine instructions encode the address of a variable

When you finish reading this chapter, you should have a good understanding of how to declare variables in your program to use the least amount of memory and produce fast-running code.

7.1 Runtime Memory Organization

As Chapter 4 discussed, operating systems (like macOS, Linux, or Windows) put different types of data into different sections (or *segments*) of main memory. Although it's possible to control the memory organization by running a linker and specifying various command-line parameters, by default Windows loads a typical program into memory using an organization like the one shown in Figure 7-1 (macOS and Linux are similar, although they rearrange some of the sections).

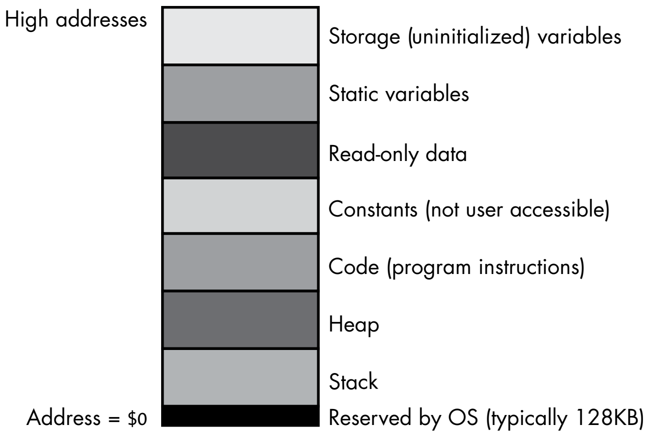

High addresses
Storage (uninitialized) variables
Static variables
Read-only data
Constants (not user accessible)
Code (program instructions)
Heap
Stack
Address = $0
Reserved by OS (typically 128KB)

Figure 7-1: Typical runtime memory organization for Windows

The operating system reserves the lowest memory addresses. Generally, your application cannot access data (or execute instructions) at the lowest addresses in memory. One reason the OS reserves this space is to help detect `NULL` pointer references. Programmers often initialize pointers with `NULL` (`0`) to indicate that the pointer is not valid. Should you attempt to access memory location `0` under such an OS, the OS will generate a *general protection fault* to indicate that it's an invalid memory location.

The remaining seven sections of memory hold different types of data associated with your program: the stack, the heap, the code, constants, read-only data, static (initialized) variables, and storage (uninitialized) variables.

Most of the time, a given application can live with the default layouts chosen for these sections by the compiler and linker/loader. In some cases, however, knowing the memory layout can help you develop shorter programs. For example, because the code section is usually read-only, you might be able to combine the code, constant, and read-only data sections into a single section, thereby saving any padding space that the compiler/linker may place between these sections. Although for large applications this is probably insignificant, for small programs it can have a big impact on the size of the executable.

Next we'll discuss each of these sections in detail.

7.1.1 The Code, Constant, and Read-Only Sections

The code (or *text*) section in memory contains the machine instructions for a program. Your compiler translates each statement you write into a sequence of one or more byte values (machine instruction opcodes). The CPU interprets these opcode values during program execution.

Most compilers also attach a program's read-only data and constant pool (constant table) sections to the code section because, like the code instructions, the read-only data is already write-protected. However, it is perfectly possible under Windows, macOS, Linux, and many other operating systems to create a separate section in the executable file and mark it as read-only. As a result, some compilers do support a separate read-only data section, and some compilers even create a different section (the constant pool) for the constants that the compiler emits. These sections contain initialized data, tables, and other objects that the program should not change during program execution.

Many compilers generate multiple code sections and leave it up to the linker to combine them into a single code segment prior to execution. To understand why, consider the following short Pascal code fragment:

```
if( SomeBooleanExpression ) then begin

    << Some code that executes 99.9% of the time >>

end
else begin

    << Some code that executes 0.01% of the time >>

end;
```

Without worrying about how it does so, assume that the compiler can figure out that the `then` section of this `if` statement executes far more often than the `else` section. An assembly programmer, wanting to write the fastest possible code, might encode this sequence as follows:

```
    << evaluate Boolean expression, leave true/false in EAX >>
    test( eax, eax );
    jz exprWasFalse;
    << Some code that executes 99.9% of the time >>
rtnLabel:
    << Code normally following the last END in the
               Pascal example >>
        .
        .
        .
```

```
// somewhere else in the code, not in the direct execution path
// of the above:

exprWasFalse:
    << Some code that executes 0.1% of the time >>

    jmp rtnLabel;
```

This assembly code might seem a bit convoluted, but keep in mind that any control transfer instruction is probably going to consume a lot of time because of pipelined operation on modern CPUs (see Chapter 9 of *WGC1* for the details). Code that executes without branching (or that falls straight through) executes the fastest. In the previous example, the common case falls straight through 99.9 percent of the time. The rare case winds up executing two branches (one to transfer to the `else` section and one to return to the normal control flow). But because this code rarely executes, it can afford to take longer to do so.

Many compilers use a little trick to move sections of code around like this in the machine code they generate—they emit the code sequentially, but place the `else` code in a separate section. The following MASM code demonstrates this technique:

```
    << evaluate Boolean expression, leave true/false in EAX >>
    test eax, eax
    jz exprWasFalse
    << Some code that executes 99.9% of the time >>
alternateCode segment

exprWasFalse:
    << Some code that executes 0.1% of the time >>

    jmp rtnLabel;
alternateCode ends

rtnLabel:
    << Code normally following the last END in the Pascal example >>
```

Even though the `else` section code appears to immediately follow the `then` section's code, placing it in a different segment tells the assembler/linker to move this code and combine it with other code in the `alternateCode` segment. This little trick, because it relies upon the assembler or linker to move the code, can simplify HLL compilers. (GCC, for example, uses this approach to move code around in the assembly language file it emits.) As a result, you will see this trick being used on occasion and can expect some compilers to produce multiple code segments.

7.1.2 The Static Variables Section

Many languages provide the ability to initialize a global variable during the compilation phase. For example, in C/C++ you could use statements like the following to provide initial values for these static objects:

```
static int i = 10;
static char ch[] = { 'a', 'b', 'c', 'd' };
```

In C/C++ and other languages, the compiler places these initial values in the executable file. When you execute the application, the OS loads the portion of the executable file that contains these static variables into memory so that the values appear at the addresses associated with those variables. Therefore, when the program in this example first begins execution, `i` and `ch` will have these values bound to them.

The static section is often called the `DATA` or `_DATA` segment in the assembly listings that most compilers produce. As an example, consider the following C code fragment:

```
#include <stdlib.h>
#include <stdio.h>

static char *c = "";
static int i = 2;
static int j = 1;
static double array[4] = {0.0, 1.0, 2.0, 3.0};

int main( void )
{

      .
      .
      .
```

Here's the MASM assembly code that the Visual C++ compiler emits for those declarations:

```
_DATA   SEGMENT
?c@@3PEADEA   DQ  FLAT:$SG6912                ; c
?i@@3HA       DD  02H                         ; i
?j@@3HA       DD  01H                         ; j
?array@@3PANA DQ  0000000000000000r      ; 0  ; array
              DQ  03ff0000000000000r     ; 1
              DQ  04000000000000000r     ; 2
              DQ  04008000000000000r     ; 3
_DATA   ENDS
```

As you can see, the Visual C++ compiler places these variables in the `_DATA` segment.

7.1.3 The Storage Variables Section

Most operating systems zero out memory prior to program execution. Therefore, if an initial value of `0` is suitable, you don't need to waste any disk space with the static object's initial value. Generally, however, compilers treat uninitialized variables in a static section as though you've initialized them with `0`, which consumes disk space. Some operating systems provide another section type, the storage variables section (also known as the *BSS section*), to avoid this wasted disk space.

This section is where compilers typically store static objects that don't have an explicit initial value. BSS, as noted in Chapter 4, stands for "block started by a symbol," which is an old assembly language term describing a pseudo-opcode you would use to allocate storage for an uninitialized static array. In modern operating systems like Windows and Linux, the compiler/linker puts all uninitialized variables into a BSS section that simply tells the OS how many bytes to set aside for that section. When the OS loads the program into memory, it reserves sufficient memory for all the objects in the BSS section and fills this range of memory with zeros. Note that the BSS section in the executable file doesn't contain any actual data, so programs that declare large uninitialized static arrays in a BSS section will consume less disk space. The following is the C/C++ example from the previous section, modified to remove the initializers so that the compiler will place the variables in the BSS section:

```
#include <stdlib.h>
#include <stdio.h>

static char *c;
static int i;
static int j;
static double array[4];

int main( void )
{
      .
      .
      .
```

Here is the Visual C++ output:

```
_BSS    SEGMENT
?c@@3PEADEA   DQ  01H DUP (?)                            ; c
?i@@3HA       DD  01H DUP (?)                            ; i
?j@@3HA       DD  01H DUP (?)                            ; j
?array@@3PANA DQ  04H DUP (?)                            ; array
_BSS    ENDS
```

Not all compilers use a BSS section. Many Microsoft languages and linkers, for example, simply combine the uninitialized objects with the static/read-only data section and explicitly give them an initial value of `0`. Although Microsoft claims that this scheme is faster, it certainly makes executable files larger if your code has large, uninitialized arrays (because each byte of the array winds up in the executable file—something that would not happen if the compiler placed the array in a BSS section). Note, however, that this is a default condition and you can change it by setting the appropriate linker flags.

7.1.4 The Stack Section

The stack is a data structure that expands and contracts in response to procedure invocations and returns, among other things. At runtime, the system places all automatic variables (nonstatic local variables), subroutine parameters, temporary values, and other objects in the stack section of memory in a special data structure called the *activation record* (which is aptly named, as the system creates it when a subroutine first begins execution and deallocates it when the subroutine returns to its caller). Therefore, the stack section in memory is very busy.

Many CPUs implement the stack using a special-purpose register called the *stack pointer*. Other CPUs (particularly some RISC CPUs) don't provide an explicit stack pointer, instead using a general-purpose register for stack implementation. If a CPU provides a stack pointer, we say that the CPU supports a *hardware stack*; if it uses a general-purpose register, then we say that it uses a *software-implemented stack*. The 80x86 is a good example of a CPU that provides a hardware stack, and the PowerPC family is a good example of a CPU family with a software-implemented stack (most PowerPC programs use R1 as the stack pointer register). The ARM CPU supports a pseudo–hardware stack; it assigns one of the general-purpose registers as the hardware stack pointer but still requires an application to explicitly maintain the stack. Systems that provide hardware stacks can generally manipulate data on the stack using fewer instructions than systems with a software-implemented stack. On the other hand, RISC CPU designers who've chosen to use a software stack implementation feel that the presence of a hardware stack actually slows down all instructions the CPU executes. In theory, you could argue that the RISC designers are right; in practice, the 80x86 family includes some of the fastest CPUs around, providing ample proof that having a hardware stack doesn't necessarily mean you'll wind up with a slow CPU.

7.1.5 The Heap Section and Dynamic Memory Allocation

Although simple programs may need only static and automatic variables, sophisticated programs need to be able to allocate and deallocate storage dynamically under program control. In the C and HLA languages, you would use the `malloc()` and `free()` functions for this purpose. C++ provides the `new` and `delete` (and `std::unique_ptr`) operators. Pascal uses `new` and `dispose`. Java and Swift use `new` (deallocation is automatic in these languages). Other languages provide comparable routines. These memory allocation routines have a few things in common:

- They let the programmer request how many bytes of storage to allocate (either by explicitly specifying the number of bytes to allocate or by specifying some data type whose size is known).
- They return a *pointer* to the newly allocated storage (that is, the address of that storage).
- They provide a facility for returning the storage space to the system once it is no longer needed so the system can reuse it in a future allocation call.

Dynamic memory allocation takes place in a section of memory known as the *heap*. Generally, an application refers to data on the heap using pointer variables, either implicitly or explicitly; some languages, like Java and Swift, implicitly use pointers behind the scenes. Thus, these objects in heap memory are usually referred to as *anonymous variables* because they are referred to by their memory address (via pointers) rather than by a name.

The OS and application create the heap section in memory after the program begins execution; the heap is never a part of the executable file. Generally, the OS and language runtime libraries maintain the heap for an application. Despite the variations in memory management implementations, it's a good idea for you to have a basic idea of how heap allocation and deallocation operate, because using them inappropriately will have a very negative impact on your application performance.

7.2 What Is a Variable?

If you consider the word *variable*, it's obvious that it describes something that *varies*. But exactly what is it that varies? Most programmers would say that it's the value that can vary during program execution. In fact, though, there are several things that can vary, so before defining a variable explicitly, we'll discuss some characteristics that variables (and other objects) may possess.

7.2.1 Attributes

An *attribute* is some feature that is associated with an object. For example, common attributes of a variable include its name, its memory address, its size (in bytes), its runtime value, and a data type associated with that value. Different objects may have different sets of attributes. For example, a data type is an object that has attributes such as a name and size, but it won't usually have a value or memory location associated with it. A constant can have attributes such as a value and a data type, but it doesn't have a memory location and it might not have a name (for example, if it's a literal constant). A variable may possess all of these attributes. Indeed, the attribute list usually determines whether an object is a constant, data type, variable, or something else.

7.2.2 Binding

Binding, introduced in Chapter 6, is the process of associating an attribute with an object. For example, when a value is assigned to a variable, the value is bound to that variable at the point of the assignment. This bond remains until some other value is bound to the variable (via another assignment operation). Likewise, if you allocate memory for a variable while the program is running, the variable is bound to the memory address at that point. The variable and address are bound until you associate a different address with the variable. Binding needn't occur at runtime. For example, values are bound to constant objects during compilation, and these bonds cannot change while the program is running. Similarly, addresses are bound to some variables at compile time, and those memory addresses cannot change during program execution (see "Binding Times" on page 150 for more details).

7.2.3 Static Objects

Static objects have an attribute bound to them prior to the application's execution. Constants are good examples of static objects; they have the same value bound to them throughout program execution.[1] Global (program-level) variables in programming languages like Pascal, C/C++, and Ada are also examples of static objects because they have the same memory address bound to them throughout the program's lifetime. The system binds attributes to a static object before the program begins execution (usually during compilation, linking, or even loading, though it is possible to bind values even earlier).

7.2.4 Dynamic Objects

Dynamic objects have some attribute bound to them during program execution. While it is running, the program may choose to change that attribute (*dynamically*). Dynamic attributes usually cannot be determined at compile time. Examples of dynamic attributes include values bound to variables at runtime and memory addresses bound to certain variables at runtime (for example, via a `malloc()` or other memory allocation function call).

7.2.5 Scope

The *scope* of an identifier is the section of the program where the identifier's name is bound to the object. Because names in most compiled languages exist only during compilation, scope is usually a static attribute (although in some languages it can be dynamic, as I'll explain shortly). By controlling where a name is bound to an object, you can reuse that name elsewhere in the program.

Most modern programming languages (such as C/C++/C#, Java, Pascal, Swift, and Ada) support the concept of *local* and *global* variables. A local

1. Swift constants defined with the `let` statement are an exception to this rule.

variable's name is bound to a particular object only within a given section of a program (for example, within a particular function). Outside the scope of that object, the name can be bound to a different object. This allows a global and a local object to share the same name without any ambiguity. This may seem potentially confusing, but being able to reuse variable names like `i` or `j` throughout a project can spare you from having to dream up equally meaningless unique variable names for loop indexes and other uses in the program. The scope of the object's declaration determines where the name applies to a given object.

In interpretive languages, where the interpreter maintains the identifier names during program execution, scope can be a dynamic attribute. For example, in various versions of the BASIC programming language, `dim` is an executable statement. Before you execute `dim`, the name you define might have a completely different meaning than it does after you execute `dim`. SNOBOL4 is another language that supports dynamic scope. Still, most programming languages avoid dynamic scope because using it can result in difficult-to-understand programs.

Technically, scope can apply to any attribute, not just names, but this book will use the term only in contexts where a name is bound to a given variable.

7.2.6 Lifetime

The *lifetime* of an attribute extends from the point when you first bind an attribute to an object to the point you break that bond, perhaps by binding a different attribute to the object. If the program associates some attribute with an object and never breaks that bond, the lifetime of the attribute is from the point of association to the point the program terminates. For example, the lifetime of a variable is from the time you first allocate memory for the variable to the moment you deallocate that variable's storage. Because a program binds static objects prior to execution (and static attributes do not change during program execution), the lifetime of a static object extends from when the program begins execution to when it terminates.

7.2.7 Variable Definition

To return to the question that started this section, we can now define *variable* as an object that can have a value bound to it dynamically. That is, the program can change the variable's value attribute at runtime. Note the operative word *can*. It is necessary only for the program *to be able* to change a variable's value at runtime; it doesn't *have* to do so for the object to be considered a variable.

While dynamic binding of a value to an object is the defining attribute of a variable, other attributes may be dynamic or static. For example, the memory address of a variable can be statically bound to the variable at compile time or dynamically bound at runtime. Likewise, variables in some languages have dynamic types that change during program execution, while other variables have static types that remain fixed over an application's execution. Only the binding of the value determines whether the object is a variable or something else (such as a constant).

7.3 Variable Storage

Values must be stored in and retrieved from memory.[2] To do this, a compiler must bind a variable to one or more memory locations. The variable's type determines the amount of storage it requires. Character variables may require as little as a single byte of storage, while large arrays or records can require thousands, millions, or more. To associate a variable with some memory, a compiler (or runtime system) binds the address of that memory location to that variable. When a variable requires two or more memory locations, the system usually binds the address of the first memory location to the variable and assumes that the contiguous locations following that address are also bound to the variable at runtime.

Three types of bindings are possible between variables and memory locations: static binding, pseudo-static (automatic) binding, and dynamic binding. Variables are generally classified as static, automatic, or dynamic based upon how they are bound to their memory locations.

7.3.1 Static Binding and Static Variables

Static binding occurs prior to runtime, at one of four possible times: at language design time, at compile time, at link time, or when the system loads the application into memory (but prior to execution). Binding at language design time is not all that common, but it does occur in some languages (especially assembly languages). Binding at compile time is common in assemblers and compilers that directly produce executable code. Binding at link time is fairly common (for example, some Windows compilers do this). Binding at load time, when the OS copies the executable into memory, is probably the most common for static variables. We'll look at each possibility in turn.

7.3.1.1 Binding at Language Design Time

An address can be assigned at language design time when a language designer associates a language-defined variable with a specific hardware address (for example, an I/O device or a special kind of memory), and that address never changes in any program. Such objects are common in embedded systems and rarely found in applications on general-purpose computer systems. For example, on an 8051 microcontroller, many C compilers and assemblers automatically associate certain names with fixed locations in the 128 bytes of data space found on the CPU. CPU register references in assembly language are good examples of variables bound to some location at language design time.

7.3.1.2 Binding at Compile Time

An address can be assigned at compile time when the compiler knows the memory region where it can place static variables at runtime. Generally,

2. Technically, you can store values in machine registers, too. We'll consider machine registers a special form of memory for the sake of this discussion.

such compilers generate absolute machine code that must be loaded at a specific address in memory prior to execution. Most modern compilers generate relocatable code and, therefore, don't fall into this category. Nevertheless, lower-end compilers, high-speed student compilers, and compilers for embedded systems often use this binding technique.

7.3.1.3 Binding at Link Time

Certain linkers and related tools can link together various relocatable object modules of an application and create an absolute load module. So, while the compiler produces relocatable code, the linker binds memory addresses to the variables (and machine instructions). Usually, the programmer specifies (via command-line parameters or a linker script file) the base address of all the static variables in the program; the linker will bind the static variables to consecutive addresses starting at the base address. Programmers who are placing their applications in read-only memory (ROM), such as a BIOS (Basic Input/Output System) ROM for a PC, often employ this scheme.

7.3.1.4 Binding at Load Time

The most common form of static binding occurs at load time. Executable formats such as Microsoft's PE/COFF and Linux's ELF usually embed relocation information in the executable file. The OS, when it loads the application into memory, decides where to place the block of static variable objects and then patches all the addresses within instructions that reference those static objects. This allows the loader (for example, the OS) to assign a different address to a static object each time it loads it into memory.

7.3.1.5 Static Variable Binding

A static variable has a memory address bound to it prior to program execution, and enjoys a couple of advantages over other variable types. Because the compiler knows a static variable's address prior to runtime, it can often use an *absolute addressing mode* or some other simple addressing mode to access that variable. Static variable access is often more efficient than other variable accesses because it doesn't require any additional setup.[3]

Another benefit of static variables is that they retain any value bound to them until you explicitly bind another value or until the program terminates. This means that static variables retain values while other events (such as procedure activation and deactivation) occur. Different threads in a multithreaded application can also share data using static variables.

Static variables also have a few disadvantages worth mentioning. First of all, because the lifetime of a static variable matches that of the program,

3. At least, on an 80x86 CPU or some other CPU that supports absolute addresses. Most RISC processors do not support absolute addressing, so the program must set up a static frame pointer or global frame register when it first begins execution, but it only has to do so once, so we can ignore the associated performance issues.

it consumes memory the entire time the program is running. This is true even if the program no longer requires the value held by the static object.

Another disadvantage to static variables (particularly when using the absolute addressing mode) is that the entire absolute address must usually be encoded as part of the instruction, which makes the instruction much larger. Indeed, on most RISC processors an absolute addressing mode isn't even available because you cannot encode an absolute address in a single instruction.

Finally, code that uses static objects is not *reentrant* (meaning two threads or processes can concurrently execute the same code sequence); this means more effort is required to use that code in a multithreaded environment (where two copies of a section of code could be executing simultaneously, both accessing the same static object). However, multithreaded operation introduces a lot of complexity that is beyond the scope of this chapter, so we'll ignore this issue for now.

NOTE *See any good textbook on operating system design or concurrent programming for more details concerning the use of static objects.* Foundations of Multithreaded, Parallel, and Distributed Programming *by Gregory R. Andrews (Addison-Wesley, 1999) is a good place to start.*

The following example demonstrates the use of static variables in a C program and shows the 80x86 code that the Visual C++ compiler generates to access them:

```
#include <stdio.h>

static int i = 5;
static int j = 6;

int main( int argc, char **argv)
{

    i = j + 3;
    j = i + 2;
    printf( "%d %d\n", i, j );
    return 0;
}



; The following are the memory declarations
; for the 'i' and 'j' variables. Note that
; these are declared in the global '_DATA'
; section.

_DATA   SEGMENT
i       DD      05H
j       DD      06H
$SG6835 DB      '%d %d', 0aH, 00H
_DATA   ENDS
```

```
main    PROC
; File c:\users\rhyde\test\t\t\t.cpp
; Line 8
;
;    int main( int argc, char **argv)
;    {
$LN3:
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 40                                 ; 00000028H
; Line 10
;
;           i = j + 3;
;
; Load the EAX register with the
; current value of the global j
; variable using the displacement-only
; addressing mode, add three to the
; value, and store back into 'i':

        mov     eax, DWORD PTR j
        add     eax, 3
        mov     DWORD PTR i, eax

; Line 11
;
;           j = i + 2;
;
        mov     eax, DWORD PTR i
        add     eax, 2
        mov     DWORD PTR j, eax

; Line 12
; Load i, j, and format string into appropriate registers
; and call printf:

        mov     r8d, DWORD PTR j
        mov     edx, DWORD PTR i
        lea     rcx, OFFSET FLAT:$SG6835
        call    printf
; Line 13
;
; RETURN 0

        xor     eax, eax
; Line 14
        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
_TEXT   ENDS
```

As the comments point out, the assembly language code the compiler emits uses the displacement-only addressing mode to access all the static variables.

7.3.2 Pseudo-Static Binding and Automatic Variables

Automatic variables have an address bound to them when a procedure or other block of code begins execution. The program releases that storage when the block or procedure completes execution. We call these objects *automatic* variables because the runtime code automatically allocates and deallocates storage for them, as needed.

In most programming languages, automatic variables use a combination of static and dynamic binding known as *pseudo-static binding*. The compiler assigns an offset from a base address to a variable name during compilation. At runtime the offset always remains fixed, but the base address can vary. For example, a procedure or function allocates storage for a block of local variables (the activation record, introduced earlier in the chapter) and then accesses the local variables at fixed offsets from the start of that block of storage. Although the program cannot determine the final memory address of the variable until runtime, the compiler can select an offset that never changes during program execution, hence the name *pseudo-static*.

Some programming languages use the term *local variables* in place of automatic variables. A local variable's name is statically bound to a given procedure or block (that is, the scope of the name is limited to that procedure or block of code). Therefore, *local* is a static attribute in this context. It's easy to see why the terms *local variable* and *automatic variable* are often confused. In some programming languages, such as Pascal, local variables are always automatic variables and vice versa. Nonetheless, always keep in mind that *local* is a static attribute and *automatic* is a dynamic one.[4]

Automatic variables have a couple of important advantages. First, they consume storage only while the procedure or block containing them is executing. This allows multiple blocks and procedures to share the same pool of memory for their automatic variable needs. Although some extra code must execute in order to manage automatic variables (in the activation record), this requires only a few machine instructions on most CPUs and has to be done only once for each procedure/block entry and exit. While in certain circumstances, the cost can be significant, the extra time and space needed to set up and tear down the activation record is usually inconsequential. Another advantage of automatic variables is that they often use a *base-plus-offset* addressing mode, where the base of the activation record is kept in a register and the offsets into the activation record are small—often 256 bytes or fewer. Therefore, CPUs don't have to encode a full 32-bit (for example) address as part of the machine instruction—just an 8-bit (or other small) displacement, yielding shorter instructions. It's also worth noting that automatic variables are "thread-safe" and code that uses automatic variables can be reentrant. This is because each thread maintains its own stack space (or similar data structure) where compilers maintain automatic

4. Some languages, such as C/C++, allow you to declare *local static variables*. Such variables have a local name whose scope is limited to the function in which you declare them, but have a lifetime that equals the execution of the entire program.

variables; therefore, each thread will have its own copy of any automatic variables the program uses.

Automatic variables do have some disadvantages, though. If you want to initialize an automatic variable, you have to use machine instructions to do so. You can't initialize an automatic variable, as you can static variables, when the program loads into memory. Also, any values maintained in automatic variables are lost whenever you exit the block or procedure containing them. As noted, automatic variables require a small amount of overhead; some machine instructions must execute in order to build and destroy the activation record containing those variables.

Here's a short C example that uses automatic variables and the 80x86 assembly code that the Microsoft Visual C++ compiler produces for it:

```
#include <stdio.h>

int main( int argc, char **argv)
{

    int i;
    int j;

    j = 1;
    i = j + 3;
    j = i + 2;
    printf( "%d %d\n", i, j );
    return 0;
}


; Data emitted for the string constant
; in the printf function call:

CONST   SEGMENT
$SG6917 DB      '%d %d', 0aH, 00H
CONST   ENDS


PUBLIC  _main
EXTRN   _printf:NEAR
; Function compile flags: /Ods

_TEXT   SEGMENT
j$ = 32
i$ = 36
argc$ = 64
argv$ = 72
main    PROC
; File c:\users\rhyde\test\t\t\t.cpp
; Line 5
$LN3:
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 56                                 ; 00000038H
; Line 10
        mov     DWORD PTR j$[rsp], 1
; Line 11
        mov     eax, DWORD PTR j$[rsp]
        add     eax, 3
        mov     DWORD PTR i$[rsp], eax
; Line 12
        mov     eax, DWORD PTR i$[rsp]
        add     eax, 2
        mov     DWORD PTR j$[rsp], eax
; Line 13
        mov     r8d, DWORD PTR j$[rsp]
        mov     edx, DWORD PTR i$[rsp]
        lea     rcx, OFFSET FLAT:$SG6917
        call    printf
; Line 14
        xor     eax, eax
; Line 15
        add     rsp, 56                                 ; 00000038H
        ret     0
main    ENDP
_TEXT   ENDS
```

Note that when accessing automatic variables, the assembly code uses a *base-plus-displacement* addressing mode (for example, `j$[rsp]`). This addressing mode is often shorter than the displacement-only or RIP-relative addressing mode that static variables use (assuming, of course, that the offset to the automatic object is within 127 bytes of the base address held in RSP).[5]

7.3.3 Dynamic Binding and Dynamic Variables

A dynamic variable has storage bound to it at runtime. In some languages, the application programmer is completely responsible for binding addresses to dynamic objects; in other languages, the runtime system automatically allocates and deallocates storage for a dynamic variable.

Dynamic variables are generally allocated on the heap via a memory allocation function such as `malloc()` or `new()` (or `std::unique_ptr`). The compiler has no way of determining the runtime address of a dynamic object, so the program must always refer to a dynamic object indirectly—that is, by using a pointer.

The big advantage to dynamic variables is that the application controls their lifetimes. Dynamic variables consume storage only as long as necessary, and the runtime system can reclaim that storage when the variable no longer requires it. Unlike automatic variables, the lifetime of a dynamic

5. Visual C++ would normally use RBP as the base register (pointing at the activation record). In this particular example, Visual C++ was able to determine that it could optimize out setting up the RBP register and access local variables using the RSP register as the base pointer register.

variable is not tied to the lifetime of some other object, such as a procedure or code block entry and exit. Memory is bound to a dynamic variable at the point the variable first needs it, and can be released when the variable no longer needs it. For variables that require considerable storage, then, dynamic allocation can make efficient use of memory.

Another advantage to dynamic variables is that most code references dynamic objects using a pointer. If that pointer value is already sitting in a CPU register, the program can usually reference that data using a short machine instruction, requiring no extra bits to encode an offset or address.

Dynamic variables have several disadvantages as well. First, some storage overhead is often necessary to maintain them. Static and automatic objects usually don't require extra storage; the runtime system, on the other hand, often requires some number of bytes to keep track of each dynamic variable in the system. This overhead ranges anywhere from 4 or 8 bytes to many dozens of bytes (in an extreme case) and keeps track of things like the current memory address of the object, the size of the object, and its type. If you're allocating small objects, like integers or characters, the amount of storage required for bookkeeping purposes could exceed the storage required for the actual data. Also, since most languages reference dynamic objects using pointer variables, those pointers require some additional storage above and beyond the actual storage for the dynamic data.

Another problem with dynamic variables is performance. Because dynamic data is usually found in memory, the CPU has to access memory (which is slower than cached memory) on nearly every dynamic variable access. Even worse, accessing dynamic data often requires two memory accesses—one to fetch the pointer's value and one to fetch the dynamic data, indirectly through the pointer. Managing the heap, where the runtime system keeps the dynamic data, can also impact performance. Whenever an application requests storage for a dynamic object, the runtime system has to search for a contiguous block of free memory large enough to satisfy the request. This search operation can be computationally expensive, depending on the heap's organization (which affects the amount of overhead storage associated with each dynamic variable). Furthermore, when releasing a dynamic object, the runtime system may need to execute some code in order to free up that storage for use by other dynamic objects. These runtime heap allocation and deallocation operations are usually far more expensive than allocating and deallocating a block of automatic variables during procedure entry/exit.

Another consideration with dynamic variables is that some languages (such as Pascal and C/C++[6]) require the application programmer to explicitly allocate and deallocate storage for dynamic variables. Without automatic allocation and deallocation, defects due to human error can creep into the code. This is why languages such as C#, Java, and Swift attempt to handle dynamic allocation automatically, even though this process can be slower.

6. Modern versions of C++ provide automatic deallocation when you're using smart pointers.

Here's a short example in C that demonstrates the kind of code that the Microsoft Visual C++ compiler generates in order to access dynamic objects allocated with `malloc()`.

```
#include <stdlib.h>
#include <stdio.h>


int main( int argc, char **argv)
{

    int *i;
    int *j;


    i = (int *) malloc( sizeof( int ) );
    j = (int *) malloc( sizeof( int ) );
    *i = 1;
    *j = 2;
    printf( "%d %d\n", *i, *j );
    free( i );
    free( j );
    return 0;
}
```

Here's the machine code the compiler generates, including (manually inserted) comments that describe the extra work needed to access dynamically allocated objects:

```
_DATA   SEGMENT
$SG6837 DB      '%d %d', 0aH, 00H
_DATA   ENDS
PUBLIC  _main
_TEXT   SEGMENT
i$ = 32
j$ = 40
argc$ = 64
argv$ = 72
main    PROC
; File c:\users\rhyde\test\t\t\t.cpp
; Line 7 // Construct the activation record
$LN3:
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 56                                 ; 00000038H

; Line 13
; Call malloc and store the returned
; pointer value into the i variable:

        mov     ecx, 4
        call    malloc
        mov     QWORD PTR i$[rsp], rax
```

```
; Line 14
; Call malloc and store the returned
; pointer value into the j variable:

        mov     ecx, 4
        call    malloc
        mov     QWORD PTR j$[rsp], rax

; Line 15
; Store 1 into the dynamic variable pointed
; at by i. Note that this requires two
; instructions.

        mov     rax, QWORD PTR i$[rsp]
        mov     DWORD PTR [rax], 1

; Line 16
; Store 2 into the dynamic variable pointed
; at by j. This also requires two instructions.

        mov     rax, QWORD PTR j$[rsp]
        mov     DWORD PTR [rax], 2

; Line 17
; Call printf to print the dynamic variables'
; values:

        mov     rax, QWORD PTR j$[rsp]
        mov     r8d, DWORD PTR [rax]
        mov     rax, QWORD PTR i$[rsp]
        mov     edx, DWORD PTR [rax]
        lea     rcx, OFFSET FLAT:$SG6837
        call    printf

; Line 18
; Free the two variables
;
        mov     rcx, QWORD PTR i$[rsp]
        call    free
; Line 19
        mov     rcx, QWORD PTR j$[rsp]
        call    free

; Line 20
; Return a function result of zero:
        xor     eax, eax
; Line 21
        add     rsp, 56                                 ; 00000038H
        ret     0
main    ENDP
_TEXT   ENDS
END
```

As you can see, accessing dynamically allocated variables via a pointer requires a lot of extra work.

7.4 Common Primitive Data Types

Computer data always has a data type attribute that describes how the program interprets that data. The data type also determines the size (in bytes) of the data in memory. Data types can be divided into two categories: *primitive data types*, which the CPU can hold in a CPU register and operate upon directly, and *composite data types*, which are composed of smaller primitive data types. In the following sections we'll review (from *WGC1*) the primitive data types found on most modern CPUs, and in the next chapter I'll begin discussing composite data types.

7.4.1 Integer Variables

Most programming languages provide some mechanism for storing integer values in memory variables. In general, a programming language uses either unsigned binary representation, two's-complement representation, or binary-coded decimal representation (or a combination of these) to represent integer values.

Perhaps the most fundamental property of an integer variable in a programming language is the number of bits allocated to represent that integer value. In most modern programming languages, the number of bits used to represent an integer value is usually 8, 16, 32, 64, or some other power of two. Many languages provide only a single size for representing integers, but some languages let you select from several different sizes. You choose the size based on the range of values you want to represent, the amount of memory you want the variable to consume, and the performance of arithmetic operations involving that value. Table 7-1 lists some common sizes and ranges for various signed, unsigned, and decimal integer variables.

Not all languages support all of these different sizes (indeed, to support all of them in the same program, you'd probably have to use assembly language). As noted earlier, some languages provide only a single size, which is usually the processor's native integer size (that is, the size of a CPU general-purpose integer register).

Languages that do provide multiple integer sizes often don't give you an explicit selection of sizes from which to choose. For example, the C programming language provides up to five different integer sizes: `char` (which is always 1 byte), `short`, `int`, `long`, and `long long`. With the exception of the `char` type, C does not specify the sizes of these integer types other than to state that `short` integers are less than or equal to `int` objects in size, `int` objects are less than or equal to `long` integers in size, and `long` integers are less than or equal to `long long` integers in size. (In fact, all four could be the same size.) C programs that depend on integers being a certain size may fail when compiled with different compilers that don't use the same sizes as the original compiler.

NOTE *C99 and C++11 include types of exact sizes: `int8_t`, `int16_t`, `int32_t`, `int64_t`, and so on.*

Table 7-1: Common Integer Sizes and Their Ranges

Size, in bits	Representation	Unsigned range
8	Unsigned	`0..255`
	Signed	`-128..+127`
	Decimal	`0..99`
16	Unsigned	`0..65,536`
	Signed	`-32768..+32,767`
	Decimal	`0..9999`
32	Unsigned	`0..4,294,967,295`
	Signed	`-2,147,483,648..+2,147,483,647`
	Decimal	`0..99999999`
64	Unsigned	`0..18,466,744,073,709,551,615`
	Signed	`-9,223,372,036,854,775,808.. +9,223,372,036,854,775,807`
	Decimal	`0..9999999999999999`
128	Unsigned	`0..340,282,366,920,938,463,563,374,607,431,768,211,455`
	Signed	`-170,141,183,460,469,231,731,687,303,715,884,105,728.. +170,141,183,460,469,231,731,687,303,715,884,105,727`
	Decimal	`0..99999999999999999999999999999999`

While it may seem inconvenient that various programming languages avoid specifying an exact size for an integer variable, keep in mind that this ambiguity is intentional. When you declare an "integer" variable in a given programming language, the language leaves it up to the compiler's implementer to choose the *best* size for that integer, based on performance and other considerations. The definition of "best" may change based on the CPU for which the compiler generates code. For example, a compiler for a 16-bit processor may choose to implement 16-bit integers because the CPU processes them most efficiently. A compiler for a 32-bit processor, however, may choose to implement 32-bit integers (for the same reason). Languages that specify the exact size of various integer formats (such as Java) can suffer as processor technology evolves and it becomes more efficient to process larger data objects. For example, when the world switched from 16-bit processors to 32-bit processors in general-purpose computer systems, it was actually faster to do 32-bit arithmetic on most of the newer processors. Therefore, compiler writers redefined *integer* to mean "32-bit integer" in order to maximize the performance of programs employing integer arithmetic.

Some programming languages provide support for unsigned integer variables as well as signed integers. At first glance, it might seem that the whole purpose behind supporting unsigned integers is to provide twice the number of positive values when negative values aren't required. In fact, there are many other reasons great programmers might choose unsigned over signed integers when writing efficient code.

The Swift programming language gives you explicit control over the size of integers. Swift provides 8-bit (signed) integers (`Int8`), 16-bit integers (`Int16`), 32-bit integers (`Int32`), and 64-bit integers (`Int64`). Swift also provides an `Int` type that's either 32 bits or 64 bits depending on the native (most efficient) integer format for the underlying CPU. Swift further provides 8-bit unsigned integers (`UInt8`), 16-bit unsigned integers (`UInt16`), 32-bit unsigned integers (`UInt32`), 64-bit unsigned integers (`UInt64`), and a generic `UInt` type whose size is determined by the native CPU size.

On some CPUs, unsigned integer multiplication and division are faster than their signed counterparts. You can compare values within the range `0..`*`n`* more efficiently using unsigned integers rather than signed integers (the unsigned case requires only a single comparison against *`n`*); this is especially important when checking bounds of array indices where the array's element index begins at `0`.

Many programming languages allow you to include variables of different sizes within the same arithmetic expression. The compiler automatically sign-extends or zero-extends operands to the larger size within an expression as needed to compute the final result. The problem with this automatic conversion is that it hides the fact that extra work is required to process the expression, and the expressions themselves don't explicitly show this. An assignment statement such as:

```
x = y + z - t;
```

could be a short sequence of machine instructions if the operands are all the same size, or it could require some additional instructions if the operands have different sizes. For example, consider the following C code:

```
#include <stdio.h>

static char c;
static short s;
static long l;

static long a;
static long b;
static long d;

int main( int argc, char **argv)
{

    l = l + s + c;
    printf( "%ld %hd %hhd\n", l, s, c );

    a = a + b + d;
    printf( "%ld %ld %ld\n", a, b, d );

    return 0;
}
```

Compiling it with the Visual C++ compiler gives the following two assembly language sequences for the two assignment statements:

```
;          l = l + s + c;
;
       movsx   eax, WORD PTR s
       mov     ecx, DWORD PTR l
       add     ecx, eax
       mov     eax, ecx
       movsx   ecx, BYTE PTR c
       add     eax, ecx
       mov     DWORD PTR l, eax
;
;          a = a + b + d;
;
       mov     eax, DWORD PTR b
       mov     ecx, DWORD PTR a
       add     ecx, eax
       mov     eax, ecx
       add     eax, DWORD PTR d
       mov     DWORD PTR a, eax
```

As you can see, the statement that operates on variables whose sizes are all the same uses fewer instructions than the one that mixes operand sizes in the expression.

When using different-sized integers in an expression, it's also important to note that not all CPUs support all operand sizes equally efficiently. While it makes sense that using an integer size larger than the CPU's general-purpose integer registers will produce inefficient code, it might not be quite as obvious that using *smaller* integer values can be inefficient as well. Many RISC CPUs work only on operands that are exactly the same size as the general-purpose registers. Smaller operands must first be zero-extended or sign-extended to the size of a general-purpose register prior to any calculations involving those values. Even on CISC processors, such as the 80x86, that have hardware support for different sizes of integers, using certain sizes can be more expensive. For example, under 32-bit operating systems, instructions that manipulate 16-bit operands require an extra *opcode prefix byte* and are therefore larger than instructions that operate on 8-bit or 32-bit operands.

7.4.2 Floating-Point/Real Variables

Like integers, many HLLs provide multiple floating-point variable sizes. Most languages provide at least two different sizes: a 32-bit single-precision floating-point format and a 64-bit double-precision floating-point format, based on the IEEE 754 floating-point standard. A few languages provide 80-bit floating-point variables (Swift is a good example), based on Intel's 80-bit extended-precision floating-point format, but that usage is increasingly rare. The later ARM processors support quad-precision floating-point arithmetic (128-bit); some variants of GCC support a `_float128` type that uses quad-precision arithmetic.

Different floating-point formats trade off space and performance for precision. Calculations involving smaller floating-point formats are usually quicker than calculations involving the larger formats. However, you give up precision to achieve improved performance and size savings (see Chapter 4 of *WGC1* for the details).

As with expressions involving integer arithmetic, you should avoid mixing different-sized floating-point operands in an expression. The CPU (or FPU) must convert all floating-point values to the same format before using them. This can involve additional instructions (consuming more memory) and additional time. Therefore, you should try to use the same floating-point types throughout an expression wherever possible.

Conversion between integer and floating-point formats is another expensive operation you should avoid. Modern HLLs attempt to keep variables' values in registers as much as possible. Unfortunately, on some modern CPUs it's impossible to move data between the integer and floating-point registers without first copying that data to memory (which is expensive, because memory is slow). Furthermore, conversion between integer and floating-point numbers often involves several specialized instructions, all of which consume time and memory. Whenever possible, avoid these conversions.

7.4.3 Character Variables

Standard character data in most modern HLLs consumes 1 byte per character. On CPUs that support byte addressing, such as the Intel 80x86 processor, a compiler can reserve a single byte of storage for each character variable and efficiently access that character variable in memory. Some RISC CPUs, however, cannot access data in memory except in 32-bit chunks (or another size other than 8 bits).

For CPUs that cannot address individual bytes in memory, HLL compilers usually reserve 32 bits for a character variable and use only the LO byte of that double-word variable for the character data. Because few programs have a large number of scalar character variables,[7] the amount of space wasted is hardly an issue in most systems. However, if you have an unpacked array of characters, then the wasted space can become significant. We'll return to this issue in Chapter 8.

Modern programming languages support the Unicode character set. Unicode characters can require between 1 and 4 bytes of memory to hold the character's data value (depending on the underlying encoding, such as UTF-8, UTF-16, or UTF-32). As time passes, Unicode will likely replace the ASCII character set for most character- and string-oriented operations except in those programs that require high-performance random access to characters within strings (where Unicode performance suffers).

7.4.4 Boolean Variables

A Boolean variable requires only a single bit to represent the two values `true` or `false`. HLLs usually reserve the smallest amount of memory possible

7. *Scalar*, in this context, means "not an array of characters."

for these variables (a byte on machines that support byte addressing, and a larger amount of memory on those CPUs that can address only 16-bit, 32-bit, or 64-bit memory values). However, this isn't always the case. Some languages (like FORTRAN) allow you to create multibyte Boolean variables (for example, the FORTRAN `LOGICAL*4` data type).

Some languages (early versions of C/C++, for example) don't support an explicit Boolean data type. Instead, they use an integer data type to represent Boolean values. Those C/C++ implementations use `0` and nonzero to represent `false` and `true`, respectively. In such languages, you get to choose the size of your Boolean variables by choosing the size of the integer you use to hold them. For example, in a typical older 32-bit implementation of the C/C++ languages, you can define 1-byte, 2-byte, or 4-byte Boolean values as shown in Table 7-2.[8]

Table 7-2: Defining Boolean Value Sizes

C integer data type	Size of Boolean object
`char`	1 byte
`short int`	2 bytes
`long int`	4 bytes

Some languages, under certain circumstances, use only a single bit of storage for a Boolean variable when that variable is a field of a record or an element of an array. We'll return to this discussion in Chapters 8–11 when considering composite data structures.

7.5 Variable Addresses and High-Level Languages

The organization, class, and type of variables in your programs can affect the efficiency of the code that a compiler produces. Additionally, issues like the order of declaration, the size of the object, and the placement of the object in memory can have a big impact on the running time of your programs. This section describes how you can organize your variable declarations to produce efficient code.

As for immediate constants encoded in machine instructions, many CPUs provide specialized addressing modes that access memory more efficiently than other, more general, addressing modes. Just as you can reduce the size and improve the speed of your programs by carefully selecting the constants you use, you can make your programs more efficient by carefully choosing how you declare variables. Whereas with constants you're primarily concerned with their values, with variables you must consider the address in memory where the compiler places them.

8. Assuming, of course, that your C/C++ compiler uses 16-bit integers for `short` integers and 32-bit integers for `long` integers.

The 80x86 is a typical example of a CISC processor that provides multiple address sizes. When running on a modern 32- or 64-bit operating system like macOS, Linux, or Windows, the 80x86 CPU supports three address sizes: 0 bit, 8 bit, and 32 bit. The 80x86 uses 0-bit displacements for register-indirect addressing modes. We'll ignore the 0-bit displacement addressing mode for now because 80x86 compilers generally don't use it to access variables you explicitly declare in your code. The 8-bit and 32-bit displacement addressing modes are the more interesting ones for the current discussion.

7.5.1 Allocating Storage for Global and Static Variables

The 32-bit displacement is, perhaps, the easiest to understand. Variables you declare in your program, which the compiler allocates in memory rather than in a register, have to appear somewhere in memory. On most 32-bit processors, the address bus is 32 bits wide, so it takes a 32-bit address to access a variable at an arbitrary location in memory. An instruction that encodes this 32-bit address can access any memory variable. The 80x86 provides the *displacement-only* addressing mode, whose effective address is exactly the 32-bit constant embedded in the instruction.

A problem with 32-bit addresses (one that gets even worse as we move to 64-bit processors with a 64-bit address) is that the address winds up consuming the largest portion of the instruction's encoding. Certain forms of the displacement-only addressing mode on the 80x86, for example, have a 1-byte opcode and a 4-byte address. Therefore, 80 percent of the instruction's size is consumed by the address. Were the 64-bit variants of the 80x86 (x86-64) to actually encode a 64-bit absolute address as part of the instruction, the instruction would be 9 bytes long and consume nearly 90 percent of the instruction's bytes. To avoid this, the x86-64 modified the displacement-only addressing mode. It no longer encodes the absolute address in memory as part of the instruction; instead, it encodes a signed 32-bit offset (±2 billion bytes) into the instruction.

On typical RISC processors, the situation is even worse. Because the instructions are uniformly 32 bits long on typical RISC CPUs, you cannot encode a 32-bit address as part of the instruction. In order to access a variable at an arbitrary 32- or 64-bit address in memory, you need to load the 32- or 64-bit address of that variable into a register and then use the register-indirect addressing mode to access it. For a 32-bit address, this could require three 32-bit instructions, as Figure 7-2 demonstrates; that's expensive in terms of both speed and space. It gets even more expensive with 64-bit addresses.

Because RISC CPUs don't run horribly slower than CISC processors, compilers rarely generate code this bad. In reality, programs running on RISC CPUs often keep base addresses to blocks of objects in registers, so they can efficiently access variables in those blocks using short offsets from the base register. But how do compilers deal with arbitrary addresses in memory?

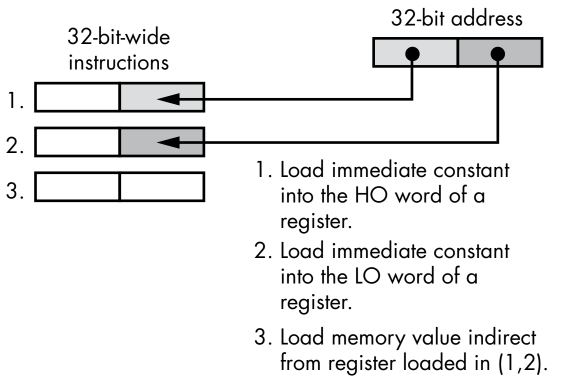

32-bit address
32-bit-wide instructions
1.
2.
3.
1. Load immediate constant into the HO word of a register.
2. Load immediate constant into the LO word of a register.
3. Load memory value indirect from register loaded in (1,2).

Figure 7-2: RISC CPU access of an absolute address

7.5.2 Using Automatic Variables to Reduce Offset Sizes

One way to avoid large instruction sizes with large displacements is to use an addressing mode with a smaller displacement. The 80x86 (and x86-64), for example, provide an 8-bit displacement form for the base-plus-indexed addressing mode. This form allows you to access data at an offset of –128 through +127 bytes around a base address contained in a register. RISC processors have similar features, although the number of displacement bits is usually larger, allowing a greater range of addresses.

By pointing a register at some base address in memory and placing your variables near that base address, you can use the shorter forms of these instructions so your program will be smaller and run faster. This isn't too difficult if you're working in assembly language and you have direct access to the CPU's registers. However, if you're working in an HLL you may not have direct access to the CPU's registers, and even if you did, you probably couldn't convince the compiler to allocate your variables at convenient addresses. How do you take advantage of this small-displacement addressing mode in your HLL programs? The answer is that you don't explicitly specify the use of this addressing mode; the compiler does it for you automatically.

Consider the following trivial function in Pascal:

```
function trivial( i:integer; j:integer ):integer;
var
    k:integer;
begin

    k := i + j;
    trivial := k;

end;
```

Upon entry into this function, the compiled code constructs an activation record (sometimes called a *stack frame*). An activation record, as you saw earlier in the chapter, is a data structure in memory where the system keeps the local data associated with a function or procedure. The activation record includes parameter data, automatic variables, the return address, temporary variables that the compiler allocates, and machine state information (for example, saved register values). The runtime system allocates storage for an activation record on the fly and, in fact, two different calls to the procedure or function may place the activation record at different addresses in memory. In order to access the data in an activation record, most HLLs point a register (usually called the *frame pointer*) at the activation record, and then the procedure or function references automatic variables and parameters at some offset from this frame pointer. Unless you have many automatic variables and parameters, or your automatic variables and parameters are quite large, these variables generally appear in memory at an offset near the base address. This means that the CPU can use a small offset when referencing variables near the base address held in the frame pointer. In the Pascal example given earlier, parameters `i` and `j` and the local variable `k` would most likely be within a few bytes of the frame pointer's address, so the compiler can encode these instructions using a small displacement rather than a large displacement. If your compiler allocates local variables and parameters in an activation record, all you have to do is arrange your variables in the activation record so that they appear near its base address. But how do you do that?

Construction of an activation record begins in the code that calls a procedure. The caller places the parameter data (if any) in the activation record. Then the execution of an assembly language `call` (or equivalent) instruction adds the return address to the activation record. At this point, construction of the activation record continues within the procedure itself. The procedure copies the register values and other important state information and then makes room in the activation record for local variables. The procedure must also update the frame-pointer register (such as EBP on the 80x86, or RBP on the x86-64) so that it points at the base address of the activation record.

To see what a typical activation record looks like, consider the following HLA procedure declaration:

```
procedure ARDemo( i:uns32; j:int32; k:dword ); @nodisplay;
var
    a:int32;
    r:real32;
    c:char;
    b:boolean;
    w:word;
begin ARDemo;
    .
    .
    .
end ARDemo;
```

Whenever an HLA program calls this `ARDemo` procedure, it builds the activation record by pushing the data for the parameters onto the stack in the order they appear in the parameter list, from left to right. Therefore, the calling code first pushes the value for the `i` parameter, then for the `j` parameter, and finally for the `k` parameter. After pushing the parameters, the program calls the `ARDemo` procedure. Immediately upon entry into the procedure, the stack contains these four items, arranged as shown in Figure 7-3, assuming the stack grows from high-memory addresses to low-memory addresses (as it does on most processors).

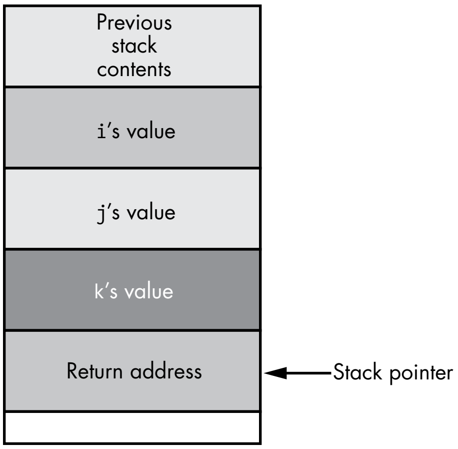


Figure 7-3: Stack organization immediately upon entry into `ARDemo`

The first few instructions in `ARDemo` push the current value of the frame-pointer register (such as EBP on the 32-bit 80x86, or RBP on the x86-64) onto the stack and then copy the value of the stack pointer (ESP/RSP on the 80x86/x86-64) into the frame pointer. Next, the code drops the stack pointer down in memory to make room for the local variables. This produces the stack organization shown in Figure 7-4 on the 80x86 CPU.

To access objects in the activation record, you must use offsets from the frame-pointer register (EBP in Figure 7-4) to the desired object.

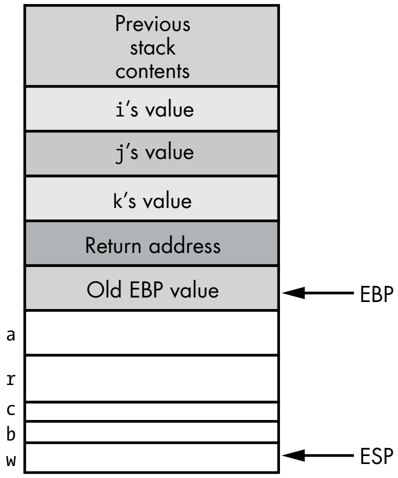


Figure 7-4: Activation record for `ARDemo` *(32-bit 80x86)*

The two items of immediate interest are the parameters and the local variables. As Figure 7-5 shows, you can access the parameters at positive offsets from the frame-pointer register, and the local variables at negative offsets from the frame-pointer register.

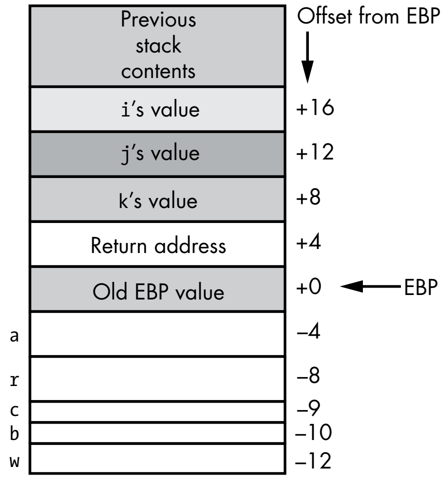


Figure 7-5: Offsets of objects in the `ARDemo` *activation record on the 32-bit 80x86*

Intel specifically reserves the EBP/RBP (base-pointer register) to point at the base of the activation record. Therefore, compilers typically use this register as the frame-pointer register when allocating activation records on the stack. Some compilers instead attempt to use the 80x86 ESP/RSP (stack pointer) register to point to the activation record because this reduces the number of instructions in the program. Whether the compiler uses EBP/RBP, ESP/RSP, or some other register as the frame pointer, the bottom line is that the compiler typically points some register at the activation record, and most of the local variables and parameters are near the activation record's base address.

As you can see in Figure 7-5, all the local variables and parameters in the `ARDemo` procedure are within 127 bytes of the frame-pointer register (EBP). This means that on the 80x86 CPU, an instruction that references one of these variables or parameters will be able to encode the offset from EBP using a single byte. As mentioned earlier, because of the way the program builds the activation record, parameters appear at positive offsets from the frame-pointer register, and local variables appear at negative offsets from the frame-pointer register.

For procedures that have only a few parameters and local variables, the CPU will be able to access all parameters and local variables using a small offset (that is, 8 bits on the 80x86, some possibly larger value on various RISC processors). Consider, however, the following C/C++ function:

```
int BigLocals( int i, int j )
{
    int array[256];
    int k;
        .
        .
        .
}
```

The activation record for this function on the 32-bit 80x86 appears in Figure 7-6.

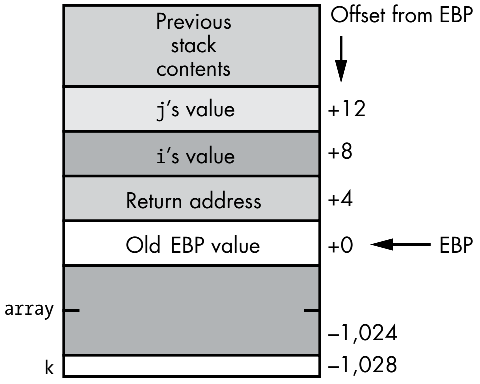


Figure 7-6: Activation record for `BigLocals()` *function*

NOTE *One difference between this activation record and the ones for the Pascal and HLA functions is that C pushes its parameters on the stack in the reverse order (that is, it pushes the last parameter first and the first parameter last). This difference, however, does not impact our discussion at all.*

The important thing to note in Figure 7-6 is that the local variables `array` and `k` have large negative offsets. With offsets of –1,024 and –1,028, the displacements from EBP to `array` and `k` are well outside the range that the compiler can encode into a single byte on the 80x86. Therefore, the compiler has no choice but to encode these displacements using a 32-bit value. Of course, this makes accessing these local variables in the function quite a bit more expensive.

Nothing can be done about the array variable in this example (no matter where you put it, the offset to the base address of the array will be at least 1,024 bytes from the activation record's base address). However, consider the activation record in Figure 7-7.

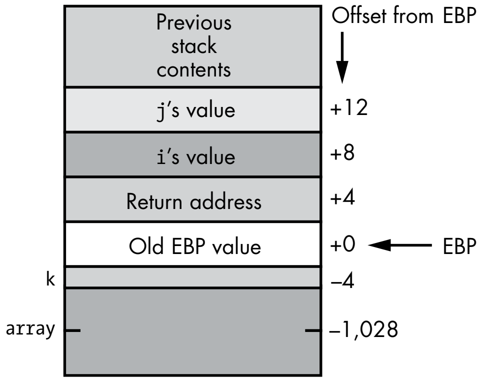


Figure 7-7: Another possible activation record layout for the `BigLocals()` function

The compiler has rearranged the local variables in this activation record. Although it still takes a 32-bit displacement to access the `array` variable, accessing `k` now uses an 8-bit displacement (on the 32-bit 80x86) because `k`'s offset is −4. You can produce these offsets with the following code:

```
int BigLocals( int i, int j );
{
    int k;
    int array[256];
        .
        .
        .
}
```

In theory, rearranging the order of the variables in the activation record isn't terribly difficult for a compiler to do, so you'd expect the compiler to make this modification so that it can access as many local variables as possible using small displacements. In practice, not all compilers actually do this optimization, for various technical and practical reasons (specifically, it can break some poorly written code that makes assumptions about the placement of variables in the activation record).

If you want to ensure that the maximum number of local variables in your procedure have the smallest possible displacements, the solution is trivial: declare all your 1-byte variables first, your 2-byte variables second, your 4-byte variables next, and so on, up to the largest local variable in your function. Generally, though, you're probably more interested in reducing the size of the maximum number of instructions in your function rather than reducing the size of the offsets required by the maximum number of variables in your function. For example, if you have 128 1-byte variables and you declare these variables first, you'll need only a 1-byte displacement if you access them. However, if you never access these variables, the fact that they have a 1-byte displacement rather than a 4-byte displacement saves you nothing. The only time you save any space is when you actually access that variable's value in memory via some machine instruction that uses a 1-byte rather than a 4-byte displacement. Therefore, to reduce your function's object code size, you want to maximize the number of instructions that use a small displacement. If you refer to a 100-byte array far more often than any other variable in your function, you're probably better off declaring that array first, even if it leaves only 28 bytes of storage (on the 80x86) for other variables that will use the shorter displacement.

RISC processors typically use a signed 12-bit or 16-bit offset to access fields of the activation record. Thus, you have more latitude with your declarations when using a RISC chip (which is good, because when you do exceed the 12-bit or 16-bit limitation, accessing a local variable gets really expensive). Unless you're declaring one or more arrays that consume more than 2,048 (12 bits) or 32,768 bytes (combined), the typical compiler for a RISC chip will generate decent code.

This same argument applies to parameters as well as local variables. However, it's rare to find code passing a large data structure (by value) to a function because of the expense involved.

7.5.3 Allocating Storage for Intermediate Variables

Intermediate variables are local to one procedure/function but global to another. You'll find them in block-structured languages—like Free Pascal, Delphi, Ada, Modula-2, Swift, and HLA—that support nested procedures. Consider the following example program in Swift:

```
import Cocoa
import Foundation

var globalVariable = 2

func procOne()
{
    var intermediateVariable = 2;

    func procTwo()
    {
        let localVariable =
            intermediateVariable + globalVariable
        print( localVariable )
    }
    procTwo()
}

procOne()
```

Note that nested procedures can access variables found in the main program (that is, global variables) as well as variables found in procedures containing the nested procedure (that is, the intermediate variables). As you've seen, local variable access is inexpensive compared to global variable access (because you always have to use a larger offset to access global objects within a procedure). Intermediate variable access, as is done in the `procTwo` procedure, is expensive. The difference between local and global variable accesses is the size of the offset/displacement coded into the instruction, with local variables typically using a shorter offset than is possible for global objects. Intermediate accesses, on the other hand, typically require several machine instructions. This makes the instruction sequence that accesses an intermediate variable several times slower and several times larger than accessing a local (or even global) variable.

The problem with using intermediate variables is that the compiler must maintain either a linked list of activation records or a table of pointers to the activation records (a *display*) in order to reference intermediate objects. To access an intermediate variable, the `procTwo` procedure must either follow a chain of links (there would be only one link in this example) or do a table lookup in order to get a pointer to `procOne`'s activation record. Worse still, maintaining the display of this linked list of pointers isn't exactly cheap. The work needed to maintain these objects has to be done on every procedure/function entry and exit, even when the procedure or function doesn't access any intermediate variables on a particular call. Although there are, arguably, some software engineering benefits to using intermediate variables (having to do with information hiding) versus a global variable, keep in mind that accessing intermediate objects is expensive.

7.5.4 Allocating Storage for Dynamic Variables and Pointers

Pointer access in an HLL provides another opportunity for optimization in your code. Pointers can be expensive to use but, under certain circumstances, they can actually make your programs more efficient by reducing displacement sizes.

A pointer is simply a memory variable whose value is the address of some other memory object (therefore, pointers are the same size as an address on the machine). Because most modern CPUs support indirection only via a machine register, indirectly accessing an object is typically a two-step process: first the code has to load the value of the pointer variable into a register, and then it has to refer (indirectly) to the object through that register.

Consider the following C/C++ code fragment:

```
int *pi;
    .
    .
    .
i = *pi;    // Assume pi is initialized with a
            // reasonable address at this point.
```

Here is the corresponding 80x86/HLA assembly code:

```
pi: pointer to int32;
    .
    .
    .
mov( pi, ebx );     // Again, assume pi has
mov( [ebx], eax );  // been properly initialized
mov( eax, i );
```

Had pi been a regular variable rather than pointer object, this code could have dispensed with the mov([ebx], eax); instruction and simply moved pi directly into eax. Therefore, the use of this pointer variable has both increased the program's size and reduced the execution speed by inserting an extra instruction into the code sequence that the compiler generates.

However, if you indirectly refer to an object several times in close succession, the compiler may be able to reuse the pointer value it has loaded into the register, amortizing the cost of the extra instruction across several different instructions. Consider the following C/C++ code sequence:

```
int *pi;
    .
    .   // Assume code in this area
    .   // initializes pi appropriately.
    .
*pi = i;
*pi = *pi + 2;
*pi = *pi + *pi;
printf( "pi = %d\n", *pi );
```

Here's the corresponding 80x86/HLA code:

```
pi: pointer to int32;
    .
    . // Assume code in this area
    . // initializes pi appropriately.
    .
// Extra instruction that we need to initialize EBX

mov( pi, ebx );

mov( i, eax );
mov( eax, [ebx] );  //  This code can clearly be optimized,
mov( [ebx], eax );  //  but we'll ignore that fact for the
add( 2, eax );      //  sake of the discussion here.
mov( eax, [ebx] );
mov( [ebx], eax );
add( [ebx], eax );
mov( eax [ebx] );
stdout.put( "pi = ", (type int32 [ebx]), nl );
```

This code loads the actual pointer value into EBX only once. From that point forward, the code will simply use the pointer value contained in EBX to reference the object at which pi is pointing. Of course, any compiler that can do this optimization can probably eliminate five redundant memory loads and stores from this assembly language sequence, but let's assume they're not redundant for the time being. Because the code didn't have to reload EBX with the value of pi every time it wanted to access the object at which pi points, there's only one instruction of overhead (mov(pi, ebx);) amortized across six instructions. That's not too bad at all.

Indeed, you could make a good argument that this code is more optimal than accessing a local or global variable directly. An instruction of the form

```
mov([ebx],eax);
```

encodes a 0-bit displacement. Therefore, this move instruction is only 2 bytes long rather than 3, 5, or even 6 bytes long. If `pi` is a local variable, it's quite possible that the original instruction that copies `pi` into EBX is only 3 bytes long (a 2-byte opcode and a 1-byte displacement). Because instructions of the form `mov([ebx],eax);` are only 2 bytes long, it only takes three instructions to "break even" on the byte count using indirection rather than an 8-bit displacement. After the third instruction that references whatever `pi` points at, the code involving the pointer is actually shorter.

You can even use indirection to provide efficient access to a block of global variables. As noted earlier, the compiler generally cannot determine the address of a global object while it's compiling your program. Therefore, it has to assume the worst case and allow for the largest possible displacement/offset when generating machine code to access a global variable. Of course, you've just seen that you can reduce the size of the displacement value from 32 bits down to 0 bits by using a pointer to the object rather than accessing the object directly. Therefore, you could take the address of the global object (with the C/C++ `&` operator, for example) and then use indirection to access the variable. The problem with this approach is that it requires a register (a precious commodity on any processor, but especially on the 32-bit 80x86, which has only six general-purpose registers to utilize). If you access the same variable many times in rapid succession, this 0-bit displacement trick can make your code more efficient. However, it's somewhat rare to access the same variable repeatedly in a short sequence of code without also needing to access several other variables. This means the compiler may have to flush the pointer from the register and reload the pointer value later, reducing the efficiency of this approach. If you're working on a RISC chip or x86-64 with many registers, you can probably employ this trick to your advantage. On a processor with a limited number of registers, though, you won't be able to employ it as often.

7.5.5 Using Records/Structures to Reduce Instruction Offset Sizes

There's also a trick you can use to access several variables with a single pointer: put all those variables into a structure and then use the structure's address. By accessing the fields of the structure via the pointer, you can get away with using smaller instructions to access the objects. This works almost exactly as you've seen for activation records (indeed, activation records are, literally, records that the program references indirectly via the frame-pointer register). About the only difference between accessing objects indirectly in a user-defined record/structure and accessing objects in the activation record is that most compilers won't let you refer to fields in a user structure/record using negative offsets. Therefore, you're limited to about half the number of bytes that are normally accessible in an activation record. For example, on the 80x86 you can access the object at offset 0 from a pointer using a 0-bit displacement and objects at offsets 1 through +127 using a single-byte displacement. Consider the following C/C++ example that uses this trick:

```
typedef struct
{
    int i;
    int j;
    char *s;
    char name[20];
    short t;
} vars;

static vars v;
vars *pv = &v;  // Initialize pv with the address of v.
        .
        .
        .
    pv->i = 0;
    pv->j = 5;
    pv->s = pv->name;
    pv->t = 0;
    strcpy( pv->name, "Write Great Code!" );
        .
        .
        .
```

A well-designed compiler will load the value of `pv` into a register exactly once for this code fragment. Because all the fields of the `vars` structure are within 127 bytes of the base address of the structure in memory, an 80x86 compiler can emit a sequence of instructions that require only 1-byte offsets, even though the `v` variable itself is a static/global object. Note, by the way, that the first field in the `vars` structure is special. Because this is at offset 0 in the structure, you can use a 0-bit displacement when accessing this field. Therefore, it's a good idea to put your most-often-referenced field first in a structure if you're going to refer to that structure indirectly.

Using indirection in your code does come at a cost. On a limited-register CPU such as the 32-bit 80x86, using this approach will tie up a register for a while, and that may effectively cause the compiler to generate worse code. If the compiler must constantly reload the register with the address of the structure in memory, the savings from this technique evaporate rather quickly. Tricks such as this one vary in effectiveness across different processors (and different compilers for the same processor), so be sure to look at the code your compiler generates to verify that a trick is actually saving rather than costing you something.

7.5.6 Storing Variables in Machine Registers

While we're on the subject of registers, it's worthwhile to point out one other 0-bit displacement way to access variables in your programs: by keeping them in machine registers. Machine registers are always the most efficient place to store variables and parameters. Unfortunately, only in assembly language and, to a limited extent, C/C++, do you have any control over whether the compiler should keep a variable or parameter in a register. In some respects, this is not bad. Good compilers do a much better job of register allocation than the casual programmer does. However, an expert programmer can do a better job of register allocation than a compiler, because the expert programmer understands the data the program will be processing and the frequency of access to a particular memory location. (And of course, the expert programmer can first look at what the compiler is doing, whereas the compiler doesn't have the benefit of seeing what the programmer has done.)

Some languages, such as Delphi, provide limited support for programmer-directed register allocation. In particular, the Delphi compiler allows you to tell it to pass the first three (ordinal) parameters for a function or procedure in the EAX, EDX, and ECX registers. This option is known as the *fastcall calling convention*, and several C/C++ compilers support it as well.

In Delphi and certain other languages, opting for the fastcall parameter passing convention is the only control you get. The C/C++ language, however, provides the `register` keyword, a storage specifier (much like the `const`, `static`, and `auto` keywords) that tells the compiler that the programmer expects to use the variable frequently so the compiler should attempt to keep it in a register. Note that the compiler can also choose to ignore the `register` keyword (in which case it reserves variable storage using automatic allocation). Many compilers ignore the `register` keyword altogether because the compiler's authors assume, somewhat arrogantly, that they can do a better job of register allocation than any programmer. Of course, on some register-starved machines such as the 32-bit 80x86, there are so few registers to work with that it might not even be possible to allocate a variable to a register throughout the execution of some function. Nevertheless, some compilers do respect the programmer's wishes and *will* allocate a few variables in registers if you request that they do so.

Most RISC compilers reserve several registers for passing parameters and several registers for local variables. Therefore, it's a good idea (if possible) to place the parameters you access most frequently first in the parameter declaration because they're probably the ones the compiler would allocate in a register.[9] The same is true for local variable declarations. Always declare frequently used local variables first, because many compilers may allocate those (ordinal) variables in registers.

One problem with compiler register allocation is that it is static. That is, the compiler determines which variables to place in registers based on an analysis of your source code during compilation, not during runtime. Compilers often make assumptions (which are usually correct) like "this function references variable `xyz` far more often than any other variable, so it's a good candidate for a register variable." Indeed, by placing the variable in a register, the compiler will certainly reduce the size of the program.

9. Many optimizing compilers are smart enough to choose which variables they keep in registers based on how the program uses those variables.

However, it could also be the case that all those references to xyz sit in code that rarely, if ever, executes. Although the compiler might save some space (by emitting smaller instructions to access registers rather than memory), the code won't run appreciably faster. After all, if the code rarely or never executes, then making that code run faster does not contribute much to the program's execution time. On the other hand, it's also quite possible to bury a single reference to some variable in a deeply nested loop that executes many times. With only one reference in the entire function, the compiler's optimizer may overlook the fact that the executing program references the variable frequently. Although compilers have gotten smarter about handling variables inside loops, the fact is, no compiler can predict how many times an arbitrary loop will execute at runtime. Human beings are much better at predicting this sort of behavior (or, at least, measuring it with a profiler) and thus are best positioned to make good decisions about variable allocation in registers.

7.6 Variable Alignment in Memory

On many processors (particularly RISC), there is another efficiency concern you must take into consideration. Many modern processors will not let you access data at an arbitrary address in memory. Instead, all accesses must take place on some native boundary (usually 4 bytes) that the CPU supports.[10] Even when a CISC processor allows memory accesses at arbitrary byte boundaries, it's often more efficient to access primitive objects (bytes, words, and double words) on a boundary that is a multiple of the object's size (see Figure 7-8).

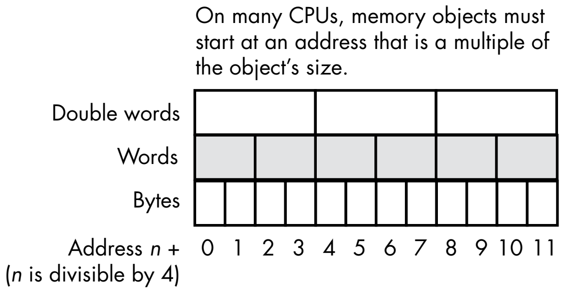

On many CPUs, memory objects must start at an address that is a multiple of the object's size.
Double words
Words
Bytes
Address n + 0 1 2 3 4 5 6 7 8 9 10 11
(n is divisible by 4)

Figure 7-8: Variable alignment in memory

If the CPU supports unaligned accesses—that is, if the CPU allows you to access a memory object on a boundary that is not a multiple of the object's primitive size—then you should be able to pack the variables into the activation record. This way, you would obtain the maximum number of variables having a short offset. However, because unaligned accesses are

10. The PowerPC supports unaligned memory access, albeit with reduced performance. Earlier versions of the ARM (earlier than ARMv6-A) did not allow unaligned memory accesses at all.

sometimes slower than aligned accesses, many optimizing compilers insert *padding bytes* into the activation record in order to ensure that all variables are aligned on a reasonable boundary for their native size (see Figure 7-9). This trades off slightly better performance for a slightly larger program.

```
char oneByte ;
short twoBytes ;
char oneByte2 ;
int fourBytes ;
```

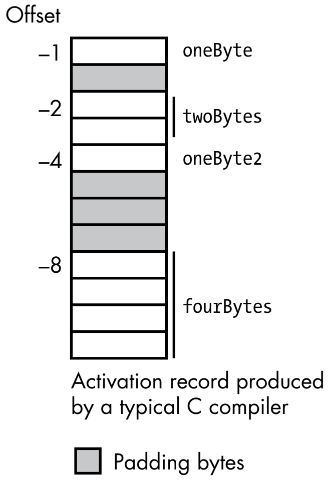


Figure 7-9: Padding bytes in an activation record

However, if you put all your double-word declarations first, your word declarations second, your byte declarations third, and your array/structure declarations last, you can improve both the speed and size of your code. The compiler usually ensures that the first local variable you declare appears at a reasonable boundary (typically a double-word boundary). By declaring all your double-word variables first, you ensure that they all appear at an address that is a multiple of 4 (because compilers usually allocate adjacent variables in your declarations in adjacent locations in memory). The first word-sized object you declare will also appear at an address that is a multiple of 4—and that means its address is also a multiple of 2 (which is best for word accesses). By declaring all your word variables together, you ensure that each one appears at an address that is a multiple of 2. On processors that allow byte access to memory, the placement of the byte variables (with respect to efficiently accessing the byte data) is irrelevant. By declaring all your local byte variables last in a procedure or function, you generally ensure that such declarations do not impact the performance of the double-word and word variables you also use in the function. Figure 7-10 shows what a typical activation record will look like if you declare your variables as in the following function:

```
int someFunction( void )
{
    int d1;   // Assume ints are 32-bit objects
    int d2;
    int d3;
    short w1; // Assume shorts are 16-bit objects
    short w2;
    char b1;  // Assume chars are 8-bit objects
    char b2;
    char b3;
        .
        .
        .
} // end someFunction
```

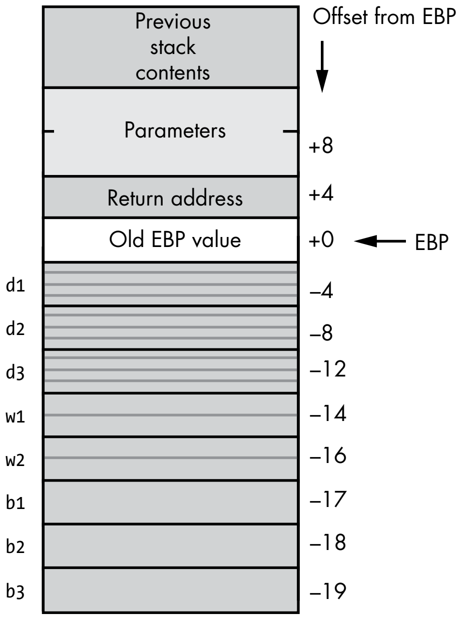


Figure 7-10: Aligned variables in an activation record (32-bit 80x86)

Note how all the double-word variables (d1, d2, and d3) begin at addresses that are multiples of 4 (–4, –8, and –12). Also, notice how all the word-sized variables (w1 and w2) begin at addresses that are multiples of 2 (–14 and –16). The byte variables (b1, b2, and b3) begin at arbitrary addresses in memory (both even and odd addresses).

Now consider the following function, which has arbitrary (unordered) variable declarations, and the corresponding activation record shown in Figure 7-11:

```
int someFunction2( void )
{
    char  b1; // Assume chars are 8-bit objects
    int   d1; // Assume ints are 32-bit objects
    short w1; // Assume shorts are 16-bit objects
    int   d2;
    short w2;
    char  b2;
    int   d3;
    char  b3;
        .
        .
        .
} // end someFunction2
```

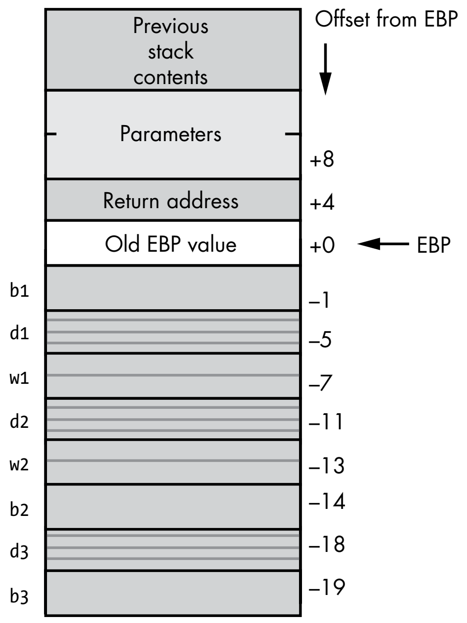


Figure 7-11: Unaligned variables in an activation record (32-bit 80x86)

As you can see, every variable except the byte variables appears at an address that is inappropriate for the object. On processors that allow memory accesses at arbitrary addresses, it may take more time to access a variable that is not aligned on an appropriate address boundary.

Some processors don't allow a program to access an object at an unaligned address. Most RISC processors, for example, can't access memory except at 32-bit address boundaries. To access a short or byte value, some RISC processors require the software to read a 32-bit value and extract the 16-bit or 8-bit value (that is, the CPU forces the software to treat bytes and words as packed data). The extra instructions and memory accesses needed to pack and unpack this data reduce the speed of memory access by a considerable amount (two or more instructions—usually more—may be needed to fetch a byte or word from memory). Writing data to memory is even worse because the CPU must first fetch the data from memory, merge the new data with the old data, and then write the result back to memory. Therefore, most RISC compilers won't create an activation record similar to the one in Figure 7-11. Instead, they'll add padding bytes so that every memory object begins at an address boundary that is a multiple of 4 bytes (see Figure 7-12).

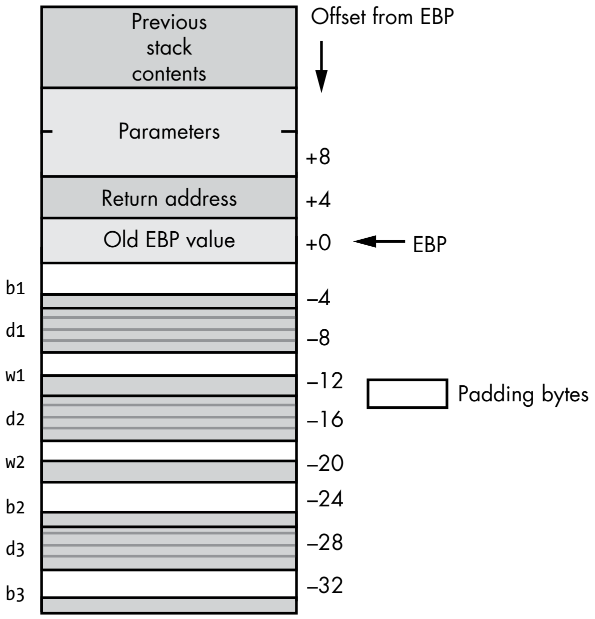


Figure 7-12: RISC compilers force aligned access by adding padding bytes.

Notice in Figure 7-12 that all of the variables are at addresses that are multiples of 32 bits. Therefore, a RISC processor has no problems accessing any of these variables. The cost, of course, is that the activation record is quite a bit larger (the local variables consume 32 bytes rather than 19 bytes).

Although the example in Figure 7-12 is typical for 32-bit RISC-based compilers, that's not to suggest that compilers for CISC CPUs don't do this as well. Many compilers for the 80x86, for example, also build this activation record in order to improve the performance of the code the compiler generates. Although declaring your variables in a misaligned fashion may not slow down your code on a CISC CPU, it may use additional memory.

Of course, if you work in assembly language, it's generally up to you to declare your variables in a manner that is appropriate or efficient for your particular processor. In HLA (on the 80x86), for example, the following two procedure declarations result in the activation records shown in Figures 7-10, 7-11, and 7-12.

```
procedure someFunction; @nodisplay; @noalignstack;
var
    d1  :dword;
    d2  :dword;
    d3  :dword;
    w1  :word;
    w2  :word;
    b1  :byte;
    b2  :byte;
    b3  :byte;
begin someFunction;
        .
        .
        .
end someFunction;


procedure someFunction2; @nodisplay; @noalignstack;
var
    b1  :byte;
    d1  :dword;
    w1  :word;
    d2  :dword;
    w2  :word;
    b2  :byte;
    d3  :dword;
    b3  :byte;
begin someFunction2;
        .
        .
        .
end someFunction2;


procedure someFunction3; @nodisplay; @noalignstack;
var
    // HLA align directive forces alignment of the next declaration.

    align(4);
    b1  :byte;
    align(4);
    d1  :dword;
    align(4);
    w1  :word;
    align(4);
    d2  :dword;
    align(4);
    w2  :word;
    align(4);
    b2  :byte;
    align(4);
    d3  :dword;
    align(4);
    b3  :byte;
begin someFunction3;
        .
        .
        .
end someFunction3;
```

HLA procedures `someFunction` and `someFunction3` will produce the fastest-running code on any 80x86 processor because all variables are aligned on an appropriate boundary; HLA procedures `someFunction` and `someFunction2` will produce the most compact activation records on an 80x86 CPU, because there is no padding between variables in the activation record. If you're working in assembly language on a RISC CPU, you'll probably want to choose the equivalent of `someFunction` or `someFunction3` to make it easier to access the variables in memory.

7.6.1 Records and Alignment

Records/structures in HLLs also have alignment issues that should concern you. Recently, CPU manufacturers have been promoting *application binary interface (ABI)* standards to promote interoperability between different programming languages and their implementations. Although not all languages and compilers adhere to these suggestions, many of the newer compilers do. Among other things, these ABI specifications describe how the compilers should organize fields within a record or structure object in memory. Although the rules vary by CPU, one that applies to most ABIs is that a compiler should align a record/structure field at an offset that is a multiple of the object's size. If two adjacent fields in the record or structure have different sizes, and the placement of the first field in the structure would cause the second field to appear at an offset that is not a multiple of that second field's native size, then the compiler will insert some padding bytes to push the second field to a higher offset that is appropriate for that second object's size.

In actual practice, ABIs for different CPUs and OSes have minor differences based on the CPUs' ability to access objects at different addresses in memory. Intel, for example, suggests that compiler writers align bytes at any offset, words at even offsets, and everything else at offsets that are a multiple of 4. Some ABIs recommend placing 64-bit objects at 8-byte boundaries within a record. The x86-64 SSE and AVX instructions require 16- and 32-byte alignment for 128-bit and 256-bit data values. Some CPUs, which have a difficult time accessing objects smaller than 32 bits, may suggest a minimum alignment of 32 bits for all objects in a record/structure. The rules vary depending on the CPU and whether the manufacturer wants to promote faster-executing code (the usual case) or smaller data structures.

If you are writing code for a single CPU (such as an Intel-based PC) with a single compiler, learn that compiler's rules for padding fields and adjust your declarations for maximum performance and minimal waste. However, if you ever need to compile your code using several different compilers, particularly compilers for several different CPUs, following one set of rules will work fine on one machine and produce less efficient code on several others. Fortunately, there are some rules that can help reduce the inefficiencies created by recompiling for a different ABI.

From a performance/memory usage standpoint, the best solution is the same rule we saw earlier for activation records: when declaring fields in a record, group all like-sized objects together and put all the larger (scalar) objects first and the smaller objects last in the record/structure. This scheme produces the least amount of waste (padding bytes) and provides the highest performance across most of the existing ABIs. The only drawback to this approach is that you have to organize the fields by their native size rather than by their logical relationship to one another. However, because all fields of a record/structure are logically related insofar as they are all members of that same record/structure, this problem isn't as bad as employing this organization for all of a particular function's local variables.

Many programmers try to add padding fields themselves to a structure. For example, the following type of code is common in the Linux kernel and other bits and pieces of overly hacked software:

```
 typedef struct IveAligned
{
    char byteValue;
    char padding0[3];
    int  dwordValue;
    short wordValue;
    char padding1[2];
    unsigned long dwordValue2;
         .
         .
         .
};
```

The `padding0` and `padding1` fields in this structure were added to manually align the `dwordValue` and `dwordValue2` fields at offsets that are even multiples of 4.

While this padding is not unreasonable, if you're using a compiler that doesn't automatically align the fields, remember that an attempt to compile this code on a different machine can produce unexpected results. For example, if a compiler aligns all fields on a 32-bit boundary, regardless of size, then this structure declaration will consume two extra double words to hold the two `paddingX` arrays. This winds up wasting space for no good reason. Keep this fact in mind if you decide to manually add the padding fields.

Many compilers that automatically align fields in a structure provide an option to turn off this feature. This is particularly true for compilers generating code for CPUs where the alignment is optional and the compiler does it only to achieve a slight performance boost. If you're going to manually add padding fields to your record/structure, you need to specify this option so that the compiler doesn't realign the fields after you've manually aligned them.

In theory, a compiler is free to rearrange the offsets of local variables within an activation record. However, it would be extremely rare for a compiler to rearrange the fields of a user-defined record or structure. Too many external programs and data structures depend on the fields of a record appearing in the same order as they are declared. This is particularly true when passing record/structure data between code written in two separate languages (for example, when calling a function written in assembly language) or when dumping record data directly to a disk file.

In assembly language, the amount of effort needed to align fields varies from pure manual labor to a rich set of features capable of automatically handling almost any ABI. Some (low-end) assemblers don't even provide record or structure data types. In such systems, the assembly programmer has to manually specify the offsets into a record structure (typically by declaring, as constants, the numeric offsets into the structure). Other assemblers (for example, NASM) provide macros that automatically generate the equates for you. In these systems, the programmer has to manually provide padding fields to align certain fields on a given boundary. Some assemblers, such as MASM, provide simple alignment facilities. You may specify the value `1`, `2`, or `4` when declaring a `struct` in MASM and the assembler will align all fields on either the alignment value you specify or at an offset that is a multiple of the object's size, whichever is smaller, by automatically adding padding bytes to the structure. Also, note that MASM adds a sufficient number of padding bytes to the end of the structure so that the whole structure's length is a multiple of the alignment size. Consider the following `struct` declaration in MASM:

```
Student  struct  2
score    word    ?   ; offset:0
id       byte    ?   ; offset 2 + 1 byte of padding
year     dword   ?   ; offset 4
id2      byte    ?   ; offset:8
Student  ends
```

In this example, MASM will add an extra byte of padding to the end of the structure so that its length is a multiple of 2 bytes.

MASM also lets you control the alignment of individual fields within a structure by using the `align` directive. The following structure declaration is equivalent to the current example (note the absence of the alignment value operand in the `struct` operand field):

```
Student  struct
score    word    ?   ; offset:0
id       byte    ?   ; offset 2
         align   2   ; Injects 1 byte of padding.
year     dword   ?   ; offset 4
id2      byte    ?   ; offset:8
         align   2   ; Adds 1 byte of padding to the end of the struct.
Student  ends
```

The default field alignment for MASM structures is unaligned. That is, a field begins at the next available offset within the structure, regardless of the field's (and the previous field's) size.

The High-Level Assembly (HLA) language probably provides the greatest control (both automatic and manual) over record field alignment. As with MASM, the default record alignment is unaligned. Also as with MASM, you can use HLA's `align` directive to manually align fields in an HLA record. The following is the HLA version of the previous MASM example:

```
type
    Student :record
        score :word;
        id    :byte;
        align(2);
        year  :dword;
        id2   :byte;
        align(2);
    endrecord;
```

HLA also lets you specify an automatic alignment for all fields in a record. For example:

```
type
    Student :record[2]  // This tells HLA to align all
                        // fields on a word boundary
        score :word;
        id    :byte;
        year  :dword;
        id2   :byte;
    endrecord;
```

There is a subtle difference between this HLA record and the earlier MASM structure (with automatic alignment). Remember, when you specify a directive of the form `Student struct 2`, MASM aligns all fields on a boundary that is a multiple of 2 or a multiple of the object's size, *whichever is smaller*. HLA, on the other hand, will always align all fields on a 2-byte boundary using this declaration, even if the field is a byte.

The ability to force field alignment to a minimum size is a nice feature if you're working with data structures generated on a different machine (or compiler) that forces this kind of alignment. However, this type of alignment can unnecessarily waste space in a record for certain declarations if you only want the fields to be aligned on their natural boundaries (which is what MASM does). Fortunately, HLA provides another syntax for record declarations that lets you specify both the maximum and minimum alignment that HLA will apply to a field:

```
recordID: record[ maxAlign : minAlign ]
    <<fields>>
endrecord;
```

The `maxAlign` item specifies the largest alignment that HLA will use within the record. HLA will align any object whose native size is larger than maxAlign on a boundary of maxAlign bytes. Similarly, HLA will align any object whose size is smaller than minAlign on a boundary of at least minAlign bytes. HLA will align objects whose native size is between minAlign and maxAlign on a boundary that is a multiple of that object's size. The following HLA and MASM record/structure declarations are equivalent:

Here's the MASM code:

```
Student  struct  4
score    word    ?   ; offset:0
id       byte    ?   ; offset 2

    ; 1 byte of padding appears here

year     dword   ?   ; offset 4
id2      byte    ?   ; offset:8

    ; 3 padding bytes appear here

courses  dword   ?   ; offset:12
Student  ends
```

Here's the HLA code:

```
type
    //  Align on 4-byte offset, or object's size, whichever
    //  is the smaller of the two. Also, make sure that the
    //  entire record is a multiple of 4 bytes long.

    Student  :record[4:1]
        score   :word;
        id      :byte;
        year    :dword;
        id2     :byte;
      courses   :dword;
    endrecord;
```

Although few HLLs provide facilities within the language's design to control the alignment of fields within records (or other data structures), many compilers provide extensions to those languages, in the form of compiler pragmas, that let programmers specify default variable and field alignment. Because few languages have standards for this, you'll have to check your particular compiler's reference manual (note that C++11 is one of the few languages that provides alignment support). Although such extensions are nonstandard, they are often quite useful, especially when you're linking code compiled by different languages or trying to squeeze the last bit of performance out of a system.

7.7 For More Information

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* 2nd ed. Essex, UK: Pearson Education Limited, 1986.

Barrett, William, and John Couch. *Compiler Construction: Theory and Practice.* Chicago: SRA, 1986.

Dershem, Herbert, and Michael Jipping. *Programming Languages, Structures and Models.* Belmont, CA: Wadsworth, 1990.

Duntemann, Jeff. *Assembly Language Step-by-Step.* 3rd ed. Indianapolis: Wiley, 2009.

Fraser, Christopher, and David Hansen. *A Retargetable C Compiler: Design and Implementation.* Boston: Addison-Wesley Professional, 1995.

Ghezzi, Carlo, and Jehdi Jazayeri. *Programming Language Concepts.* 3rd ed. New York: Wiley, 2008.

Hoxey, Steve, Faraydon Karim, Bill Hay, and Hank Warren, eds. *The PowerPC Compiler Writer's Guide.* Palo Alto, CA: Warthman Associates for IBM, 1996.

Hyde, Randall. *The Art of Assembly Language.* 2nd ed. San Francisco: No Starch Press, 2010.

Intel. "Intel 64 and IA-32 Architectures Software Developer Manuals." Updated November 11, 2019. *https://software.intel.com/en-us/articles/intel-sdm.*

Ledgard, Henry, and Michael Marcotty. *The Programming Language Landscape.* Chicago: SRA, 1986.

Louden, Kenneth C. *Compiler Construction: Principles and Practice.* Boston: Cengage, 1997.

Louden, Kenneth C., and Kenneth A. Lambert. *Programming Languages: Principles and Practice.* 3rd ed. Boston: Course Technology, 2012.

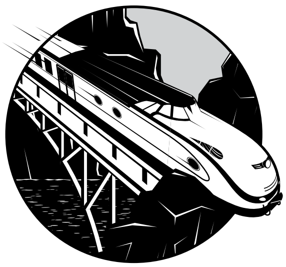
Parsons, Thomas W. *Introduction to Compiler Construction.* New York: W. H. Freeman, 1992.

Pratt, Terrence W., and Marvin V. Zelkowitz. *Programming Languages, Design and Implementation.* 4th ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Sebesta, Robert. *Concepts of Programming Languages.* 11th ed. Boston: Pearson, 2016.

8

ARRAY DATA TYPES



High-level language abstractions hide how the machine deals with *composite data types* (a complex data type composed of smaller data objects). Although these abstractions are often convenient, if you don't understand the details behind them you might inadvertently use a construct that generates unnecessary code or runs slower than need be. In this chapter, we'll take a look at one of the most important composite data types: the array. We'll consider the following topics:

- The definition of an array
- How to declare arrays in various languages
- How arrays are represented in memory
- Accessing elements of arrays
- Declaring, representing, and accessing multidimensional arrays
- Row-major and column-major multidimensional array access

- Dynamic versus static arrays
- How using arrays can impact the performance and size of your applications

Arrays are very common in modern applications, so you should have a solid understanding of how programs implement and use them in memory in order to write great code. This chapter will give you the foundation you need to use arrays more efficiently in your programs.

8.1 Arrays

Arrays are one of the most common composite (or *aggregate*) data types, yet few programmers fully grasp how they operate. Once they understand how arrays work at the machine level, programmers frequently view them from a completely different perspective.

Abstractly, an array is an aggregate data type whose members (elements) are all of the same type. To select a member from the array, you specify the member's array index with an integer (or with some value whose underlying representation is an integer, such as character, enumerated, and Boolean types). In this chapter, we'll assume that all of the integer indices of an array are numerically contiguous. That is, if both `x` and `y` are valid indices of the array, and if `x` < `y`, then all `i` such that `x` < `i` < `y` are also valid indices. We'll also typically assume that array elements occupy contiguous locations in memory, although this is not required by the general definition of an array. An array with five elements appears in memory as shown in Figure 8-1.

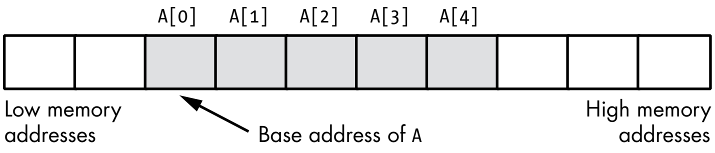


Figure 8-1: Array layout in memory

The *base address* of an array is the address of its first element that occupies the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, and so on. Note that the indices do not have to start at `0`; they can start with any number as long as they are contiguous. However, discussing array access is easier if the first index is `0`, so arrays in this chapter begin at index `0` unless otherwise indicated.

Whenever you apply the indexing operator to an array, the result is the unique array element specified by that index. For example, `A[i]` chooses the `i`th element from array `A`.

8.1.1 Array Declarations

Array declarations are very similar across many high-level languages (HLLs). This section presents examples in several languages.

8.1.1.1 Declaring Arrays in C, C++, and Java

C, C++, and Java all let you declare an array by specifying the total number of elements. The syntax for an array declaration in these languages is:

```
data_type  array_name [ number_of_elements ];
```

Here are some sample C/C++ array declarations:

```
char CharArray[ 128 ];
int intArray[ 8 ];
unsigned char ByteArray[ 10 ];
int *PtrArray[ 4 ];
```

If you declare these arrays as automatic variables, C/C++ initializes them with whatever bit patterns happen to exist in memory. If, on the other hand, you declare these arrays as static objects, C/C++ initializes each array element with `0`. If you want to initialize an array yourself, you can use the following C/C++ syntax:

```
data_type array_name[ number_of_elements ] = {element_list};
```

Here's a typical example:

```
int intArray[8] = {0,1,2,3,4,5,6,7};
```

The C/C++ compiler stores these initial array values in the object code file, and the operating system will load these values into the memory locations associated with `intArray` when it loads the program into memory. To see how this works, consider the following short C/C++ program:

```
#include <stdio.h>
static int intArray[8] = {1,2,3,4,5,6,7,8};
static int array2[8];

int main( int argc, char **argv )
{
    int i;
    for( i=0; i<8; ++i )
    {
        array2[i] = intArray[i];
    }
    for( i=7; i>= 0; --i )
    {
        printf( "%d\n", array2[i] );
    }
    return 0;
}
```

Microsoft's Visual C++ compiler emits the following 80x86 assembly code for the two array declarations:

```
_DATA    SEGMENT
intArray DD      01H
         DD      02H
         DD      03H
         DD      04H
         DD      05H
         DD      06H
         DD      07H
         DD      08H
$SG6842  DB      '%d', 0aH, 00H
_DATA    ENDS
_BSS     SEGMENT
_array2  DD      08H DUP (?)
_BSS     ENDS
```

Each DD ("define double word") statement reserves 4 bytes of storage, and the operand specifies their initial value when the OS loads the program into memory. The intArray declaration appears in the _DATA segment, which in the Microsoft memory model can contain initialized data. The array2 variable, on the other hand, is declared inside the _BSS segment, where MSVC++ places uninitialized variables (the ? character in the operand field tells the assembler that the data is uninitialized; the 8 dup (?) operand tells the assembler to duplicate the declaration eight times). When the OS loads the _BSS segment into memory, it simply zeros out all the memory associated with that segment. In both the initialized and uninitialized cases, the compiler allocates all eight elements of these arrays in sequential memory locations.

8.1.1.2 Declaring Arrays in HLA

HLA's array declaration syntax takes the following form, which is semantically equivalent to the C/C++ declaration:

```
array_name : data_type [ number_of_elements ];
```

Here are some examples of HLA array declarations that allocate storage for uninitialized arrays (the second example assumes you have defined the integer data type in a type section of the HLA program):

```
static

 // Character array with elements 0..127.

 CharArray: char[128];

 // "integer" array with elements 0..7.

 IntArray: integer[ 8 ];
 // Byte array with elements 0..9.

 ByteArray: byte[10];

 // Double word array with elements 0..3.

 PtrArray: dword[4];
```

You can also initialize the array elements using declarations like the following:

```
RealArray: real32[8] :=
    [ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 ];

IntegerAry: integer[8] :=
    [ 8, 9, 10, 11, 12, 13, 14, 15 ];
```

Both of these definitions create arrays with eight elements. The first definition initializes each 4-byte `real32` array element with one of the values in the range `0.0..7.0`. The second declaration initializes each `integer` array element with one of the values in the range `8..15`.

8.1.1.3 Declaring Arrays in Pascal/Delphi

Pascal/Delphi uses the following syntax to declare an array:

```
array_name : array[ lower_bound..upper_bound ] of data_type;
```

As in the previous examples, *`array_name`* is the identifier and *`data_type`* is the type of each element in this array. In Pascal/Delphi (unlike C/C++, Java, and HLA) you specify the upper and lower bounds of the array rather than the array's size. The following are typical array declarations in Pascal:

```
type
    ptrToChar = ^char;
var
    CharArray: array[ 0..127 ] of char;   // 128 elements
    IntArray: array[ 0..7 ] of integer;   // 8 elements
    ByteArray: array[ 0..9 ] of char;     // 10 elements
    PtrArray: array[ 0..3 ] of ptrToChar; // 4 elements
```

Although these Pascal examples start their indices at `0`, Pascal does not require it. The following is a perfectly valid array declaration in Pascal:

```
var
   ProfitsByYear : array[ 1998..2028 ] of real; // 31 elements
```

The program that declares this array would use indices `1998..2028` when accessing elements of this array, not `0..30`.

Many Pascal compilers provide a very useful feature to help you locate defects in your programs. Whenever you access an element of an array, these compilers automatically insert code that will verify that the array index is within the bounds specified by the declaration. This extra code will stop the program if the index is out of range. For example, if an index into `ProfitsByYear` is outside the range `1998..2028`, the program will abort with an error.[1]

8.1.1.4 Declaring Arrays in Swift

Array declarations in Swift are a bit different from those of other C-based languages. Swift array declarations take one of the following two (equivalent) forms:

```
var data_type array_name = Array<element_type>()
var data_type array_name = [element_type]()
```

Unlike in other languages, arrays in Swift are purely dynamic. You don't normally specify the number of elements when you first create the array; instead, you add elements to the array as needed using functions like `append()`. If you want to predeclare an array with some number of elements, you can use a special array constructor form as follows:

```
var data_type array_name = Array<element_type>( repeating: initial_value, count: elements )
```

In this example, *`initial_value`* is a value of type *`element_type`* and *`elements`* is the number of elements to create in the array. For example, the following Swift code creates two arrays of 100 `Int` values, each initialized to `0`:

```
var intArray = Array<Int>( repeating: 0, count: 100 )
var intArray2 = [Int]( repeating: 0, count: 100 )
```

Note that you can still extend the size of this array (for example, by using the `append()` function); because Swift arrays are dynamic, their size can grow or shrink at runtime.

It is also possible to create a Swift array with initial values:

```
var intArray = [1, 2, 3]
var strArray = ["str1", "str2", "str3"]
```

Swift, like Pascal, checks the validity of array indices at runtime. Swift will raise an exception if you attempt to access an array element that doesn't exist.

1. Many Pascal compilers provide an option to turn off the bounds-checking feature once your program is fully tested; doing so improves the efficiency of the resulting program.

8.1.1.5 Declaring Arrays with Noninteger Index Values

Generally, array indices are integer values, although some languages allow other *ordinal types* (data types that use an underlying integer representation). For example, Pascal allows `char` and `boolean` array indices. In Pascal, it's perfectly reasonable and useful to declare an array as follows:

```
alphaCnt : array[ 'A'..'Z' ] of integer;
```

You access elements of `alphaCnt` using a character expression as the array index. For example, consider the following Pascal code, which initializes each element of `alphaCnt` to `0`:

```
for ch := 'A' to 'Z' do
    alphaCnt[ ch ] := 0;
```

Assembly language and C/C++ treat most ordinal values as special instances of integer values, so they are legal array indices. Most implementations of BASIC allow a floating-point number as an array index, although BASIC always truncates the value to an integer before using it as an index.

NOTE *BASIC allows you to use floating-point values as array indices because the original language did not provide support for integer expressions; it provided only real and string values.*

8.1.2 Array Representation in Memory

Abstractly, an array is a collection of variables that you access using an index. Semantically, you can define an array any way you please, as long as it maps distinct indices to distinct objects in memory and always maps the same index to the same object. In practice, however, most languages utilize a few common algorithms that provide efficient access to the array data.

The most common implementation of arrays is to store elements in consecutive memory locations. As noted earlier, most programming languages store the first element of an array at a low memory address and then store the following elements in successively higher memory locations.

Consider the following C program:

```
#include <stdio.h>

static char array[8] = {0,1,2,3,4,5,6,7};



int main( void )
{

    printf( "%d\n", array[0] );
}
```

Here is the corresponding PowerPC assembly code that GCC emits for it:

```
        .align 2
_array:
        .byte   0   ; Note that the assembler stores the byte
        .byte   1   ; values on successive lines into
        .byte   2   ; contiguous memory locations.
        .byte   3
        .byte   4
        .byte   5
        .byte   6
        .byte   7
```

The number of bytes an array consumes is the number of elements multiplied by the number of bytes per element. In the previous example, each array element is a single byte, so the array consumes the same number of bytes as it has elements. However, for arrays with larger elements, the entire array size (in bytes) is correspondingly larger. Consider the following C code:

```
#include <stdio.h>

static int array[8] = {0,0,0,0,0,0,0,1};


int main( void )
{
    printf( "%d\n", array[0] );
}
```

Here's the corresponding GCC assembly language output:

```
        .align 2
_array:
        .long   0
        .long   0
        .long   0
        .long   0
        .long   0
        .long   0
        .long   0
        .long   1
```

Many languages also add a few bytes of padding at the end of an array so that the array's total length will be a multiple of a convenient value like 2 or 4 (making it easy to compute indices into the array using shifts or to add extra padding bytes for the next object in memory; see Chapter 3 of *WGC1* for details). However, a program must not count on those extra padding bytes, because they may or may not be present. Some compilers always put them in, others never do, and still others put them in depending on the type of object that immediately follows the array in memory.

Many optimizing compilers try to start an array at a memory address that is a multiple of a common size like 2, 4, or 8 bytes. Effectively, this adds padding bytes before the beginning of the array or, if you prefer to think of it this way, after the previous object in memory (see Figure 8-2).

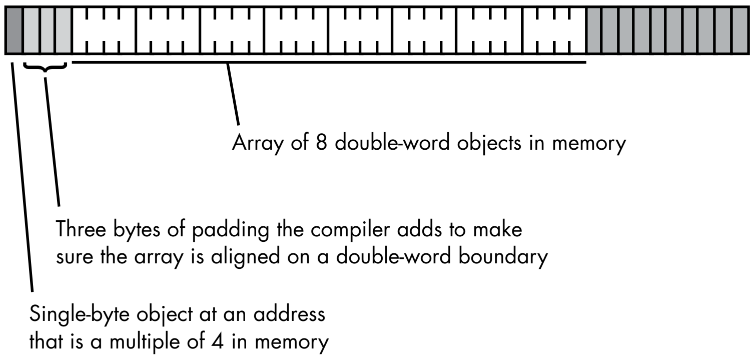

Array of 8 double-word objects in memory
Three bytes of padding the compiler adds to make sure the array is aligned on a double-word boundary
Single-byte object at an address that is a multiple of 4 in memory

Figure 8-2: Adding padding bytes before an array

On machines that do not support byte-addressable memory, compilers that attempt to place the first element of an array on an easily accessed boundary will allocate storage for an array on whatever boundary the machine supports. In the previous example, notice that the `.align 2` directive precedes the `_array` declaration. In Gas syntax, the `.align` directive tells the assembler to adjust the memory address of the next object declared in the source file so that it starts at an address that is a multiple of some power (specified by `.align`'s operand) of 2. In this example, `.align 2` tells the assembler to align the first element of `_array` on an address boundary that is a multiple of 4 (that is, 2^2).

If the size of each array element is less than the minimum-sized memory object the CPU supports, the compiler implementer has two options:

1. Allocate the smallest accessible memory object for each element of the array.
2. Pack multiple array elements into a single memory cell.

Option 1 has the advantage of being fast, but it wastes memory because each array element carries some extra storage that it doesn't need. The following C example allocates storage for an array whose element size is 5 bytes, where each element is a structure object consisting of a 4-byte `long` object and a 1-byte `char` object (we'll look at C structures in Chapter 11).

```
#include <stdio.h>

typedef struct
{
    long a;
    char b;
} FiveBytes;

static FiveBytes shortArray[2] = {{2,3}, {4,5}};
```

```
int main( void )
{
    printf( "%ld\n", shortArray[0].a );
}
```

When GCC compiles this code to run on a PowerPC processor, which requires double-word alignment for `long` objects, the compiler automatically inserts 3 bytes of padding between each element:

```
.data
        .align 2   ; Ensure that _shortArray begins on an
                   ; address boundary that is a multiple
                   ; of 4.
_shortArray:
        .long   2  ; shortArray[0].a
        .byte   3  ; shortArray[0].b
        .space  3  ; Padding, to align next element to 4 bytes
        .long   4  ; shortArray[1].a
        .byte   5  ; shortArray[1].b
        .space  3  ; Padding, at end of array.
```

Option 2 is compact but slower, as it requires extra instructions to pack and unpack data when accessing array elements. Compilers on such machines often provide an option that lets you specify whether you want the data packed or unpacked so you can choose between space and speed.

Keep in mind that if you're working on a byte-addressable machine (like the 80x86), then you probably don't have to worry about this issue. However, if you're using an HLL and your code might run on a different machine at some point in the future, you should choose an array organization that is efficient on all machines (that is, an organization that pads each element of the array with extra bytes).

8.1.3 Swift Array Implementation

Although the examples so far have included arrays in Swift, Swift arrays have a different implementation. First of all, Swift arrays are an opaque type[2] based on `struct` objects, rather than just a collection of elements in memory. Swift doesn't guarantee that array elements appear in continuous memory locations; thus, you can't assume that object elements and certain other element types in a Swift array are stored contiguously. As a workaround, Swift provides the `ContiguousArray` type specification. To guarantee that array elements appear in contiguous memory locations, you can specify `ContiguousArray` rather than `Array` when declaring array variables in Swift, like so:

```
var data_type array_name = ContiguousArray<element_type>()
```

The internal implementation of a Swift array is a structure containing a count (current number of array elements), a capacity (current number

2. Opaque types have an implementation that is not visible to the programmer.

of allocated array elements), and a pointer to the storage holding the array elements. Because Swift arrays are an opaque type, this implementation could change at any time; however, somewhere in the structure there will be a pointer to the actual array data in memory.

Swift allocates storage for arrays dynamically, which means you'll never see the array storage embedded in the object code file that the Swift compiler produces (unless the Swift language definition changes to support statically allocated arrays). You can increase the size of an array by appending elements to it, but if you attempt to extend it beyond its current capacity, the Swift runtime system may need to dynamically relocate an array object. For performance reasons, Swift uses an exponential allocation scheme: whenever you append a value to an array that would exceed its capacity, the Swift runtime system will allocate twice (or some other constant) as much storage as the current capacity, copy the data from the current array buffer to the new buffer, and then point the array's internal pointer at the new block. One important aspect of this process is that you can never assume that the pointer to the array's data remains static or that the array's data remains in the same buffer location in memory—at different points in time, the array could appear in different locations in memory.

8.1.4 Accessing Elements of an Array

If you allocate all the storage for an array in contiguous memory locations, and the first index of the array is `0`, then accessing an element of a one-dimensional array is simple. You can compute the address of any given element of an array using the following formula:

```
Element_Address = Base_Address + index * Element_Size
```

`Element_Size` is the number of bytes that each array element occupies. Therefore, if the array contains elements of type `byte`, the *`Element_Size`* field is `1` and the math is very easy. If each element of the array is a `word` (or other 2-byte type), then `Element_Size` is `2`, and so on. Consider the following Pascal array declaration:

```
var  SixteenInts : array[0..15] of integer;
```

To access an element of the `SixteenInts` array on a byte-addressable machine, assuming 4-byte integers, you'd use this calculation:

```
Element_Address = AddressOf( SixteenInts ) + index * 4
```

In HLA assembly language (where you'd have to do this calculation manually rather than having the compiler do it for you), you could use code like this to access array element `SixteenInts[index]`:

```
mov( index, ebx );
mov( SixteenInts[ ebx*4 ], eax );
```

To see this in action, consider the following Pascal/Delphi program and the resulting 32-bit 80x86 code (which I obtained by disassembling the *.exe* output from the Delphi compiler and pasting the result back into the original Pascal code):

```
program x(input,output);
var
    i :integer;
    sixteenInts :array[0..15] of integer;

    function changei(i:integer):integer;
    begin
        changei := 15 - i;
    end;

    // changei         proc    near
    //                 mov     edx, 0Fh
    //                 sub     edx, eax
    //                 mov     eax, edx
    //                 retn
    // changei         endp


begin

    for i:= 0 to 15 do
        sixteenInts[ changei(i) ] := i;

    //                 xor     ebx, ebx
    //
    // loc_403AA7:
    //                 mov     eax, ebx
    //                 call    changei
    //
    // Note the use of the scaled-index addressing mode
    // to multiply the array index by 4 prior to accessing
    // elements of the array:
    //
    //                 mov     ds:sixteenInts[eax*4], ebx
    //                 inc     ebx
    //                 cmp     ebx, 10h
    //                 jnz     short loc_403AA7

end.
```

As in the HLA example, the Delphi compiler uses the 80x86 scaled-index addressing mode to multiply the index into the array by the element size (4 bytes). The 80x86 provides four different scaling values for the scaled-index addressing mode: 1, 2, 4, or 8 bytes. If the array's element size is not one of these four values, the machine code must explicitly multiply the index by the array element's size. The following Delphi/Pascal code (and corresponding 80x86 code from the disassembly) demonstrates this process using a record that has 9 bytes of active data (Delphi rounds this up to the next multiple of 4 bytes, so it actually allocates 12 bytes for each element of the array of records):

```
program x(input,output);
type
    NineBytes=
        record
            FourBytes       :integer;
            FourMoreBytes   :integer;
            OneByte         :char;
        end;

var
    i               :integer;
    NineByteArray   :array[0..15] of NineBytes;

    function changei(i:integer):integer;
    begin
        changei := 15 - i;
    end;

    // changei         proc    near
    //                 mov     edx, 0Fh
    //                 sub     edx, eax
    //                 mov     eax, edx
    //                 retn
    // changei         endp


begin

    for i:= 0 to 15 do
        NineByteArray[ changei(i) ].FourBytes := i;

//                  xor     ebx, ebx
//
//  loc_403AA7:
//                  mov     eax, ebx
//                  call    changei
//
//          // Compute EAX = EAX * 3
//
//                  lea     eax, [eax+eax*2]
//
//          // Actual index used is index*12 ((EAX*3) * 4)
//
//                  mov     ds:NineByteArray[eax*4], ebx
//                  inc     ebx
//                  cmp     ebx, 10h
//                  jnz     short loc_403AA7


end.
```

Microsoft's C/C++ compilers emit comparable code (also allocating 12 bytes for each element of the array of records).

8.1.5 Padding vs. Packing

These Pascal examples reiterate an important point: compilers generally pad each array element to a multiple of 4 bytes, or whatever size is most convenient for the machine's architecture, to improve access to array elements (and record fields) by ensuring that they are always aligned on a reasonable memory boundary. Some compilers give you the option of eliminating the padding at the end of each array element, so that successive array elements immediately follow the previous element in memory. In Pascal/Delphi, for example, you can achieve this by using the `packed` keyword:

```
program x(input,output);

// Note the use of the "packed" keyword.
// This tells Delphi to pack each record
// into 9 consecutive bytes, without
// any padding at the end of the record.

type
    NineBytes=
        packed record
            FourBytes       :integer;
            FourMoreBytes   :integer;
            OneByte         :char;
        end;

var
    i               :integer;
    NineByteArray   :array[0..15] of NineBytes;

    function changei(i:integer):integer;
    begin
        changei := 15 - i;
    end;

    // changei          proc near
    //                  mov     edx, 0Fh
    //                  sub     edx, eax
    //                  mov     eax, edx
    //                  retn
    // changei          endp


begin

    for i:= 0 to 15 do
        NineByteArray[ changei(i) ].FourBytes := i;
```

```
//                xor     ebx, ebx
//
// loc_403AA7:
//                mov     eax, ebx
//                call    changei
//
//      // Compute index (eax) = index * 9
//      // (computed as index = index + index*8):
//
//                lea     eax, [eax+eax*8]
//
//                mov     ds:NineBytes[eax], ebx
//                inc     ebx
//                cmp     ebx, 10h
//                jnz     short loc_403AA7


end.
```

The `packed` reserved word is just a hint to a Pascal compiler. A generic Pascal compiler can choose to ignore it; the Pascal standard does not make any explicit claims about its impact on a compiler's code generation. Delphi uses the `packed` keyword to tell the compiler to pack array (and record) elements on a byte boundary rather than a 4-byte boundary. Other Pascal compilers actually use this keyword to align objects on bit boundaries.

NOTE *See your compiler's documentation for more information about the* `packed` *keyword.*

Few other languages provide a way, within the generic language definition, to pack data into a given boundary. In the C/C++ languages, for example, many compilers provide pragmas or command-line switches to control array element padding, but these facilities are almost always specific to a particular compiler.

In general, choosing between packed and padded array elements (when you have a choice) is usually a tradeoff between speed and space. Packing lets you save a small amount of space for each array element at the cost of slower access to it (for example, when accessing a `dword` object at an odd address in memory). Furthermore, computing the index into an array whose element size is not a convenient multiple of 2 (or, better yet, a power of 2) can require more instructions, which also slows down programs that access elements of such arrays.

Of course, some machine architectures don't allow misaligned data access, so if you're writing portable code that must compile and run on different CPUs, you shouldn't count on the fact that array elements can be tightly packed into memory. Some compilers may not give you this option.

Before closing this discussion, it's worthwhile to emphasize that the best array element sizes are those that are some power of 2. Generally, it takes only a single instruction to multiply any array index by a power of 2 (that single instruction is a shift-left instruction). Consider the following

C program and the assembly output produced by Borland's C++ compiler, which uses arrays that have 32-byte elements:

```
typedef struct
{
    double EightBytes;
    double EightMoreBytes;
    float  SixteenBytes[4];
} PowerOfTwoBytes;

int i;
PowerOfTwoBytes ThirtyTwoBytes[16];

int changei(int i)
{
    return 15 - i;
}


int main( int argc, char **argv )
{
    for( i=0; i<16; ++i )
    {
        ThirtyTwoBytes[ changei(i) ].EightBytes = 0.0;
    }

    // @5:
    //  push      ebx
    //  call      _changei
    //  pop       ecx           // Remove parameter
    //
    // Multiply index (in EAX) by 32.
    // Note that (eax << 5) = eax * 32
    //
    //  shl       eax,5
    //
    // 8 bytes of zeros are the coding for
    // (double) 0.0:
    //
    //  xor       edx,edx
    //  mov       dword ptr [eax+_ThirtyTwoBytes],edx
    //  mov       dword ptr [eax+_ThirtyTwoBytes+4],edx
    //
    // Finish up the for loop here:
    //
    //  inc       dword ptr [esi]   ;ESI points at i.
    // @6:
    //  mov       ebx,dword ptr [esi]
    //  cmp       ebx,16
    //  jl        short @5

    return 0;
}
```

As you can see, the Borland C++ compiler emits a `shl` instruction to multiply the index by 32.

8.1.6 Multidimensional Arrays

A *multidimensional* array is one that lets you select an element of the array using two or more independent index values. A classic example is a two-dimensional data structure (matrix) that tracks product sales versus date. One index into the table could be the date, while the other index could be the product's identification (some integer designation). The element of the array selected by these two indices would be the total sales of that product on a given date. A three-dimensional extension of this example could be sales of products by date and by country. Again, a combination of product value, date value, and country value would address an element in the array to give you the sales of that product within that country on the specified date.

Most CPUs can easily handle one-dimensional arrays using an indexed addressing mode. Unfortunately, there is no magic addressing mode that lets you easily access the elements of multidimensional arrays. That's going to take some work and several machine instructions.

8.1.6.1 Declaring Multidimensional Arrays

An "*m* by *n*" array has `m` × `n` elements and requires `m` × `n` × *`Element_Size`* bytes of storage. With one-dimensional arrays, the syntax is very similar among HLLs. However, their syntax starts to differ with multidimensional arrays.

In C, C++, and Java, you use the following syntax to declare a multidimensional array:

```
data_type array_name [dim1][dim2]...[dimn];
```

For example, here's a three-dimensional array declaration in C/C++:

```
int threeDInts[ 4 ][ 2 ][ 8 ];
```

This example creates an array with 64 elements organized with a depth of 4 by 2 rows by 8 columns. Assuming each `int` object requires 4 bytes, this array consumes 256 bytes of storage.

Pascal's syntax supports two equivalent ways of declaring multidimensional arrays. The following example demonstrates both:

```
var
    threeDInts:
        array[0..3] of array[0..1] of array[0..7] of integer;

    threeDInts2: array[0..3, 0..1, 0..7] of integer;
```

The first Pascal declaration is technically an *array of arrays*, whereas the second declaration is a standard multidimensional array.

Semantically, there are only two major differences in the way different languages handle multidimensional arrays: whether the array declaration specifies the overall size of each array dimension or the upper and lower bounds; and whether the starting index is `0`, `1`, or a user-specified value.

8.1.6.2 Declaring Swift Multidimensional Arrays

Swift doesn't support a native multidimensional array, but rather an array of arrays. For most programming languages, where an array object is strictly the sequence of array elements in memory, an array of arrays and a multidimensional array are the same thing (see the Pascal examples given earlier). However, Swift uses descriptor (`struct`-based) objects to specify an array. Like string descriptors, Swift arrays consist of a data structure that contains various fields (such as capacity, current size, and a pointer to data; see "Swift Array Implementation" on page 234 for more details). When you create an array of arrays, you're actually creating an array of these descriptors, with each pointing at a subarray. Consider the following (equivalent) Swift array-of-arrays declarations and sample program:

```
import Foundation

var a1 = [[Int]]()
var a2 = ContiguousArray<Array<Int>>()
a1.append( [1,2,3] )
a1.append( [4,5,6] )
a2.append( [1,2,3] )
a2.append( [4,5,6] )

print( a1 )
print( a2 )
print( a1[0] )
print( a1[0][1] )
```

Running this program produces the following output:

```
[[1, 2, 3], [4, 5, 6]]
[[1, 2, 3], [4, 5, 6]]
[1, 2, 3]
2
```

This is reasonable—for two-dimensional arrays you'd expect this type of output. However, internally, `a1` and `a2` are one-dimensional arrays with two elements each. Those two elements are array descriptors that themselves point at arrays (each containing three elements in this example). It is unlikely that the six array elements associated with `a2` will appear in contiguous memory locations, even though `a2` is a contiguous array type. The two array descriptors held in `a2` may appear in contiguous memory locations, but that doesn't necessarily carry over to the six data elements at which they collectively point.

Because Swift allocates array storage dynamically, the rows in a two-dimensional array could have differing element counts. Consider the following modification to the previous Swift program:

```
import Foundation

var a2 = ContiguousArray<Array<Int>>()
a2.append( [1,2,3] )
a2.append( [4,5] )

print( a2 )
print( a2[0] )
print( a2[0][1] )
```

Running this program produces the following output:

```
[[1, 2, 3], [4, 5]]
[1, 2, 3]
2
```

Notice how the two rows in the `a2` array have different sizes. This could be useful or a source of defects, depending on what you're trying to accomplish.

One way to get standard multidimensional array storage in Swift is to declare a one-dimensional `ContiguousArray` with sufficient elements for all the elements of the multidimensional array. Then use the row-major (or column-major) functionality to compute the index into the array (see "Implementing Row-Major Ordering" on page 244 and "Implementing Column-Major Ordering" on page 247).

8.1.6.3 Mapping Multidimensional Array Elements to Memory

Now that you've seen how arrays are declared, you need to know how to implement them in memory. The first challenge is storing a multidimensional object into a one-dimensional memory space.

Consider a Pascal array of the following form:

```
A:array[0..3,0..3] of char;
```

This array contains 16 bytes organized as four rows of four characters. You need to map each of the 16 bytes in this array to each of the 16 contiguous bytes in main memory. Figure 8-3 shows one way to do this.

You can map positions within the array grid to memory addresses in different ways, as long as you adhere to two rules:

- No two entries in the array can occupy the same memory location.
- Each element in the array always maps to the same memory location.

Therefore, what you really need is a function with two input parameters (one for a row and one for a column value) that produces an offset into a contiguous block of 16 memory locations.

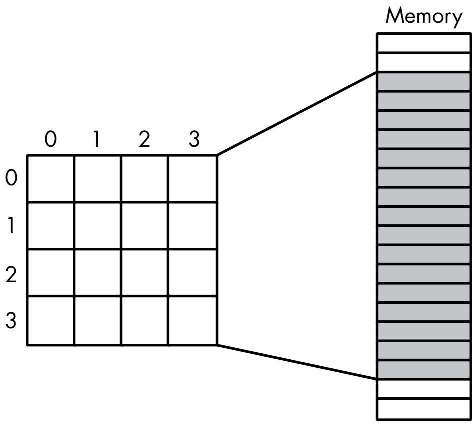


Figure 8-3: Mapping a 4×4 array to sequential memory locations

Now, any old function that satisfies these two constraints will work fine. However, what you really want is a mapping function that can compute efficiently at runtime and works for arrays with any number of dimensions and any bounds on those dimensions. While there are numerous options that fit this bill, most HLLs use one of two organizations: *row-major ordering* and *column-major ordering*.

8.1.6.4 Implementing Row-Major Ordering

Row-major ordering assigns array elements to successive memory locations by moving across the rows and then down the columns. Figure 8-4 demonstrates this mapping for `A[col,row]`.

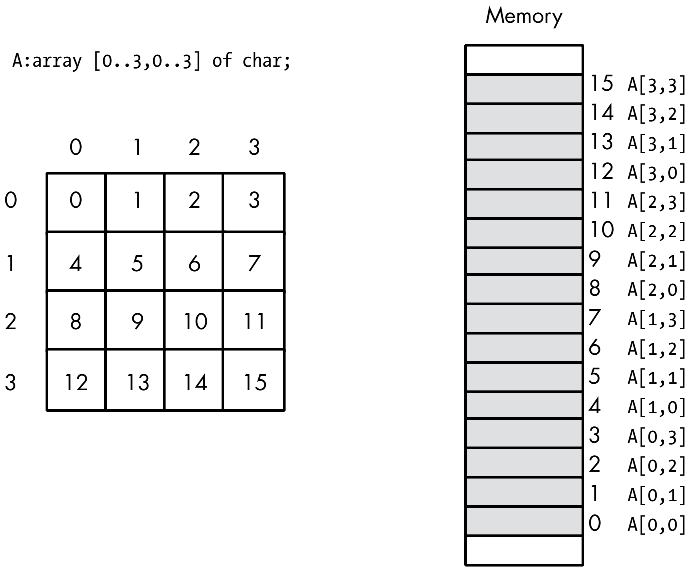


Figure 8-4: Row-major ordering for a 4×4 array

Row-major ordering is the method employed by most high-level programming languages, including Pascal, C/C++/C#, Java, Ada, and Modula-2. It is very easy to implement and easy to use in machine language. The conversion from a two-dimensional structure to a linear sequence is very intuitive. Figure 8-5 provides another view of row-major ordering for a 4×4 array.

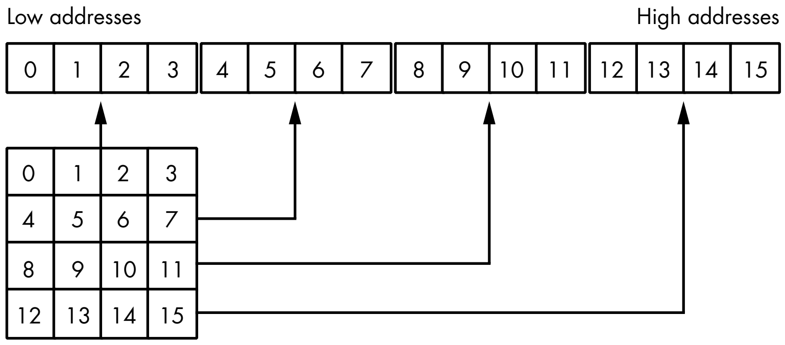


Figure 8-5: Another view of row-major ordering for a 4x4 array

The function that converts the set of multidimensional array indices into a single offset is a slight modification of the formula for computing the address of an element of a one-dimensional array. The generic formula to compute the offset into a two-dimensional row-major-ordered array given an access of the form:

```
array[ colindex ][ rowindex ]
```

is:

```
Element_Address =
    Base_Address +
        (colindex * row_size + rowindex) * Element_Size
```

As usual, `Base_Address` is the address of the first element of the array (`A[0][0]` in this case), and `Element_Size` is the size of an individual element of the array in bytes. `Row_size` is the number of elements in one row of the array (`4`, in this case, because each row has four elements). Assuming `Element_Size` is `1` and `row_size` is `4`, this formula computes the offsets shown in Table 8-1 from the base address.

For a three-dimensional array, the formula to compute the offset into memory is only slightly more complex. Consider a C/C++ array declaration given as follows:

```
someType array[depth_size][col_size][row_size];
```

Table 8-1: Offsets for Two-Dimensional Row-Major-Ordered Array

Column index	Row index	Offset into array
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

If you have an array access similar to `array[`*`depth_index`*`] [`*`col_index`*`] [`*`row_index`*`]`, then the computation that yields the offset into memory is:

```
Address =
    Base +
        ((depth_index * col_size + col_index) *
            row_size + row_index) * Element_Size
```

Again, *`Element_Size`* is the size, in bytes, of a single array element.
For a four-dimensional array, declared in C/C++ as:

```
type A[bounds0] [bounds1] [bounds2] [bounds3];
```

the formula for computing the address of an array element when accessing element `A[i][j][k][m]` is:

```
Address =
    Base +
        (((i * bounds1 + j) * bounds2 + k) * bounds3 + m) *
            Element_Size
```

If you have an *n*-dimensional array declared in C/C++ as follows:

```
dataType array[bn-1][bn-2]...[b0];
```

and you want to access the following element of this array:

```
array[a_{n-1}][a_{n-2}]...[a_1][a_0]
```

then you can compute the address of a particular array element using the following algorithm:

```
Address := $a_{n-1}$
for i := n-2 downto 0 do
    Address := Address * $b_i$ + $a_i$
Address := Base_Address + Address * Element_Size
```

It would be very rare for a compiler to actually execute such a loop in order to compute an array index. There's usually a small number of dimensions and the compiler will unroll the loop, thereby avoiding the overhead of the loop control instructions.

8.1.6.5 Implementing Column-Major Ordering

Column-major ordering, the other common array element address function, is used by FORTRAN, OpenGL, and various dialects of BASIC (such as older versions of Microsoft BASIC) to index arrays. A column-major-ordered array (accessing `A[col,row]`) is organized as shown in Figure 8-6.

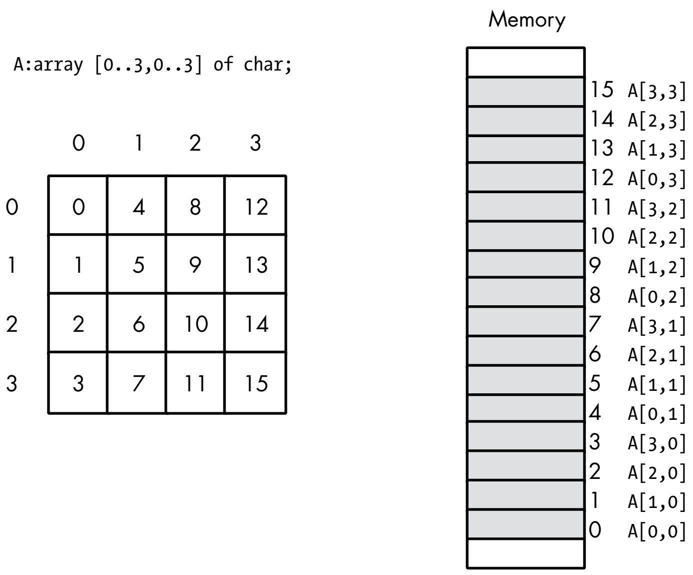


Figure 8-6: Column-major ordering

The formula for computing the address of an array element when using column-major ordering is very similar to that for row-major ordering. The difference is that you reverse the order of the index and size variables in the computation. That is, rather than working from the leftmost index to the rightmost, you operate on the indices from the rightmost toward the leftmost.

For a two-dimensional column-major array:

```
Element_Address =
    Base_Address +
        (rowindex * col_size + colindex) *
            Element_Size
```

For a three-dimensional column-major array:

```
Element_Address =
    Base_Address +
        ((rowindex * col_size + colindex) *
            depth_size + depthindex) *
                Element_Size
```

And so on. Other than using these new formulas, accessing elements of an array using column-major ordering is identical to accessing arrays using row-major ordering.

8.1.6.6 Accessing Elements of a Multidimensional Array

It's so easy to access an element of a multidimensional array in an HLL that many programmers do so without considering the associated costs. In this section, to give you a clearer picture of these costs, we'll look at some of the assembly language sequences compilers commonly generate to access elements of a multidimensional array. Because arrays are one of the more common data structures found in modern applications, and multidimensional arrays are also quite common, compiler designers have put a lot of work into ensuring that they compute array indices as efficiently as possible. Given a declaration such as:

```
int ThreeDInts[ 8 ][ 2 ][ 4 ];
```

and an array reference like the following:

```
ThreeDInts[i][j][k] = n;
```

accessing the array element (using row-major ordering) requires computing the following:

```
Element_Address =
    Base_Address +
        ((i * col_size + j) * // col_size = 2
            row_size + k) *   // row_size = 4
                Element_Size
```

In brute-force assembly code, this might be:

```
intmul( 2, i, ebx );    // EBX = 2*i
add( j, ebx );          // EBX = 2*i + j
intmul( 4, ebx );       // EBX = (2*i + j)*4
add( k, ebx );          // EBX = (2*i + j)*4 + k
mov( n, eax );
mov( eax, ThreeDInts[ebx*4] );  // ThreeDInts[i][j][k] = n; assumes 4-byte ints
```

In practice, however, compiler authors avoid using the 80x86 `intmul` (`imul`) instruction because it is slow. Many different machine idioms can be used to simulate multiplication using a short sequence of addition, shift, and "load effective address" instructions. Most optimizing compilers use sequences that compute the array element address rather than the brute-force code that uses a multiply instruction.

Consider the following C program, which initializes the 16 elements of a 4×4 array:

```
int i, j;
int TwoByTwo[4][4];

int main( int argc, char **argv )
{
    for( j=0; j<4; ++j )
    {
        for( i=0; i<4; ++i )
        {
            TwoByTwo[i][j] = i+j;
        }
    }
    return 0;
}
```

Now consider the assembly code that the Borland C++ v5.0 compiler (an old compiler) emits for the `for` loop in this example:

```
    mov       ecx,offset _i
    mov       ebx,offset _j
   ;
   ;    {
   ;        for( j=0; j<4; ++j )
   ;
?live1@16: ; ECX = &i, EBX = &j
    xor       eax,eax
    mov       dword ptr [ebx],eax ;i = 0
    jmp       short @3
   ;
   ;        {
   ;            for( i=0; i<4; ++i )
   ;
```

```
@2:
    xor       edx,edx
    mov       dword ptr [ecx],edx ; j = 0

; Compute the index to the start of the
; current column of the array as
; base( TwoByTwo ) + eax*4. Leave this
; "column base address" in EDX:

    mov       eax,dword ptr [ebx]
    lea       edx,dword ptr [_TwoByTwo+4*eax]
    jmp       short @5
   ;
   ;            {
   ;                TwoByTwo[i][j] = i+j;
   ;
?live1@48: ; EAX = @temp0, EDX = @temp1, ECX = &i, EBX = &j
@4:

;
    mov       esi,eax                  ; Compute i+j
    add       esi,dword ptr [ebx]      ; EBX points at j's value

    shl       eax,4                    ; Multiply row index by 16

; Store the sum (held in ESI) into the specified array element.
; Note that EDX contains the base address plus the column
; offset into the array. EAX contains the row offset into the
; array. Their sum produces the address of the desired array
; element.

    mov       dword ptr [edx+eax],esi  ; Store sum into element

    inc       dword ptr [ecx]          ; increment i by 1
@5:
    mov       eax,dword ptr [ecx]      ; Fetch i's value
    cmp       eax,4                    ; Is i less than 4?
    jl        short @4                 ; If so, repeat inner loop
    inc       dword ptr [ebx]          ; Increment j by 1
@3:
    cmp       dword ptr [ebx],4        ; Is j less than 4?
    jl        short @2                 ; If so, repeat outer loop.
   ;


        .
        .
        .
; Storage for the 4x4 (x4 bytes) two-dimensional array:
; Total = 4*4*4 = 64 bytes:

    align   4
_TwoByTwo   label   dword
    db  64  dup(?)
```

In this example, the computation `rowIndex * 4 + columnIndex` is handled by the following four instructions (which also store away the array element):

```
; EDX = base address + columnIndex * 4

    mov       eax,dword ptr [ebx]
    lea       edx,dword ptr [_TwoByTwo+4*eax]
      .
      .
      .
; EAX = rowIndex, ESI = i+j

    shl       eax,4                    ; Multiply row index by 16
    mov       dword ptr [edx+eax],esi  ; Store sum into element
```

Note that this code sequence used the scaled-index addressing mode (along with the `lea` instruction) and the `shl` instruction to do the necessary multiplications. Because multiplication tends to be an expensive operation, most compilers avoid using it when calculating indices into multidimensional arrays. Nevertheless, by comparing this code against the examples given for one-dimensional array access, you can see that two-dimensional array access is a bit more expensive in terms of the number of machine instructions you must use to compute the index into the array.

Three-dimensional array access is even costlier than two-dimensional array access. Here is a C/C++ program that initializes the elements of a three-dimensional array:

```
#include <stdlib.h>
int i, j, k;
int ThreeByThree[3][3][3];

int main( int argc, char **argv )
{
    for( j=0; j<3; ++j )
    {
        for( i=0; i<3; ++i )
        {
            for( k=0; k<3; ++k )
            {
                // Initialize the 27 array elements
                // with a set of random values:

                ThreeByThree[i][j][k] = rand();
            }
        }
    }
    return 0;
}
```

And here's the 32-bit 80x86 assembly language output that the Microsoft Visual C++ compiler produces:

```
; Line 9
        mov     DWORD PTR j, 0     // for( j = 0;...;... )
        jmp     SHORT $LN4@main

$LN2@main:
        mov     eax, DWORD PTR j   // for( ...;...;++j )
        inc     eax
        mov     DWORD PTR j, eax

$LN4@main:
        cmp     DWORD PTR j, 4     // for( ...;j<4;... )
        jge     $LN3@main

; Line 11
        mov     DWORD PTR i, 0     // for( i=0;...;... )
        jmp     SHORT $LN7@main

$LN5@main:
        mov     eax, DWORD PTR i   // for( ...;...;++i )
        inc     eax
        mov     DWORD PTR i, eax

$LN7@main:
        cmp     DWORD PTR i, 4     // for( ...;i<4;... )
        jge     SHORT $LN6@main

; Line 13
        mov     DWORD PTR k, 0     // for( k=0;...;... )
        jmp     SHORT $LN10@main

$LN8@main:
        mov     eax, DWORD PTR k   // for( ...;...;++k )
        inc     eax
        mov     DWORD PTR k, eax

$LN10@main:
        cmp     DWORD PTR k, 3     // for( ...; k<3;... )
        jge     SHORT $LN9@main

; Line 18
        call    rand
        movsxd  rcx, DWORD PTR i   // Index =( (( ( i*3 + j ) * 3 + k ) * 4 )
        imul    rcx, rcx, 36       // 00000024H
        lea     rdx, OFFSET FLAT:ThreeByThree
        add     rdx, rcx
        mov     rcx, rdx
        movsxd  rdx, DWORD PTR j
        imul    rdx, rdx, 12
        add     rcx, rdx
        movsxd  rdx, DWORD PTR k
```

```
//  ThreeByThree[i][j][k] = rand();

        mov     DWORD PTR [rcx+rdx*4], eax

; Line 19
        jmp     SHORT $LN8@main // End of for( k = 0; k<3; ++k )
$LN9@main:
; Line 20
        jmp     SHORT $LN5@main // End of for( i = 0; i<4; ++i )
$LN6@main:
; Line 21
        jmp     $LN2@main       // End of for( j = 0; j<4; ++j )
$LN3@main:
```

If you're interested, you can write your own short HLL programs and analyze the assembly code emitted for *n*-dimensional arrays (*n* being greater than or equal to 4).

The choice of column-major or row-major array ordering is generally dictated by your compiler, if not by the programming language definition. No compiler I'm aware of will let you choose which array ordering you prefer on an array-by-array basis (or even across a whole program, for that matter). However, there's really no need to do this, as you can easily simulate either storage mechanism by simply changing the definitions of "rows" and "columns" in your programs.

Consider the following C/C++ array declaration:

```
int array[ NumRows ][ NumCols ];
```

Normally, you'd access an element of this array using a reference like this:

```
element = array[ rowIndex ][ colIndex ]
```

If you increment through all the column index values for each row index value (which you also increment), you'll access sequential memory locations when accessing elements of this array. That is, the following C `for` loop initializes sequential locations in memory with `0`:

```
for( row=0; row<NumRows; ++row )
{
    for( col=0; col<NumCols; ++col )
    {
        array[ row ][ col ] = 0;
    }
}
```

If `NumRow` and `NumCols` are the same value, then accessing the array elements in column-major rather than row-major order is trivial. Simply swap the indices in the previous code fragment to obtain:

```
for( row=0; row<NumRows; ++row )
{
    for( col=0; col<NumCols; ++col )
    {
        array[ col ][ row ] = 0;
    }
}
```

If *NumCols* and *NumRows* are not the same value, you'll have to manually compute the index into the column-major array and allocate the storage in a one-dimensional array, as follows:

```
int columnMajor[ NumCols * NumRows ]; // Allocate storage
    .
    .
    .
for( row=0; row<NumRows; ++row)
{
    for( col=0; col<NumCols; ++col )
    {
        columnMajor[ col*NumRows + row ] = 0;
    }
}
```

Swift users who want a true multidimensional array implementation (not an array-of-arrays implementation) will need to allocate storage for the whole array as a single `ContiguousArray` type and then compute the indices into the array manually:

```
import Foundation

// Create a 3-D array[4][4][4]:

var a1 = ContiguousArray<Int>( repeating:0, count:4*4*4 )

for var i in 0...3
{
    for var j in 0...3
    {
        for var k in 0...3
        {
            a1[ (i*4+j)*4 + k ] = (i*4+j)*4 + k
        }
    }
}
print( a1 )
```

Here's the output from this program:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63]
```

Although it's possible to access arrays using a column-major organization if your application requires it, you should exercise extreme caution when accessing arrays in a manner other than the language's default scheme. Many optimizing compilers are smart enough to recognize when you're accessing arrays in the default manner, and they generate far better code in those circumstances. Indeed, the examples presented so far have explicitly accessed arrays in uncommon ways in order to thwart the compilers' optimizers. Consider the following C code and the Visual C++ output (with optimization enabled):

```
#include <stdlib.h>
int i, j, k;
int ThreeByThreeByThree[3][3][3];

int main( int argc, char **argv )
{
    // The important difference to note here is how
    // the loops are arranged with the indices i, j, and k
    // used so that i changes the slowest and k changes
    // most rapidly (corresponding to row-major ordering).

    for( i=0; i<3; ++i )
    {
        for( j=0; j<3; ++j )
        {
            for( k=0; k<3; ++k )
            {
                ThreeByThreeByThree[i][j][k] = 0;
            }
        }
    }
    return 0;
}
```

Here's the Visual C++ assembly language output for the `for` loops in the previous code. In particular, note how the compiler substituted an 80x86 `stosd` instruction in place of the three loops:

```
    push    edi
;
; The following code zeros out the 27 (3*3*3) elements
; of the ThreeByThreeByThree array.

    mov ecx, 27                 ; 0000001bH
    xor eax, eax
    mov edi, OFFSET FLAT:_ThreeByThreeByThree
    rep stosd
```

If you rearrange your indices so that you're not storing zeros into consecutive memory locations, Visual C++ will not compile to the `stosd` instruction. Even if the end result is the zeroing of the entire array, the compiler believes that the semantics of `stosd` are different. (Imagine two threads in a program that are both reading and writing `ThreeByThreeByThree` array elements concurrently; the program's behavior could be different based on the order of the writes to the array.)

In addition to compiler semantics, there are also good hardware reasons not to change the default array ordering. Modern CPU performance is highly dependent on the effectiveness of the CPU's cache. Because cache performance depends on the temporal and spatial locality of the data in the cache, you must be careful not to access data in a way that disturbs locality. In particular, accessing array elements in a manner that is inconsistent with their storage order will dramatically impact spatial locality and in turn hurt performance. The moral of the story: adopt the compiler's array organization unless you really know what you're doing.

8.1.6.7 Improving Array Access Efficiency in Your Applications

Follow these rules when using arrays in your applications:

- Never use a multidimensional array when a one-dimensional array will work. This is not to suggest that you should simulate multidimensional arrays by manually computing a row-major (or column-major) index into a one-dimensional array, but if you can express an algorithm using a one-dimensional array rather than a multidimensional array, you should.
- When you must use multidimensional arrays in your application, try to use array bounds that are powers of 2 or, at least, multiples of 4. Compilers can compute indices into such arrays much more efficiently than arrays with arbitrary bound values.
- When accessing elements of a multidimensional array, try to do so in a manner than supports sequential memory access. For row-major-ordered arrays, this implies sequencing through the rightmost index the fastest and the leftmost index the slowest (and vice versa for column-major ordered arrays).
- If your language supports operations on entire rows or columns (or other large pieces of the array) with a single operation, use those facilities rather than accessing individual elements using nested loops. Often, the loop overhead, amortized over each array element you access, is greater than the cost of the index calculation and element access. This is particularly important when the array operation is the only thing happening in the loop(s).
- Always keep in mind the issues of spatial and temporal locality when accessing array elements. Accessing a large number of array elements in a random (or non–cache-friendly) fashion can cause thrashing in the cache and virtual memory subsystem.[3]

3. See *WGC1* for a discussion of thrashing.

The last point is particularly important. Consider the following HLA program:

```
program slow;
#include ( "stdlib.hhf" )
begin slow;

    // A dynamically allocated array accessed as follows:
    // array [12][1000][1000]

    malloc( 12_000_000 ); // Allocate 12,000,000 bytes
    mov( eax, esi );

    // Initialize each byte of the array to 0:

    for( mov( 0, ecx ); ecx < 1000; inc( ecx ) ) do

        for( mov( 0, edx ); edx < 1000; inc( edx ) ) do

            for( mov( 0, ebx ); ebx < 12; inc( ebx ) ) do

                // Compute the index into the array
                // as EBX*1_000_000 + EDX*1_000 + ECX

                intmul( 1_000_000, ebx, eax );
                intmul( 1_000, edx, edi );
                add( edi, eax );
                add( ecx, eax );
                mov( 0, (type byte [esi+eax]) );

            endfor;

        endfor;

    endfor;

end slow;
```

Simply swapping the loops around—so that the EBX loop is the outermost loop and the ECX loop is the innermost loop—can make this program run up to 10 times faster. The reason is that the program, as currently written, accesses an array stored in row-major order nonsequentially. Changing the rightmost index (ECX) most frequently and the leftmost index (EBX) least frequently, then, means this program will access memory sequentially. This allows the cache to work better, which dramatically improves program performance.

8.1.7 Dynamic vs. Static Arrays

Some languages allow you to declare arrays whose size isn't known until the program is running. Such arrays are quite useful, because many programs cannot predict how much space they will need for a data structure until they receive input from a user. For example, consider a program that reads a text file from disk, line by line, into an array of strings. Until the program actually reads the file and counts the number of lines, it doesn't know how many elements it will need for the array of strings. When writing the program, the programmer had no way of knowing how large the array would need to be.

Languages that provide support for such arrays generally call them *dynamic arrays*. This section explores the issues surrounding them and their counterpart, *static arrays*. A good place to start is with some definitions:

Static array (or pure static array)
An array whose size the program knows during compilation. This means the compiler/linker/operating system can allocate storage for the array before the program begins execution.

Pseudo-static array
An array whose size is known to the compiler, but for which the program doesn't actually allocate storage until runtime. Automatic variables (that is, nonstatic local variables in a function or procedure) are good examples of pseudo-static objects. The compiler knows their exact size while compiling the program, but the program doesn't actually allocate storage for them in memory until the function or procedure containing the declaration executes.

Pseudo-dynamic array
An array whose size the compiler cannot determine prior to program execution. Typically, the program determines the size of the array at runtime as a result of user input or as part of some other calculation. Once the program allocates storage for a pseudo-dynamic array, however, the size of the array remains fixed until the program either terminates or deallocates storage for that array. In particular, you cannot change the size of a pseudo-dynamic array to add or delete selected elements without deallocating the storage for the whole array.

Dynamic array (or pure dynamic array)
An array whose size the compiler cannot determine until the program runs and, in fact, cannot even be sure of once it creates the array. A program may change the size of a dynamic array at any time, adding or deleting elements, without affecting the values already present in the array (of course, if you delete some array elements, their values are lost).

NOTE *Static and pseudo-static arrays are examples of the static and automatic objects discussed previously in this book. See Chapter 7 for a review.*

8.1.7.1 One-Dimensional Pseudo-Dynamic Arrays

Most languages that claim support for dynamic arrays actually support pseudo-dynamic arrays. That is, you may specify the size of an array when you first create it, but once you've done so, you can't easily change the array's size without first deallocating the original storage for the array. Consider the following Visual Basic statement:

```
dim dynamicArray[ i * 2 ]
```

Assuming `i` is an integer variable that you've assigned some value prior to this statement's execution, upon encountering this statement Visual Basic will create an array with `i`×`2` elements. In languages that do support (pseudo-) dynamic arrays, array declarations are usually executable statements, whereas in languages that don't support dynamic arrays, such as C and Pascal, they are not. They are simply declarations that the compiler processes for bookkeeping reasons, but for which the compiler generates no machine code.

Although standard C/C++ does not support pseudo-dynamic arrays, the GNU C/C++ implementation does. Therefore, it's legal to write a function like the following in GNU C/C++:

```
void usesPDArray( int aSize )
{
    int array[ aSize ];
        .
        .
        .
} /* end of function usesPDArray */
```

Of course, if you use this feature in GCC, you'll only be able to compile your programs with GCC.[4] That's why you won't see many C/C++ programmers using this type of code in their programs.

If you're using a language like C/C++ that doesn't support pseudo-dynamic arrays, but does provide a generic memory allocation function, then you can easily create arrays that act just like one-dimensional pseudo-dynamic arrays. This is particularly easy in languages that don't check the range of array indices, like C/C++. Consider the following code:

```
void usesPDArray( int aSize )
{
    int *array;

    array = (int *) malloc( aSize * sizeof( int ) );
        .
        .
        .
    free( array );

} /* end of function usesPDArray */
```

One issue with using a memory allocation function like `malloc()` is that you must remember to explicitly free the storage prior to returning from the function (as the `free()` call does in this case). Some versions of the C

4. As it turns out, Clang on the Mac also supports this feature.

standard library include a `talloc()` function that allocates dynamic storage on the stack. Calls to `talloc()` are much faster than calls to `malloc()` and `free()`, and `talloc()` automatically frees up the storage when you return.

8.1.7.2 Multidimensional Pseudo-Dynamic Arrays

If you want to create multidimensional pseudo-dynamic arrays, that's another problem altogether. With a one-dimensional pseudo-dynamic array, the program really doesn't need to keep track of the array bounds for any reason but to verify that the array index is valid. For multidimensional arrays, however, the program must maintain additional information about the upper and lower bounds of each dimension of the array; the program needs that size information to compute the offset of an array element from a list of array indices, as you saw earlier in the chapter. So, in addition to maintaining a pointer containing the address of the base element of the array, programs using pseudo-dynamic arrays must also keep track of the array bounds.[5] This collection of information—the base address, number of dimensions, and bounds for each dimension—is known as a *dope vector*. In a language like HLA, C/C++, or Pascal, you'd typically create a `struct` or `record` to maintain the dope vector (see Chapter 11 for more information about `struct`s and records). Here's an example of a dope vector you might create for a two-dimensional integer array using HLA:

```
type
    dopeVector2D :
        record
            ptrToArray :pointer to int32;
            bounds :uns32[2];
        endrecord;
```

Here's the HLA code you would use to read the bounds of a two-dimensional array from the user and allocate storage for the pseudo-dynamic array using this dope vector:

```
var
    pdArray :dopeVector2D;
        .
        .
        .
stdout.put( "Enter array dimension #1:" );
stdin.get( pdArray.bounds[0] );
stdout.put( "Enter array dimension #2:" );
stdin.get( pdArray.bounds[4] );  //Remember, '4' is a
                                 // byte offset into bounds.
```

5. Technically, the code doesn't need to maintain the size of the last array dimension if the program doesn't check the validity of array indices applied to the array. In general, however, most languages that support pseudo-dynamic arrays maintain all the information.

```
// To allocate storage for the array, we must
// allocate bounds[0]*bounds[4]*4 bytes:

mov( pdArray.bounds[0], eax );

// bounds[0]*bounds[4] -> EAX

intmul( pdArray.bounds[4], eax );

// EAX := EAX * 4 (4=size of int32).

shl( 2, eax );

// Allocate the bytes for the array.

malloc( eax );

// Save away base address.

mov( eax, pdArray.ptrToArray );
```

This example emphasizes that the program must compute the size of the array as the product of the array dimensions and the element size. When processing static arrays, the compiler can compute this product during compilation. When working with dynamic arrays, however, the compiler must emit machine instructions to compute this product at runtime, which means your program will be slightly larger and slightly slower than if you had used a static array.

If a language doesn't directly support pseudo-dynamic arrays, you'll have to translate a list of indices into a single offset using the row-major function (or something comparable). This is true in HLLs as well as assembly language. Consider the following C++ example, which uses row-major ordering to access an element of a pseudo-dynamic array:

```
typedef struct
{
    int *ptrtoArray;
    int bounds[2];
} dopeVector2D;

dopeVector2D pdArray;
        .
        .
        .
    // Allocate storage for the pseudo-dynamic array:

    cout << "Enter array dimension #1:";
    cin >> pdArray.bounds[0];
    cout << "Enter array dimension #2:" ;
    cin >> pdArray.bounds[1];
    pdArray.ptrtoArray =
        new int[ pdArray.bounds[0] * pdArray.bounds[1] ];
        .
        .
        .
    // Set all the elements of this dynamic array to
    // successive integer values:

    k = 0;
    for( i=0; i < pdArray.bounds[0]; ++i )
    {
        for( j=0; j < pdArray.bounds[1]; ++j )
        {
            // Use row-major ordering to access
            // element [i][j]:

            *(pdArray.ptrtoArray + i*pdArray.bounds[1] + j) = k;
            ++k;
        }
    }
```

As for one-dimensional pseudo-dynamic arrays, memory allocation and deallocation can be more expensive than the actual array access—particularly if you allocate and deallocate many small arrays.

A big problem with multidimensional dynamic arrays is that the compiler doesn't know the array bounds at compile time, so it can't generate array access code that's as efficient as what's possible for pseudo-static and static arrays. As an example, consider the following C code:

```
#include <stdlib.h>

int main( int argc, char **argv )
{

    // Allocate storage for a 3x3x3 dynamic array:

    int *iptr = (int*) malloc( 3*3*3 *4 );
    int depthIndex;
    int rowIndex;
    int colIndex;

    // A pseudo-static 3x3x3 array for comparison:

    int ssArray[3][3][3];

    // The following nested for loops initialize all
    // the elements of the dynamic 3x3x3 array with
    // zeros:

    for( depthIndex=0; depthIndex<3; ++depthIndex )
    {
        for( rowIndex=0; rowIndex<3; ++rowIndex )
        {
            for( colIndex=0; colIndex<3; ++colIndex )
            {
                iptr
                [
                    // Row-major order computation:

                      ((depthIndex*3) + rowIndex)*3
                    + colIndex

                ] = 0;
            }
        }
    }

    // The following three nested loops are comparable
    // to the above, but they initialize the elements
    // of a pseudo-static array. Because the compiler
    // knows the array bounds at compile time, it can
    // generate better code for this sequence.

    for( depthIndex=0; depthIndex<3; ++depthIndex )
    {
        for( rowIndex=0; rowIndex<3; ++rowIndex )
        {
            for( colIndex=0; colIndex<3; ++colIndex )
            {
                ssArray[depthIndex][rowIndex][colIndex] = 0;
            }
        }
    }

    return 0;
}
```

Here's the pertinent portion of the PowerPC code that GCC emits for this C program (manually annotated). The important thing to notice here is that the dynamic array code is forced to use an expensive multiply instruction, whereas the pseudo-static array code doesn't need this instruction.

```
    .section __TEXT,__text,regular,pure_instructions

_main:

// Allocate storage for local variables
// (192 bytes, includes the ssArray,
// loop control variables, other stuff,
// and padding to 64 bytes):

    mflr r0
    stw r0,8(r1)
    stwu r1,-192(r1)

// Allocate 108 bytes of storage for
// the 3x3x3 array of 4-byte ints.
// This call to  malloc leaves the
// pointer to the array in R3.
```

```
    li r3,108
    bl L_malloc$stub

    li r8,0     // R8= depthIndex
    li r0,0

    // R10 counts off the number of
    // elements in rows we've processed:

    li r10,0

// Top of the outermost for loop

L16:
    // Compute the number of bytes
    // from the beginning of the
    // array to the start of the
    // row we are about to process.
    // Each row contains 12 bytes and
    // R10 contains the number of rows
    // processed so far. The product
    // of 12 by R10 gives us the number
    // of bytes to the start of the
    // current row. This value is put
    // into R9:

    mulli r9,r10,12

    li r11,0    // R11 = rowIndex

// Top of the middle for loop

L15:
    li r6,3     // R6/CTR = colIndex

    // R3 is the base address of the array.
    // R9 is the index to the start of the
    // current row, computed by the MULLI
    // instruction, above. R2 will now
    // contain the base address of the
    // current row in the array.

    add r2,r9,r3

    // CTR = 3

    mtctr r6

    // Repeat the following loop
    // once for each element in
    // the current row of the array:

L45:
    stw r0,0(r2)    // Zero out current element
    addi r2,r2,4    // Move on to next element
    bdnz L45        // Repeat loop CTR times

    addi r11,r11,1  // Bump up RowIndex by 1
    addi r9,r9,12   // Index of next row in array
    cmpwi cr7,r11,2 // Repeat for RowIndex=0..2
    ble+ cr7,L15

    addi r8,r8,1    // Bump up depthIndex by 1
    addi r10,r10,3  // Bump up element cnt by 3
    cmpwi cr7,r8,2  // Repeat for depthIndex=0..2
    ble+ cr7,L16

////////////////////////////////////////////////////
//
// Here's the code that initializes the pseudo-static
// array:

    li r8,0         // DepthIndex = 0
    addi r10,r1,64  // Compute base address of ssArray
    li r0,0
    li r7,0         // R7 is index to current row
L31:
    li r11,0        // RowIndex = 0
    slwi r9,r7,2    // Convert row/int index to
                    // row/byte index (int_index*4)
L30:
    li r6,3         // # iterations for colIndex
    add r2,r9,r10   // Base+row_index = row address
    mtctr r6        // CTR = 3

// Repeat innermost loop three times:

L44:
    stw r0,0(r2)    // Zero out current element
    addi r2,r2,4    // Bump up to next element
    bdnz L44        // Repeat CTR times

    addi r11,r11,1  // Bump up RowIndex by 1
    addi r9,r9,12   // R9=Adrs of start of next row
    cmpwi cr7,r11,2 // Repeat until RowIndex >=3
    ble+ cr7,L30

    addi r8,r8,1    // Bump up depthIndex by 1
    addi r7,r7,9    // Index of next depth in array
    cmpwi cr7,r8,2
    ble+ cr7,L31

    lwz r0,200(r1)
    li r3,0
    addi r1,r1,192
    mtlr r0
    blr
```

Different compilers and different optimization levels handle dynamic and pseudo-static array access in different ways. Some compilers generate the same code for both sequences, but many do not. The bottom line is that multidimensional dynamic array access is never faster than pseudo-static multidimensional array access, and it is sometimes slower.

8.1.7.3 Pure Dynamic Arrays

Pure dynamic arrays are even more difficult to manage. You'll rarely find them outside of very high-level languages like APL, SNOBOL4, LISP, and Prolog. The one notable exception is Swift, whose arrays are pure dynamic arrays. Most languages that support pure dynamic arrays don't force you to explicitly declare or allocate storage for an array. Instead, you just use elements of an array, and if an element isn't currently present in the array, the language automatically creates it for you. So, what happens if you currently have an array with elements `0` through `9` and you decide to use element `100`? Well, the result is language dependent. Some languages that support pure dynamic arrays will automatically create array elements `10..100` and initialize elements `10..99` with `0` (or some other default value). Other languages may allocate only element `100` and keep track of the fact that the other elements are not yet present in the array. Regardless, the extra bookkeeping necessary for each access to the array can be quite expensive. That's why languages that support pure dynamic arrays aren't more popular—they tend to execute programs slowly.

If you're using a language that supports dynamic arrays, keep in mind the costs associated with array access in that language. If you're using a language that doesn't support dynamic arrays, but does support memory allocation/deallocation (for example, C/C++, Java, or assembly), you can implement dynamic arrays yourself. You'll be painfully aware of the costs of using such an array, because you'll probably have to write all the code that manipulates its elements, although that's not an altogether bad thing. If you're using C++, you can even overload the array index operator (`[ ]`) to hide the complexity of dynamic array element access. Generally, though, programmers who need the true semantics of dynamic arrays will usually choose a language that directly supports them. Again, if you choose to go this route, just be mindful of the costs.

8.2 For More Information

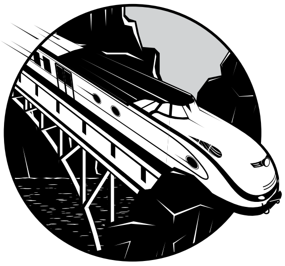
Duntemann, Jeff. *Assembly Language Step-by-Step.* 3rd ed. Indianapolis: Wiley, 2009.

Hyde, Randall. *The Art of Assembly Language.* 2nd ed. San Francisco: No Starch Press, 2010.

Knuth, Donald. *The Art of Computer Programming, Volume I: Fundamental Algorithms. 3rd ed.* Boston: Addison-Wesley Professional, 1997.

9

POINTER DATA TYPES



Pointers are the data type equivalent of a `goto` statement. Used carelessly, they can turn a robust and efficient program into a buggy and inefficient junk pile. Unlike `goto` statements, however, pointers can be difficult to avoid in many common programming languages.

There are no "pointers considered harmful" papers in academic journals like Dijkstra's "Go To Statement Considered Harmful" letter.[1] Many languages, like Java and Swift, attempt to restrict pointers, but several popular languages still use them, so great programmers need to be able to deal with them. To that end, this chapter will discuss:

- The memory representation of pointers
- How high-level languages implement pointers

1. Edgar Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM* 11, no. 3 (1968).

- Dynamic memory allocation and its relationship to pointers
- Pointer arithmetic
- How memory allocators work
- Garbage collection
- Common pointer problems

By understanding the low-level implementation and use of pointers, you'll be able to write high-level code that is more efficient, safer, and more readable. This chapter will provide the information you need to use pointers appropriately and avoid the problems normally associated with them.

9.1 The Definition of a Pointer

A pointer is simply a variable whose value refers to some other object. High-level languages like Pascal and C/C++ hide the simplicity of pointers behind a wall of abstraction. HLL programmers generally rely on the high degree of abstraction provided by the language because they don't want to know what's going on behind the scenes. They just want a "black box" that produces predictable results. In the case of pointers, though, the abstraction may be *too* effective; pointers seem intimidating and opaque to many programmers. Well, fear not! Pointers are actually easy to deal with.

To understand how pointers work, I'll use the array data type as an example. Consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

Even if you don't know Pascal, the concept here is easy to understand. `M` is an array of 1,024 integers, indexed from `M[0]` to `M[1023]`. Each array element can hold an independent integer value. In other words, this array gives you 1,024 different integer variables, each of which you access via an array index (the variable's sequential position within the array) rather than by name.

The statement `M[0] := 100;` stores the value `100` into the first element of the array `M`. Now consider the following two statements:

```
i := 0; (* assume "i" is an integer variable *)
M [i] := 100;
```

These two statements do the same thing as `M[0] := 100;`. In fact, you can use any integer expression producing a value in the range `0..1023` as an index into this array. The following statements still perform the same operation as our earlier statements:

```
i := 5;         (* assume all variables are integers*)
j := 10;
k := 50;
m [i*j-k] := 100;
```

But now look at the following statements:

```
M [1] := 0;
M [ M [1] ] := 100;
```

At first glance, these statements might seem confusing; however, they perform the same operation as in the previous examples. The first statement stores `0` into array element `M[1]`. The second statement fetches the value of `M[1]`, which is `0`, and uses that value to determine where to store the value `100`.

If you think this example is reasonable—perhaps bizarre, but usable nonetheless—then you'll have no problems with pointers, because `M[1]` is a pointer! Well, not really, but if you were to change `M` to "memory" and treat each element of this array as a separate memory location, then it would meet the definition of a pointer—that is, a memory variable whose value is the address of some other memory object.

9.2 Pointer Implementation in High-Level Languages

Although most languages implement pointers using memory addresses, a pointer is actually an abstraction of a memory address. Therefore, a language could define a pointer using any mechanism that maps the value of the pointer to the address of some object in memory. Some implementations of Pascal, for example, use offsets from a fixed memory address as pointer values. Some languages (including dynamic languages like Lisp) might actually implement pointers by using *double indirection*; that is, the pointer object contains the address of some memory variable whose value is the address of the object to access. This approach may seem somewhat convoluted, but it offers certain advantages in a complex memory management system, making it easier and more efficient to reuse blocks of memory. However, for simplicity's sake we'll assume that, as defined earlier, a pointer is a variable whose value is the address of some other object in memory. This is a safe assumption for many of the high-performance HLLs you're likely to encounter, such as C, C++, and Delphi.

You can indirectly access an object using a pointer with two 80x86 machine instructions, as follows:

```
mov( PointerVariable, ebx ); // Load pointer variable into a register.
mov( [ebx], eax );           // Use register-indirect mode to access data.
```

Now consider the double-indirect pointer implementation described earlier. Access to data via double indirection is less efficient than the straight pointer implementation because it takes an extra machine instruction to fetch the data from memory. This isn't obvious even in an HLL like C/C++ or Pascal, where using double indirection is explicit:

```
i = **cDblPtr;
i := pDblPtr^^;
```

This is syntactically similar to single indirection. In assembly language, however, you'll see the extra work involved:

```
mov( hDblPtr, ebx );  // Get the pointer to a pointer
mov( [ebx], ebx );    // Get the pointer to the value
mov( [ebx], eax );    // Get the value
```

Contrast this with the two earlier assembly instructions to access an object using single indirection. Because double indirection requires 50 percent more code (and twice as many slow memory accesses) than single indirection, you can see why many languages implement pointers using single indirection. To verify this, consider the machine code produced by a couple of different compilers when processing the following C code:

```
static int i;
static int j;
static int *cSnglPtr;
static int **cDblPtr;

int main( void )
{
        .
        .
        .
    j = *cSnglPtr;
    i = **cDblPtr;
        .
        .
        .
```

Here's the GCC output for the PowerPC processor:

```
; j = *cSnglPtr;

        addis r11,r31,ha16(_j-L1$pb)
        la r11,lo16(_j-L1$pb)(r11)
        addis r9,r31,ha16(_cSnglPtr-L1$pb)
        la r9,lo16(_cSnglPtr-L1$pb)(r9)
        lwz r9,0(r9)  // Get the ptr into register R9
        lwz r0,0(r9)  // Get the data at the pointer
        stw r0,0(r11) // Store into j

; i = **cDblPtr;
;
; Begin by getting the address of cDblPtr into R9:

        addis r11,r31,ha16(_i-L1$pb)
        la r11,lo16(_i-L1$pb)(r11)
        addis r9,r31,ha16(_cDblPtr-L1$pb)
        la r9,lo16(_cDblPtr-L1$pb)(r9)

        lwz r9,0(r9)  // Get the dbl ptr into R9
        lwz r9,0(r9)  // Get the ptr into R9
        lwz r0,0(r9)  // Get the value into R9
        stw r0,0(r11) // Store value into i
```

As you can see in this PowerPC example, fetching the value using double indirection takes one more instruction than it does using single indirection. Of course, the total number of instructions is rather large here, so this extra instruction doesn't contribute as much to the execution time as it does on the 80x86 where fewer instructions are involved. Consider the following GCC code output for the 32-bit 80x86:

```
; j = *cSnglPtr;

        movl    cSnglPtr, %eax
        movl    (%eax), %eax
        movl    %eax, j

; i = **cDblPtr;

        movl    cDblPtr, %eax
        movl    (%eax), %eax
        movl    (%eax), %eax
        movl    %eax, i
```

As we saw with the PowerPC code, double indirection requires extra machine instructions, so programs using double indirection will be larger and slower.

Notice that the PowerPC instruction sequences are twice as long as the 80x86 instruction sequences.[2] One positive way of viewing this is that double indirection has less of an impact on the execution time of the PowerPC code than it does on the 80x86 code. That is, the extra instruction represents only 13 percent of the total in the PowerPC code, versus 25 percent of the total in the 80x86 code.[3] This brief example should demonstrate that execution time and code space are not processor independent. Bad coding practices (such as using double indirection when it's not required) can have more impact on some processors than others.

9.3 Pointers and Dynamic Memory Allocation

Pointers typically reference anonymous variables that you allocate on the heap using memory allocation/deallocation functions like `malloc()`/`free()`, `new()`/`dispose()`, and `new()`/`delete()` (`std::make_unique` in C++17). Objects that you allocate on the heap are known as *anonymous variables* because you refer

2. This, by the way, is not a general rule concerning PowerPC versus 80x86 code. Memory references on the PowerPC are very costly; that's why the PowerPC code here is so long. However, the PowerPC has four times as many registers, so in real applications the code isn't always larger.

3. Keep in mind, however, that memory accesses are very slow if the data is not sitting in the cache. In that case, the majority of the time spent in this code will be waiting for memory, not executing instructions, so—all other things being equal—the two code sequences will have more comparable execution times.

to them by their address rather than by name. While the pointer variable may have a name, that name applies to the pointer's data (an address), not the object referenced by this address.

NOTE *The heap, as Chapter 7 explained, is a region in memory reserved for dynamic storage allocation.*

Dynamic languages handle memory allocation and deallocation operations in a transparent, automatic fashion. The application simply uses the dynamic data and leaves it up to the runtime system to allocate memory as needed and reuse storage for a different purpose when it is no longer needed. Without the need to explicitly allocate and deallocate memory for pointer variables, applications written in dynamic languages (such as AWK or Perl) are usually much easier to program and often contain far fewer errors. But this comes at the cost of efficiency, as they often run much slower than programs written in other languages. Conversely, traditional languages (such as C/C++) that require programmers to explicitly manage memory often produce more efficient applications, although the memory management code is prone to a higher percentage of defects due to its additional complexity.

9.4 Pointer Operations and Pointer Arithmetic

Most HLLs that provide a pointer data type let you assign addresses to pointer variables, compare pointer values for equality or inequality, and indirectly reference an object via a pointer. Some languages also allow additional operations, as you'll see in this section.

Many programming languages enable you to do limited arithmetic with pointers. At the very least, these languages allow you to add an integer constant to, or subtract one from, a pointer. To understand the purpose of these two arithmetic operations, recall the syntax of the `malloc()` function in the C standard library:

```
ptrVar = malloc( bytes_to_allocate );
```

The parameter you pass `malloc()` specifies the number of bytes of storage to allocate. A good C programmer generally supplies an expression like `sizeof(int)` as this parameter. The `sizeof()` function returns the number of bytes needed by its single parameter. Therefore, `sizeof(int)` tells `malloc()` to allocate at least enough storage for an `int` variable. Now consider the following call to `malloc()`:

```
ptrVar = malloc( sizeof( int ) * 8 ); // An array of 8 integers
```

If the size of an integer is 4 bytes, this call to `malloc()` will allocate storage for 32 bytes, at consecutive addresses in memory (see Figure 9-1).

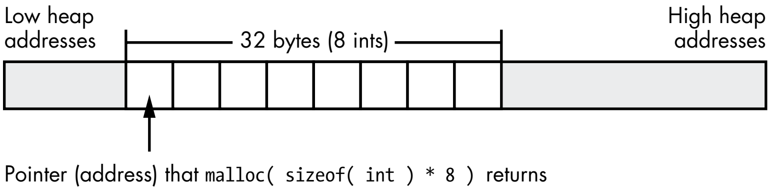


Figure 9-1: Memory allocation via `malloc(sizeof(int) * 8 )`

The pointer that `malloc()` returns contains the address of the first integer in this set, so the C program can directly access only the very first of these eight integers. To access the individual addresses of the other seven integers, you need to add an integer offset to that *base* address. On machines that support byte-addressable memory (such as the 80x86), the address of each successive integer in memory is the address of the previous integer plus the integer size. For example, if a call to the C standard library `malloc()` routine returns the memory address `$0300_1000`, then the eight integers that `malloc()` allocates will reside at the memory addresses shown in Table 9-1.

Table 9-1: Integer Addresses Allocated for Base Address `$0300_1000`

Integer	Memory address
First	`$0300_1000..$0300_1003`
Second	`$0300_1004..$0300..1007`
Third	`$0300_1008..$0300_100b`
Fourth	`$0300_100c..$0300_100f`
Fifth	`$0300_1010..$0300_1013`
Sixth	`$0300_1014..$0300..1017`
Seventh	`$0300_1018..$0300_101b`
Eighth	`$0300_101c..$0300_101f`

9.4.1 Adding an Integer to a Pointer

Because the eight integers in the previous section are exactly 4 bytes apart, you add 4 to the address of the first integer to obtain the address of the second integer. Likewise, the address of the third integer is the address of the second integer plus 4 bytes, and so on. In assembly language, you could access these eight integers using code like the following:

```
// malloc returns storage for eight
//  int32 objects in EAX.

malloc( @size( int32 ) * 8 );

mov( 0, ecx );
mov( ecx, [eax] );     // Zero out the 32 bytes
mov( ecx, [eax+4] );   // (4 bytes at a time).
mov( ecx, [eax+8] );
mov( ecx, [eax+12] );
mov( ecx, [eax+16] );
mov( ecx, [eax+20] );
mov( ecx, [eax+24] );
mov( ecx, [eax+28] );
```

Notice the use of the 80x86 indexed addressing mode to access the eight integers that `malloc()` allocates. The EAX register maintains the base (first) address of the eight integers that this code allocates, and the constant in the addressing mode of the `mov()` instruction indicates the offset of the specific integer from this base address.

Most CPUs use byte addresses for memory objects. Therefore, when a program allocates multiple copies of some *n*-byte object in memory, the objects won't begin at consecutive memory addresses; instead, they'll appear in memory at addresses that are *n* bytes apart. Some machines, however, don't allow a program to access memory at any arbitrary address; they require it to access data on address boundaries that are a multiple of a word, a double word, or even a quad word. Any attempt to access memory on some other boundary will raise an exception and potentially halt the application. If an HLL supports pointer arithmetic, it must take this fact into consideration and provide a generic pointer arithmetic scheme that's portable across different CPU architectures. The most common solution that HLLs use when adding an integer offset to a pointer is to multiply that offset by the size of the object that the pointer references. That is, if you have a pointer `p` to a 16-byte object in memory, then `p + 1` points 16 bytes beyond where `p` points. Likewise, `p + 2` points 32 bytes beyond the address contained in `p`. As long as the size of the data object is a multiple of the required alignment size (which the compiler can enforce by adding padding bytes, if necessary), this scheme avoids problems on architectures that require aligned data access. Consider, for example, the following C/C++ code:

```
int *intPtr;
        .
        .
        .
    // Allocate storage for eight integers:

    intPtr = malloc( sizeof( int ) * 8 );

    // Initialize each of these integer values:

    *(intPtr+0) = 0;
    *(intPtr+1) = 1;
    *(intPtr+2) = 2;
    *(intPtr+3) = 3;
    *(intPtr+4) = 4;
    *(intPtr+5) = 5;
    *(intPtr+6) = 6;
    *(intPtr+7) = 7;
```

This example demonstrates how C/C++ uses pointer arithmetic to specify an integer-sized offset from the base pointer address.

It's important to note that the addition operator only makes sense between a pointer and an integer value. For example, in C/C++ you can indirectly access objects in memory using an expression like `*(p + i)` (where `p` is a pointer to an object and `i` is an integer value). It doesn't make sense to add two pointers together. Similarly, it isn't logical to add other data types with a pointer—for example, adding a floating-point value to a pointer. (What does it mean to reference the data at some base address plus 1.5612?) Operations on pointers involving strings, characters, and other data types don't make much sense, either. Integers (signed and unsigned) are the only reasonable values to add to a pointer.

On the other hand, not only can you add an integer to a pointer, but you can also add a pointer to an integer and the result is still a pointer (both `p + i` and `i + p` are legal). This is because addition is *commutative*—the order of the operands does not affect the result.

9.4.2 Subtracting an Integer from a Pointer

Subtracting an integer from a pointer references a memory location immediately before the base address held in the pointer. However, subtraction is not commutative, and subtracting a pointer from an integer is not a legal operation (`p - i` is legal, but `i - p` is not).

In C/C++, `*(p - i)` accesses the `i`th object immediately before the object at which `p` points. In 80x86 assembly language, as in assembly on many processors, you can also specify a negative constant offset when using an indexed addressing mode. For example:

```
mov( [ebx-4], eax );
```

Keep in mind, 80x86 assembly language uses byte offsets, not object offsets (as C/C++ does). Therefore, this statement loads into EAX the double word in memory immediately preceding the memory address in EBX.

9.4.3 Subtracting a Pointer from a Pointer

In contrast to addition, it makes sense to subtract the value of one pointer variable from another. Consider the following C/C++ code, which proceeds through a string of characters looking for the first `e` character that follows the first `a` that it finds (you could use the result of such a calculation, for example, to extract a substring):

```
int distance;
char *aPtr;
char *ePtr;
    .
    .
    .
aPtr = someString;  // Get ptr to start of string in aPtr.
```

```
// While we're not at the end of the string
// and the current char isn't 'a':

while( *aPtr != '\0' && *aPtr != 'a' )
{
    // Move on to the next character pointed at by aPtr.

    aPtr = aPtr + 1;
}

// while we're not at the end of the string
// and the current character isn't 'e'
//
// Start at the 'a' char (or end of string if no 'a').

ePtr = aPtr;
while( *ePtr != '\0' && *ePtr != 'e' )
{
    // Move on to the next character pointed at by aPtr.
    ePtr = ePtr + 1;
}

// Now compute the number of characters between
// the 'a' and the 'e' (counting the 'a' but not
// counting the 'e'):

distance = (ePtr - aPtr);
```

Subtracting one pointer from the other produces the number of data objects that exist between them (in this case, `ePtr` and `aPtr` point at characters, so this subtraction produces the number of characters, or bytes if 1-byte characters, between the two pointers).

The subtraction of two pointer values makes sense only if they both reference the same data structure (for example, an array, string, or record) in memory. Although assembly language will allow you to subtract two pointers that point at completely different objects in memory, their difference will probably have very little meaning.

For pointer subtraction in C/C++, the base types of the two pointers must be identical (that is, the two pointers must contain the address of two objects whose types are identical). This restriction exists because pointer subtraction in C/C++ produces the number of objects, not the number of bytes, between the two pointers. Computing the number of objects between a byte in memory and a double word in memory wouldn't make any sense. The result would be neither a byte count nor a double-word count.

The subtraction of two pointers can return a negative number if the left pointer operand is at a lower memory address than the right pointer operand. Depending on your language and its implementation, you might need to take the absolute value of the result if you're interested only in the distance between the two pointers and you don't care which pointer contains the greater address.

9.4.4 Comparing Pointers

Comparisons are another set of operations that make sense for pointers. Almost every language that supports pointers allows you to compare two pointers to see whether or not they are equal. A pointer comparison tells you whether the pointers reference the same object in memory. Some languages (such as assembly and C/C++) also let you compare two pointers to see if one pointer is less than or greater than the other. Like subtracting two pointers, comparing two pointers makes sense only if they have the same base type and point into the same data structure. If one pointer is less than another, this tells you that the pointer references an object within the data structure that appears before the object whose address the second pointer contains. The converse is true for the greater-than comparison. This short example in C demonstrates pointer comparison:

```
#include <stdio.h>

int iArray[256];
int *ltPtr;
int *gtPtr;



int main( int argc, char **argv )
{
    int lt;
    int gt;

    // Put the address of the "argc" element
    // of iArray into ltPtr. This is done
    // so that the optimizer doesn't completely
    // eliminate the following code (as would
    // happen if we just specified a constant
    // index):

    ltPtr = &iArray[argc];

    // Put the address of the eighth array
    // element into gtPtr.

    gtPtr = &iArray[7];

    // Assuming you don't type seven or more
    // command-line parameters when running
    // this program, the following two
    // assignments should set lt and gt to 1.

    lt = ltPtr < gtPtr;
    gt = gtPtr > ltPtr;
    printf( "lt:%d, gt:%d\n", lt, gt );
    return 0;
}
```

At the (x86-64) machine language level, addresses are simply 64-bit quantities, so the machine code can compare these pointers as though they're 64-bit integer values. Here's the x86-64 assembly code that Visual C++ emits for this example:

```
;
; Grab ARGC (passed to the program in rcx), use
; it as an index into iArray (4 bytes per element,
; hence the "*4" in the scaled-index addressing mode),
; compute the address of this array element (using the
; LEA -- load effective address -- instruction), and
; store the resulting address into ltPtr:
; Line 24
        movsxd  rax, ecx ; rax=rcx
; Line 37
        xor     edx, edx ;edx = 0
        mov     r8d, edx ;Initialize boolean result w/false
        lea     rcx, OFFSET FLAT:iArray ;rcx = base address of iArray
        lea     rcx, QWORD PTR [rcx+rax*4] ;rcx = &iArray[argc]

        lea     rax, OFFSET FLAT:iArray+28 ;rax=&iArray[7] (7*4 = 28)
        mov     QWORD PTR ltPtr, rcx ;ltPtr = &iArray[argc]
        cmp     rax, rcx ;carry flag = !(ltPtr < gtPtr)
        mov     QWORD PTR gtPtr, rax ;gtPtr = &iArray[7]
        seta    r8b ;r8b = ltPtr < gtPtr (which is !gtPtr > ltPtr)
        cmp     rcx, rax ;Carry flag = !(gtPtr > ltPtr)
; Line 38
        lea     rcx, OFFSET FLAT:??_C@_0O@KJKFINNE@lt?3?$CFd?0?5gt?3?$CFd?6?$AA@
        setb    dl ;dl = !(ltPtr < gtPtr ) (which is !(gtPtr > ltPtr)
        call    printf
;
```

Other than the trickery behind computing `true` (`1`) or `false` (`0`) after comparing the two addresses, this code is a very straightforward compilation to machine code.

9.4.5 Using Logical AND/OR Operations with Pointers

On byte-addressable machines, it makes sense to logically AND an address with a bit string value, because masking off the low-order (LO) bits in an address is an easy way to align it on a boundary that is a power of 2. For example, if the 32-bit 80x86 EBX register contains an arbitrary address, the following assembly language statement rounds the pointer in EBX down to an address that is a multiple of 4 bytes:

```
and( $FFFF_FFFC, ebx );
```

This operation is very useful when you want to ensure that memory is accessed on a nice memory boundary. For example, suppose you have a memory allocation function that can return a pointer to a block of memory that begins at an arbitrary byte boundary. To ensure that the data structure the pointer points to begins on a double word (dword) boundary, you can use assembly code like the following:

```
// # of bytes to allocate

mov( nBytes, eax );

// Provide a "cushion" for rounding.

add( 3, eax );

// Allocate the memory (returns pointer in EAX).

malloc( eax );

// Round up to the next higher dword, if not dword-aligned.

add( 3, eax );

// Make the address a multiple of 4.

and( $ffff_fffc, eax );
```

This code allocates an extra 3 bytes when calling `malloc()` so that it can add 0, 1, 2, or 3 to the address that `malloc()` returns in order to align the object on a `dword` address. On return from `malloc()`, the code adds 3 to the address and, if it wasn't already a multiple of 4, the address will cross the next `dword` boundary. Using the AND instruction reduces the address back to the previous `dword` boundary (either the next `dword` boundary, or the original address if it was already `dword`-aligned).

9.4.6 Using Other Operations with Pointers

Beyond addition, subtraction, comparison, and possibly AND or OR operations, very few arithmetic operations make sense with pointer operands. What does it mean to multiply a pointer by some integer value (or another pointer)? What does division of pointers mean? What do you get when you shift a pointer to the left by one bit position? You could make up some sort of definition for these operations, but considering the original arithmetic definitions, these operations just aren't reasonable for pointers.

Several languages (including C/C++ and Pascal) restrict other pointer operations. There are several good reasons for limiting what a programmer can do with a pointer, such as:

- Code involving pointers is notoriously difficult to optimize. By limiting the number of pointer operations, the compiler can make assumptions about the code that it could not otherwise. This allows the compiler (in theory) to produce better machine code.
- Code containing pointer manipulations is more likely to be defective. Limiting programmers' options in this area helps prevent pointer abuse, and leads to more robust code.

NOTE *The section "Common Pointer Problems" on page 286 describes the most serious of these errors and ways to avoid them in your code.*

- Some pointer operations—particularly certain arithmetic operations—are not portable across CPU architectures. For example, on some segmented architectures (such as the original 16-bit 80x86), subtracting the values of two pointers may not produce an expected result.
- The proper use of pointers can help create efficient programs, but the converse is also true: the improper use of pointers can destroy program efficiency. By limiting the number of pointer operations it supports, a language can prevent the kinds of code inefficiencies that often result from the gratuitous use of pointers.

The major problem with these justifications for limiting pointer operations is that most exist to protect programmers from themselves, and indeed, many programmers (especially beginners) benefit from the discipline these restrictions enforce. However, for careful programmers who do not abuse pointers, these restrictions may eliminate some opportunities for writing great code. Therefore, languages that provide a rich set of pointer operations, like C/C++ and assembly, are popular with advanced programmers who prefer absolute control over the use of pointers in their programs.

9.5 A Simple Memory Allocator Example

To demonstrate the performance and memory costs of using dynamically allocated memory and pointers to it, this section presents a simple memory allocation/deallocation system. By considering the operations associated with memory allocation and deallocation, you'll be more aware of their costs and better equipped to use them in an appropriate way.

An extremely simple (and fast) memory allocation scheme would maintain a single variable that forms a pointer into the heap region of memory. Whenever a memory allocation request comes along, the system makes a copy of this heap pointer to return to the application. The heap management routines add the size of the memory request to the address held in the pointer variable and verify that the memory request won't try to use more memory than is available in the heap. (Some memory managers return an error indication, like a `NULL` pointer, when the memory request is too great; others raise an exception.) The problem with this simple memory management scheme is that it wastes memory because there's no *garbage collection* mechanism for the application to free the memory so it can be reused later. Garbage collection is one of the main purposes of a heap management system.

The only catch is that supporting garbage collection requires some overhead. The memory management code will need to be more sophisticated, will take longer to execute, and will require some additional memory to maintain the internal data structures the heap management system uses. Consider an easy implementation of a heap manager that supports garbage collection on a 32-bit system. This simple system maintains a (linked) list of free memory blocks. Each free memory block in the list requires two `dword` values: one specifying the size of the free block, and the other containing the address of the next free block in the list (that is, the link); see Figure 9-2.

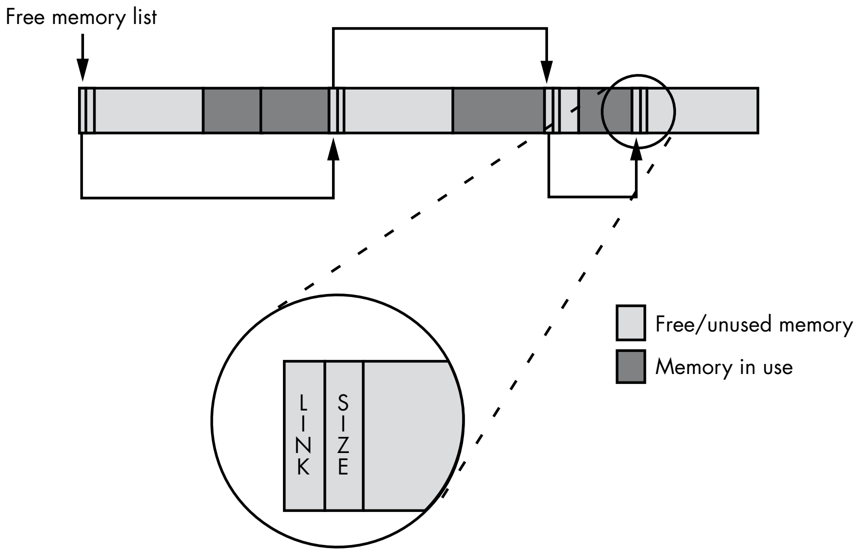


Figure 9-2: Heap management using a list of free memory blocks

The system initializes the heap with a `NULL` link pointer, and the size field contains the size of the heap's entire free space. When a memory allocation request comes along, the heap manager searches through the list to find a free block with enough memory to satisfy the request. This search process is one of the defining characteristics of a heap manager. Some common search algorithms are first-fit search and best-fit search. A *first-fit search*, as its name suggests, scans the list of blocks until it finds the *first* block of memory large enough to satisfy the allocation request. A *best-fit search* scans the entire list and finds the *smallest* block large enough to satisfy the request. The advantage of the best-fit algorithm is that it tends to preserve larger blocks better than the first-fit algorithm, so the system is still able to satisfy larger subsequent allocation requests when they arrive. The first-fit algorithm, on the other hand, just grabs the first suitably large block it finds, even if there's a smaller block that would suffice, which may limit the system's ability to handle future large memory requests.

That said, the first-fit algorithm does have a couple of advantages over the best-fit algorithm. The most obvious is that it is usually faster. The best-fit algorithm has to scan through every block in the free block list in order to find the smallest one large enough to satisfy the allocation request (unless, of course, it finds a perfectly sized block along the way). The first-fit algorithm, on the other hand, can stop once it finds a block large enough to satisfy the request.

Another advantage to the first-fit algorithm is that it tends to suffer less from a degenerate condition known as *external fragmentation*. Fragmentation occurs after a long sequence of allocation and deallocation requests.

Remember, when the heap manager satisfies a memory allocation request, it usually creates two blocks of memory: one in-use block for the request, and one free block that contains the remaining bytes from the original block (assuming the request did not exactly match the block size). After operating for a while, the best-fit algorithm may have produced lots of leftover blocks of memory that are too small to satisfy an average memory request, making them effectively unusable. As these small fragments accumulate throughout the heap, they can end up consuming a fair amount of memory. This can lead to a situation where the heap doesn't have a sufficiently large block to satisfy a memory allocation request even though there is enough total free memory available (spread throughout the heap). See Figure 9-3 for an example of this condition.

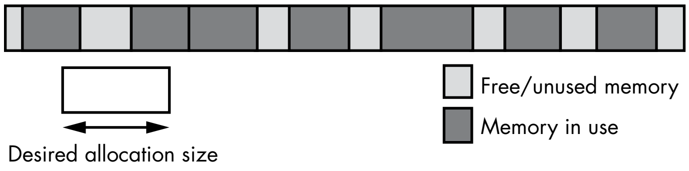


Figure 9-3: Memory fragmentation

There are other memory allocation strategies in addition to the first-fit and best-fit search algorithms. Some of these execute faster, some have less memory overhead, some are easy to understand (and some are very complex), some produce less fragmentation, and some can combine and use noncontiguous blocks of free memory. Memory/heap management is one of the more heavily studied subjects in computer science, and there's a considerable amount of literature explaining the benefits of one scheme over another. For more information on memory allocation strategies, check out a good book on OS design.

9.6 Garbage Collection

Memory allocation is only half of the story. As mentioned earlier, the heap manager also has to provide a call that allows an application to free memory it no longer needs for future reuse—a process known as garbage collection. In C and HLA, for example, an application accomplishes this by calling the `free()` function. At first blush, `free()` might seem to be a very simple function to write. All it has to do is append the previously allocated and now unused block to the end of the free list, right? The problem with this trivial implementation of `free()` is that it almost guarantees that the heap will become fragmented and unusable in very short order. Consider the situation in Figure 9-4.

If `free()` simply takes the block to be freed and appends it to the free list, the memory organization in Figure 9-4 produces three free blocks. However, because these three blocks are contiguous, the heap manager should really combine them into a single free block, so that it will be able to satisfy a larger request. Unfortunately, this operation would require it to scan the free block list to determine if there are any free blocks adjacent to the block the system is freeing. While you could come up with a data structure that makes it easier to combine adjacent free blocks, such schemes generally add 8 or more bytes of overhead with each block on the heap. Whether this is a reasonable tradeoff depends on the average size of a memory allocation. If the applications that use the heap manager tend to allocate small objects, the extra overhead for each memory block could wind up consuming a large percentage of the heap space. However, if most allocations are large, the few bytes of overhead won't matter much.

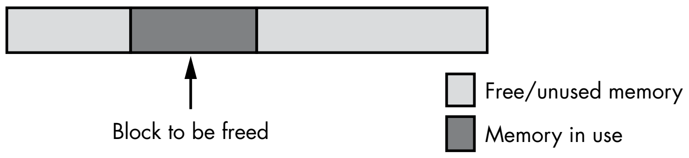


Figure 9-4: Freeing a memory block

9.7 The OS and Memory Allocation

The performance of the algorithms and data structures used by the heap manager is only one piece of the performance puzzle. The heap manager ultimately needs to request blocks of memory from the operating system. At one extreme, the OS handles all memory allocation requests directly. At the other extreme, the heap manager is a runtime library routine that links with your application, first requesting large blocks of memory from the OS and then doling out pieces of them as allocation requests arrive from the application.

The problem with making direct memory allocation requests to the operating system is that OS API calls are often very slow. This is because they generally involve switching between kernel mode and user mode on the CPU (which is not fast). Therefore, a heap manager that the OS implements directly will not perform well if your application makes frequent calls to the memory allocation and deallocation routines.

Because of the high overhead of an OS call, most languages implement their own versions of the `malloc()` and `free()` functions within their runtime library. On the very first memory allocation, the `malloc()` routine requests a large block of memory from the OS, and the application's `malloc()` and `free()` routines manage this block of memory themselves. If an allocation request comes along that the `malloc()` function cannot fulfill in the block it originally created, `malloc()` will request another large block (generally much larger than the request) from the OS and add that block to the end of its free list. Because the application's `malloc()` and `free()` routines call the OS only occasionally, the application doesn't suffer the performance hit associated with frequent OS calls.

However, keep in mind that this procedure is very implementation- and language-specific; it's dangerous to assume that `malloc()` and `free()` are relatively efficient when writing software that requires high-performance components. The only portable way to ensure a high-performance heap manager is to develop your own application-specific set of allocation/deallocation routines. Writing such routines is beyond the scope of this book (and most standard heap management functions perform well for a typical program), but you should know you have this option.

9.8 Heap Memory Overhead

A heap manager often exhibits two types of overhead: performance (speed) and memory (space). Until now, this discussion has mainly dealt with the performance aspects, but now we'll turn our attention to memory.

Each block the system allocates requires some amount of overhead beyond the storage the application requests; at the very least, this overhead is a few bytes to keep track of the block's size. Fancier (higher-performance) schemes may require additional bytes, but typically the overhead is between 8 and 64 bytes. The heap manager can keep this information in a separate internal table, or it can attach the block size and other memory management information directly to the block it allocates.

Saving this information in an internal table has a couple of advantages. First, it is difficult for the application to accidentally overwrite the information stored there; attaching the data to the heap memory blocks themselves doesn't provide as much protection against this possibility. Second, putting memory management information in an internal data structure allows the memory manager to easily determine if a given pointer is valid (that is, points at some block of memory that the heap manager believes it has allocated).

The advantage of attaching the control information directly to each block that the heap manager allocates is that it's very easy to locate this information, whereas storing the information in an internal table might require a search operation.

Another issue that affects the overhead associated with the heap manager is the *allocation granularity*—the minimum number of bytes the heap manager supports. Although most heap managers allow you to request an allocation as small as 1 byte, they may actually allocate some minimum number of bytes greater than 1. Generally, the engineer designing the memory allocation functions chooses a granularity guaranteeing that any object allocated on the heap will begin at a reasonably aligned memory address for that object. Thus, most heap managers allocate memory blocks on a 4-, 8-, or 16-byte boundary. For performance reasons, many heap managers begin each allocation on a cache line boundary, usually 16, 32, or 64 bytes. Whatever the granularity, if the application requests some number of bytes that is less than or not a multiple of the heap manager's granularity, the heap manager will allocate extra bytes of storage (see Figure 9-5). This amount varies by heap manager (and possibly even by version of a specific heap manager), so programmers should never assume that their application has more memory available than they request; if they're tempted to do so, they should request more memory upfront.

The extra memory the heap manager allocates results in another form of fragmentation called *internal fragmentation* (also shown in Figure 9-5). Like external fragmentation, internal fragmentation produces small amounts of leftover memory throughout the system that cannot satisfy future allocation requests. Assuming random-sized memory allocations, the average amount of internal fragmentation that occurs on each allocation is one-half the granularity size. Fortunately, the granularity size is quite small for most memory managers (typically 16 bytes or less), so after thousands and thousands of memory allocations you'll lose only a couple dozen or so kilobytes to internal fragmentation.

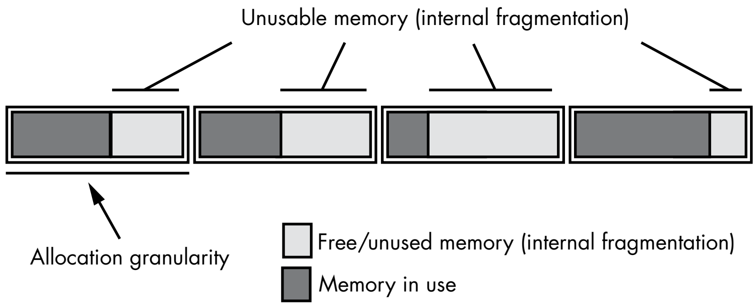

Unusable memory (internal fragmentation)
Allocation granularity
Free/unused memory (internal fragmentation)
Memory in use

Figure 9-5: Allocation granularity and internal fragmentation

Between the costs associated with allocation granularity and the memory control information, a typical memory request may require between 8 and 64 bytes plus whatever the application requests. If you're making large memory allocation requests (hundreds or thousands of bytes), the overhead bytes won't consume a large percentage of memory on the heap. However, if you allocate lots of small objects, the memory consumed by internal fragmentation and memory control information may represent a significant portion of your heap area. For example, consider a simple memory manager that always allocates blocks of data on 4-byte boundaries and requires a single 4-byte length value that it attaches to each allocation request for memory storage. This means that the minimum amount of storage the heap manager requires for each allocation is 8 bytes. If you make a series of `malloc()` calls to allocate a single byte, the application won't be able to use almost 88 percent of the memory it allocates. Even if you allocate 4-byte values on each allocation request, the heap manager consumes two-thirds of the memory for overhead purposes. However, if your average allocation is a block of 256 bytes, the overhead requires only about 2 percent of the total memory allocation. In short, the larger your allocation request, the less impact the control information and internal fragmentation will have on your heap.

Many software engineering studies in computer science journals have found that memory allocation/deallocation requests cause a significant loss of performance. In such studies, the authors often obtained performance improvements of 100 percent or better by simply implementing their own simplified, application-specific, memory management algorithms rather than calling the standard runtime library or OS kernel memory allocation code. Hopefully, this section has made you aware of this potential problem in your own code.

9.9 Common Pointer Problems

Programmers make six common mistakes when using pointers. Some of these mistakes immediately stop a program with a diagnostic message. Others are subtler, yielding incorrect results without otherwise reporting an error. Still others simply negatively affect the program's performance. Great programmers are always aware of the risks of using pointers and avoid these mistakes:

- Using an uninitialized pointer
- Using a pointer that contains an illegal value such as `NULL`
- Continuing to use storage after it has been freed
- Failing to free storage once the program is done using it
- Accessing indirect data using the wrong data type
- Performing invalid pointer operations

9.9.1 Using an Uninitialized Pointer

Using a pointer variable before you've assigned a valid memory address to the pointer is a very common error. Beginning programmers often don't realize that declaring a pointer variable reserves storage only for the pointer itself, not for the data that the pointer references. The following short C/C++ program demonstrates this problem:

```
int main()
{
    static int *pointer;

    *pointer = 0;
}
```

Although static variables you declare are, technically, initialized with `0` (that is, `NULL`), static initialization doesn't initialize the pointer with a valid address. Therefore, when this program executes, the variable pointer won't contain a valid address, and the program will fail. To avoid this problem, ensure that all pointer variables contain a valid address prior to dereferencing those pointers. For example:

```
int main()
{
     static int i;

     static int *pointer = &i;

    *pointer = 0;
}
```

Of course, there's no such thing as a truly uninitialized variable on most CPUs. Variables are initialized in two different ways:

- The programmer explicitly gives them an initial value.
- They inherit whatever bit pattern happens to be in memory when the system binds storage to them.

Much of the time, garbage bit patterns laying around in memory don't correspond to a valid memory address. Attempting to *dereference* such an invalid pointer (that is, to access the data in memory at which it points) raises a Memory Access Violation exception, if your OS is capable of trapping this exception.

Sometimes, however, those random bits in memory just happen to correspond to a valid memory location you can access. In this situation, the CPU accesses the specified memory location without aborting the program. A novice programmer might think that accessing random memory is preferable to aborting a program. However, ignoring the error is far worse because your defective program continues to run without alerting you. If you store data using an uninitialized pointer, you may very well overwrite the values of other important variables in memory. This can produce some problems that are very difficult to locate.

9.9.2 Using a Pointer That Contains an Illegal Value

The second common mistake programmers make with pointers is assigning them invalid values ("invalid" in the sense of not containing the address of an actual object in memory). This can be considered a more general case of the first problem; without initialization, the garbage bits in memory supply the invalid address. The effects are the same. If you attempt to dereference a pointer containing an invalid address, you will either get a Memory Access Violation exception or access an unexpected memory location. Take care when dereferencing a pointer variable and make sure that you've assigned a valid address to the pointer before using it.

9.9.3 Continuing to Use Storage After It Has Been Freed

The third mistake is known as the *dangling pointer problem.* To understand it, consider the following Pascal code fragment:

```
(* Allocate storage for a new object of type p  *)

new( p );

(* Use the pointer *)

p^ := 0;
    .
    . (* Code that uses the storage associated with p *)
    .
```

```
(* free the storage associated with pointer p *)

dispose( p );

    .
    . (* Code that doesn't reference p *)
    .
(* Dangling pointer                              *)

p^ := 5;
```

This program allocates some storage and saves the address of that storage in the p variable. The code uses the storage for a while and then frees it, returning it to the system for other uses. Note that calling dispose() doesn't change any data in the allocated memory. It doesn't change the value of p in any way; p still points at the block of memory allocated earlier by new(). However, calling dispose() does tell the system that the program no longer needs this block of memory so that the system can use the memory for other purposes. The dispose() function cannot enforce the fact that you'll never access this data again, however. You're simply promising that you won't. Of course, this code fragment breaks that promise: the last statement stores the value 5 at the address pointed to by p in memory.

The biggest problem with dangling pointers is that sometimes you can get away with using them, so you won't immediately know there's a problem. As long as the system doesn't reuse the storage you've freed, using a dangling pointer produces no ill effects in your program. However, with each additional call to new(), the system may decide to reuse the memory released by that previous call to dispose(). When it does reuse the memory, any subsequent attempt to dereference the dangling pointer may produce some unintended consequences. The problems can include reading data that has been overwritten, overwriting the new data, and (in the worst case) overwriting system heap management pointers (which will probably cause your program to crash). The solution is clear: never use a pointer value once you free the storage associated with that pointer.

9.9.4 Failing to Free Storage After Using It

Of all these mistakes, failing to free allocated storage probably has the least impact on the proper operation of your program. The following C code fragment demonstrates this problem:

```
// Pointer to storage in "ptr" variable.

ptr = malloc( 256 );
    .
    . // Code that doesn't free "ptr"
    .
ptr = malloc( 512 );

// At this point, there is no way to reference the
// original block of 256 bytes allocated by malloc.
```

In this example, the program allocates 256 bytes of storage and references this storage using the `ptr` variable. Later, the program allocates another block of 512 bytes and overwrites the value in `ptr` with the address of this new block. The former address value in `ptr` is lost. And because the program has overwritten this former value, there's no way to pass the address of the first 256 bytes to the `free()` function. As a result, these 256 bytes of memory are no longer available to your program.

While making 256 bytes of memory inaccessible to your program might not seem like a big deal, imagine that this code executes within a loop. With each iteration of the loop, the program loses another 256 bytes of memory. After a sufficient number of repetitions, the program exhausts the memory available on the heap. This problem is often called a *memory leak* because the effect is as if the memory bits were leaking out of your computer during program execution.

Memory leaks are less of a problem than dangling pointers. Indeed, there are only two problems with memory leaks:

- The danger of running out of heap space (which, ultimately, may cause the program to abort, though this is rare)
- Performance problems due to virtual memory page swapping (*thrashing*)

Nevertheless, freeing all of the storage you allocate is a good habit to develop.

NOTE *When your program quits, the OS will reclaim all of the storage, including the data lost via memory leaks. Therefore, memory lost via a leak is lost only to your program, not the whole system.*

9.9.5 Accessing Indirect Data Using the Wrong Data Type

Another problem with pointers is that their lack of type-safe access makes it easy to accidentally use the wrong data type. Some languages, like assembly, cannot and do not enforce pointer type checking. Others, like C/C++, make it very easy to override the type of the object a pointer references. For example, consider the following C/C++ program fragment:

```
char *pc;
    .
    .
    .
pc = malloc( sizeof( char ));
    .
    .
    .
// Typecast pc to be a pointer to an integer
// rather than a pointer to a character:

*((int *) pc) = 5000;
```

Generally, if you attempt to assign the value 5000 to the object pointed to by pc, the compiler will complain bitterly. The value 5000 won't fit in the amount of storage associated with a character (char) object, which is 1 byte. This example, however, uses *type casting* (or *coercion*) to tell the compiler that pc really contains a pointer to an integer rather than a pointer to a character. Therefore, the compiler will assume that this assignment is legal.

However, if pc doesn't actually point at an integer object, the last statement in this sequence can be disastrous. Characters are 1 byte long, and integers are usually larger. If the integer is larger than 1 byte, this assignment will overwrite some number of bytes beyond the 1 byte of storage that malloc() allocated. Whether or not this is catastrophic depends upon what data immediately follows the character object in memory.

9.9.6 Performing Illegal Operations on Pointers

The last category of common pointer mistakes has to do with operations on the pointers themselves. Arbitrary pointer arithmetic can lead to a pointer that points outside the range of the data originally allocated. By doing some crazy arithmetic, you can even modify a pointer so that it doesn't point at a correct object. Consider the following (really nasty) C code:

```
int  i [4] = {1,2,3,4};
int *p     = &i[0];
      .
      .
      .
    p = (int *)((char *)p + 1);
    *p = 5;
```

This example casts p as a pointer to a char. Then it adds 1 to the value in p. As the compiler thinks that p is pointing at a character (because of the cast), it actually adds the value 1 to the address held in p. The last instruction in this sequence stores the value 5 into the memory address pointed at by p, which is now 1 byte into the 4 bytes set aside for the i[0] element. On some machines, this will cause a fault; on others, it will store a bizarre value into i[0] and i[1].

Comparing two pointers for less than or greater than when the two pointers do not point to the same object (typically an array or struct) is another example of an illegal operation on a pointer, as is casting a pointer as an integer and assigning an integer value to that pointer, which can produce unexpected results.

9.10 Pointers in Modern Languages

Because of the problems described in the previous section, modern HLLs (like Java, C#, Swift, and C++11/C++14) try to eliminate manual memory allocation and deallocation. These languages let you create new objects on the heap (typically using a new() function) but don't provide any facilities for explicitly deallocating that storage. Instead, the language's runtime system tracks memory usage and automatically recovers the storage, via garbage collection, once the program is no longer using it. This eliminates most (but not all) of the problems with uninitialized and dangling pointers. It also lowers the likelihood of memory leaks. These new languages dramatically reduce the number of problems related to errant pointer use.

Of course, ceding control over memory allocation and deallocation introduces some problems of its own. In particular, you give up the ability to control the memory allocation lifetime. Now, the runtime system determines when to garbage-collect unused data, so large chunks of data could still be reserved for some time after you've finished using them.

9.11 Managed Pointers

Some programming languages provide very limited pointer capabilities. For example, standard Pascal allows only a few operations on pointers: assignment (copy), comparison (for equality/inequality), and dereferencing. It does not support pointer arithmetic, meaning many types of mistakes with pointers are impossible.[4] At the other extreme is C/C++, which allows different arithmetic operations on pointers that make the language very powerful but introduce the likelihood of defects in the code.

Modern language systems (for example, C# and the Microsoft Common Language Runtime system) introduce *managed pointers*, which allow various arithmetic operations on pointers, providing greater flexibility than a language like standard Pascal, but with restrictions that help avoid many common pointer pitfalls. For example, in these languages you cannot add an arbitrary integer to an arbitrary pointer (as is possible in C/C++). If you want to add an integer to a pointer and obtain a legal result, the pointer must contain the address of an array object (or other collection of like elements in memory). Furthermore, the integer's value must be limited to a value that does not exceed the size of the data type (that is, the runtime system enforces array bounds checking).

While using managed pointers won't eliminate all pointer problems, it does prevent wiping out data outside the range of a data object referenced by a pointer. It also helps prevent security issues in software, such as attempts to break into a system by providing illegal offsets in pointer arithmetic.

9.12 For More Information

Duntemann, Jeff. *Assembly Language Step-by-Step*. 3rd ed. Indianapolis: Wiley, 2009.

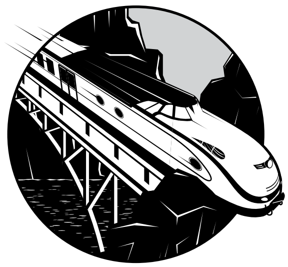
Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

Oualline, Steve. *How Not to Program in C++*. San Francisco: No Starch Press, 2003.

4. However, most real-world Pascal compilers provide extensions that allow pointer arithmetic, so Pascal has all the same issues with pointers as C/C++.

10

STRING DATA TYPES



After integers, character strings are probably the most commonly used data type in modern programs; after arrays, they're the second most commonly used composite data type. A string is a sequence of objects. Most often, the term *string* describes a sequence of character values, but it's also possible to have strings of integers, real values, Boolean values, and so on (for example, I've already discussed bit strings in this book and in *WGC1*). In this chapter, though, we'll stick to character strings.

In general, a character string possesses two main attributes: a *length* and some *character data*. Character strings can also possess other attributes, such as the *maximum length* allowable for that particular variable or a *reference count* specifying how many different string variables refer to the same character string. We'll look at these attributes and how programs can use them, as well as the various string formats and possible string operations. Specifically, this chapter discusses the following topics:

- Character string formats including zero-terminated strings, length-prefixed strings, HLA strings, and 7-bit strings

- When to use (and when not to use) standard library string processing functions
- Static, pseudo-dynamic, and dynamic strings
- Reference counting and strings
- Unicode and UTF-8/UTF-16/UTF-32 character data in strings

String manipulation consumes a fair amount of CPU time in today's applications. Therefore, it's important to understand how programming languages represent and operate on character strings if you want to write code that manipulates strings efficiently. This chapter provides the basic information you'll need to do so.

10.1 Character String Formats

Different languages use different data structures to represent strings. Some string formats use less memory, others allow faster processing, some are more convenient to use, some are easy for compiler writers to implement, and some provide additional functionality for the programmer and operating system.

Although their internal representations vary, all string formats have one thing in common: the character data. This is a sequence of 0 or more bytes (the term *sequence* implies that the order of the characters is important). How a program references this sequence of characters varies by format. In some string formats, the sequence of characters is kept in an array; in other string formats the program maintains a pointer to the sequence of characters elsewhere in memory.

All character string formats share the length attribute; however, they use several different ways to represent the length of a string. Some string formats use a special *sentinel character* to mark the end of the string. Other formats precede the character data with a numeric value that specifies the number of characters in the sequence. Still others encode the length as a numeric value in a variable that is not connected to the character sequence. Some character string formats use a special bit (set or cleared) to mark the end of a string. Finally, some string formats use a combination of these methods. How a particular string format determines the length of a string can have a big impact on the performance of the functions that manipulate those strings. It can also affect how much extra storage is needed to represent string data.

Some string formats provide additional attributes, such as a maximum length and reference count values, that certain string functions can use to operate on string data more efficiently. These extra attributes are optional insofar as they aren't strictly necessary to define a string value. They do, however, allow string manipulation functions to provide certain tests for correctness or to work more efficiently than they would otherwise.

To help you better understand the reasoning behind the design of character strings, let's look at some common string representations popularized by various languages.

10.1.1 Zero-Terminated Strings

Without question, *zero-terminated strings* (see Figure 10-1) are probably the most common string representation in use today, because this is the native string format for C, C++, and several other languages. In addition, you'll find zero-terminated strings used in programs written in languages that don't have a specific native string format, such as assembly language.

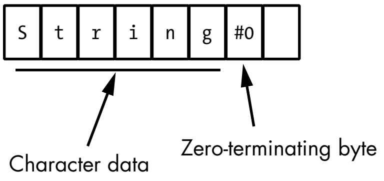


Figure 10-1: Zero-terminated string format

A zero-terminated ASCII string, also called an *ASCIIz* string or a *zstring*, is a sequence containing zero or more 8-bit character codes and ending with a byte containing `0`—or, in the case of Unicode (UTF-16), a sequence containing zero or more 16-bit character codes and ending with a 16-bit word containing `0`. For UTF-32 strings, each item in the string is 32 bits (4 bytes) wide, ending with a 32-bit `0` value. For example, in C/C++, the ASCIIz string `"abc"` requires 4 bytes: 1 byte for each of the three characters `a`, `b`, and `c`, followed by a `0` byte.

Zero-terminated strings have a few advantages over other string formats:

- Zero-terminated strings can represent strings of any practical length with only 1 byte of overhead (2 bytes in UTF-16, 4 in UTF-32).
- Given the popularity of the C/C++ programming languages, high-performance string processing libraries are available that work well with zero-terminated strings.
- Zero-terminated strings are easy to implement. Indeed, except for dealing with string literal constants, the C/C++ programming languages don't provide native string support. As far as those languages are concerned, strings are just arrays of characters. That's probably why C's designers chose this format in the first place—so they wouldn't have to clutter up the language with string operators.
- You can easily represent zero-terminated strings in any language that provides the ability to create an array of characters.

However, zero-terminated strings also have disadvantages that mean they are not always the best choice for representing character string data:

- String functions often aren't very efficient when operating on zero-terminated strings. Many string operations need to know the length of the string before working on the string data. The only reasonable way to compute the length of a zero-terminated string is to scan the string from the beginning to the end. The longer your strings are, the slower this function runs, so the zero-terminated string format isn't the best choice if you need to process long strings.

- Although it's a minor problem, you cannot easily represent the character code `0` (such as the NUL character in ASCII and Unicode) with the zero-terminated string format.
- Zero-terminated strings don't contain any information that tells you how long the string can grow beyond the terminating `0` byte. Therefore, some string functions, like concatenation, can only extend the length of an existing string variable and check for overflow if the caller explicitly passes the maximum length.

As noted, one nice feature of zero-terminated strings is that you can easily implement them using pointers and arrays of characters. Consider the following C/C++ statement:

```
someCharPtrVar = "Hello World";
```

Here's the code the Borland C++ v5.0 compiler generates for this statement:

```
;       char *someCharPtrVar;
   ;       someCharPtrVar = "Hello World";
   ;
@1:
; "offset" means "take the address of" and "s@" is
; the compiler-generated label where the string
; "Hello World" can be found.

    mov       eax,offset s@
        .
        .
        .
_DATA   segment dword public use32 'DATA'
;       s@+0:
        ; Zero-terminated sequence of characters
        ; emitted for the literal string "Hello World":

s@      label   byte
        db      "Hello World",0

        ;       s@+12:
        db      "%s",0
        align   4
_DATA   ends
```

The Borland C++ compiler simply emits the literal string `"Hello World"` to the global data segment in memory and then loads the `someCharPtrVar` variable with the address of the first character of this string literal in the data segment. From that point forward, the program can refer to the string data indirectly via this pointer. This is a very convenient scheme from the compiler writer's point of view.

When using zero-terminated strings in a language like C, C++, Python, or any of a dozen other languages that have adopted C's string format, you can improve the performance of your string-handling code sequences by keeping a few points in mind:

- Try to use the language's runtime library functions rather than attempting to code comparable functions yourself. Most compiler vendors provide highly optimized versions of their string functions that will probably run many times faster than code you would write yourself.
- Once you've computed the length of a string by scanning the entire string, save that length for future use (rather than recomputing it every time you need it).
- Avoid copying string data from one string variable to another. Doing so is one of the more expensive operations (after length computation) in applications using zero-terminated strings.

The following subsections discuss each point in turn.

10.1.1.1 When to Use C Standard Library String Functions

Some programmers are skeptical that someone else could write faster or higher-quality code. But when it comes to standard library functions, you should avoid the temptation to replace them with code of your own choosing. Unless the library code you're considering is especially bad, chances are you won't come close to duplicating its efficiency. This is especially true for string functions that handle zero-terminated strings in languages like C and C++. There are three main reasons why standard libraries generally perform better than code you write yourself: experience, maturity, and inline substitution.

The typical programmer who writes compiler runtime libraries has a lot of experience with string-handling functions. Although in the past new compilers were often accompanied by notoriously inefficient libraries, over time compiler programmers have gained considerable experience writing those library routines and have figured out how to deliver well-written string-handling functions. Unless you've spent considerable time writing those same types of routines, it's highly unlikely that your code will perform as well as theirs. Many compiler vendors purchase their standard library code from a third party that specializes in writing library code, so now, even if the compiler you're using is fairly new, it may have a good library. Few commercial compilers today contain horribly inefficient library code. For the most part, only research or "hobby" compilers contain library code so bad that you can easily write something better. Consider a simple example—the C standard library `strlen()` (string length) function. Here's a typical implementation of `strlen()` that an inexperienced programmer might write:

```
#include <stdlib.h>
#include <stdio.h>
```

```
int myStrlen( char *s )
{
    char *start;

    start = s;
    while( *s != 0 )
    {
        ++s;
    }
    return s - start;
}

int main( int argc, char **argv )
{


    printf( "myStrlen = %d", myStrlen( "Hello World" ));
    return 0;
}
```

The 80x86 machine code that Microsoft's Visual C++ compiler generates for `myStrlen()` is probably what any assembly programmer would expect:

```
myStrlen PROC                                           ; COMDAT
; File c:\users\rhyde\test\t\t\t.cpp
; Line 10                      // Pointer to string (s) is passed in RCX register.
        cmp     BYTE PTR [rcx], 0   // Is *s = 0?
        mov     rax, rcx            // Save ptr to start of string to compute length
        je      SHORT $LN3@myStrlen // Bail if we hit the end of the string
$LL2@myStrlen:
; Line 12
        inc     rcx                 // Move on to next char in string
        cmp     BYTE PTR [rcx], 0   // Hit the 0 byte yet?
        jne     SHORT $LL2@myStrlen // If not, repeat loop
$LN3@myStrlen:
; Line 14
        sub     rcx, rax            // Compute length of string.
        mov     eax, ecx            // Return function result in EAX.
; Line 15
        ret     0
myStrlen ENDP
```

No doubt, an experienced assembly language programmer could rearrange these particular instructions to speed them up a bit. Indeed, even an average 80x86 assembly language programmer could point out that the 80x86 `scasb` instruction does most of the work in this code sequence. Although this code is fairly short and easy to understand, by no means will it run as fast as possible. An expert assembly language programmer might note that this loop repeats one iteration for each character in the string and accesses the characters in memory 1 byte at a time, and might improve upon it by unrolling[1] the loop and processing more than one character per loop iteration. For example, consider the following HLA standard library `zstr.len()` function, which computes the length of a zero-terminated string by processing four characters at a time:

```
unit stringUnit;

#include( "strings.hhf" );


/*******************************************************************/
/*                                                                 */
/* zlen-                                                           */
/*                                                                 */
/* Returns the current length of the z-string passed as a parm.    */
/*                                                                 */
/*******************************************************************/

procedure zstr.len( zstr:zstring ); @noframe;
const
    zstrp   :text := "[esp+8]";

begin len;

    push( esi );
    mov( zstrp, esi );

    // We need to get ESI dword-aligned before proceeding.
    // If the LO 2 bits of ESI contain 0s, then
    // the address in ESI is a multiple of 4. If they
    // are not both 0, then we need to check the 1,
    // 2, or 3 bytes starting at ESI to see if they
    // contain a zero-terminator byte.

    test( 3, esi );
    jz ESIisAligned;

    cmp( (type char [esi]), #0 );
    je SetESI;
    inc( esi );
    test( 3, esi );
    jz ESIisAligned;

    cmp( (type char [esi]), #0 );
    je SetESI;
    inc( esi );
    test( 3, esi );
    jz ESIisAligned;

    cmp( (type char [esi]), #0 );
```

1. *Unrolling* is an optimization technique that speeds up execution time by eliminating loop control instructions and loop test instructions.

```
    je SetESI;
    inc( esi );                 // After this, ESI is aligned.


    ESIisAligned:
        sub( 32, esi );         // To counteract add immediately below.
    ZeroLoop:
        add( 32, esi );         // Skip chars this loop just processed.
    ZeroLoop2:
        mov( [esi], eax );      // Get next four chars into EAX.
        and( $7f7f7f7f, eax );  // Clear HO bit (note:$80->$00!)
        sub( $01010101, eax );  // $00 and $80->$FF, all others have pos val.
        and( $80808080, eax );  // Test all HO bits.  If any are set, then
        jnz MightBeZero0;       // we've got a $00 or $80 byte.

        mov( [esi+4], eax );    // The following are all inline expansions
        and( $7f7f7f7f, eax );  // of the above (we'll process 32 bytes on
        sub( $01010101, eax );  // each iteration of this loop).
        and( $80808080, eax );
        jnz MightBeZero4;

        mov( [esi+8], eax );
        and( $7f7f7f7f, eax );
        sub( $01010101, eax );
        and( $80808080, eax );
        jnz MightBeZero8;

        mov( [esi+12], eax );
        and( $7f7f7f7f, eax );
        sub( $01010101, eax );
        and( $80808080, eax );
        jnz MightBeZero12;

        mov( [esi+16], eax );
        and( $7f7f7f7f, eax );
        sub( $01010101, eax );
        and( $80808080, eax );
        jnz MightBeZero16;

        mov( [esi+20], eax );
        and( $7f7f7f7f, eax );
        sub( $01010101, eax );
        and( $80808080, eax );
        jnz MightBeZero20;

        mov( [esi+24], eax );
        and( $7f7f7f7f, eax );
        sub( $01010101, eax );
        and( $80808080, eax );
        jnz MightBeZero24;

        mov( [esi+28], eax );
        and( $7f7f7f7f, eax );
        sub( $01010101, eax );
        and( $80808080, eax );
        jz ZeroLoop;

    // The following code handles the case where we found a $80
    // or a $00 byte. We need to determine whether it was a 0
    // byte and the exact position of the 0 byte. If it was a
    // $80 byte, then we've got to continue processing characters
    // in the string.


    // Okay, we've found a $00 or $80 byte in positions
    // 28..31. Check for the location of the 0 byte, if any.

        add( 28, esi );
        jmp MightBeZero0;

    // If we get to this point, we've found a 0 byte in
    // positions 4..7:

    MightBeZero4:
        add( 4, esi );
        jmp MightBeZero0;

    // If we get to this point, we've found a 0 byte in
    // positions 8..11:

    MightBeZero8:
        add( 8, esi );
        jmp MightBeZero0;

    // If we get to this point, we've found a 0 byte in
    // positions 12..15:

    MightBeZero12:
        add( 12, esi );
        jmp MightBeZero0;

    // If we get to this point, we've found a 0 byte in
    // positions 16..19:

    MightBeZero16:
        add( 16, esi );
        jmp MightBeZero0;

    // If we get to this point, we've found a 0 byte in
    // positions 20..23:

    MightBeZero20:
        add( 20, esi );
        jmp MightBeZero0;

    // If we get to this point, we've found a 0 byte in
    // positions 24..27:

    MightBeZero24:
        add( 24, esi );
    // If we get to this point, we've found a 0 byte in
    // positions 0..3 or we've branched here from one of the
    // above conditions

    MightBeZero0:
        mov( [esi], eax );          // Get the original 4 bytes.
        cmp( al, 0 );               // See if the first byte contained 0.
        je SetESI;
        cmp( ah, 0 );               // See if the second byte contained 0.
        je SetESI1;
        test( $FF_0000, eax );      // See if byte #2 contained a 0.
        je SetESI2;
        test( $FF00_0000, eax );    // See if the HO byte contained 0.
        je SetESI3;

    // Well, it must have been a $80 byte we encountered.
    // (Fortunately, they are rare in ASCII strings, so all this
    // extra computation rarely occurs). Jump back into the 0
    // loop and continue processing.

        add( 4, esi );              // Skip bytes we just processed.
        jmp ZeroLoop2;              // Don't bother adding 32 in the ZeroLoop!

    // The following computes the length of the string by subtracting
    // the current ESI value from the original value and then adding
    // 0, 1, 2, or 3, depending on where we branched out
    // of the MightBeZero0 sequence above.

    SetESI3:
        sub( zstrp, esi );          // Compute length
        lea( eax, [esi+3] );        // +3 since it was in the HO byte.
        pop( esi );
        ret(4);

    SetESI2:
        sub( zstrp, esi );          // Compute length
        lea( eax, [esi+2] );        // +2 since zero was in byte #2
        pop( esi );
        ret(4);

    SetESI1:
        sub( zstrp, esi );          // Compute length
        lea( eax, [esi+1] );        // +1 since zero was in byte #1
        pop( esi );
        ret(4);

    SetESI:
        mov( esi, eax );
        sub( zstrp, eax );          // Compute length. No extra addition since
        pop( esi );                 // 0 was in LO byte.
        ret( _parms_ );

end len;
end stringUnit;
```

Even though this function is much longer and more complex than the simple example given earlier, it runs faster because it processes four characters per loop iteration rather than one, which means it executes far fewer loop iterations. Also, this code reduces loop overhead by unrolling eight copies of the loop (that is, expanding eight copies of the loop body inline), which saves the execution of 87 percent of the loop control instructions. As a result, this code runs anywhere from two to six times faster than the code given earlier; the exact savings depend upon the length of the string.[2]

The second reason to avoid writing your own library functions is the maturity of the code. Most popular optimizing compilers available today have been around for a while. During this time, the compiler vendors have used their routines, determined where the bottlenecks lie, and optimized their code. When you write your own version of a standard library string-handling function, you probably won't have comparable time to dedicate to optimizing it—you've got your entire application to worry about. Because of project time constraints, you'll likely never go back and rewrite that string function to improve its performance. Even if there's a slight performance advantage to your routine now, the compiler vendor may very well update their library in the future, and you could take advantage of those improvements by simply relinking the updated code with your project. However, if you write the library code yourself, it will never improve unless you explicitly update it yourself. Most people are too busy working on new projects to go back and clean up their old code, so the likelihood of improving self-written string functions in the future is quite low.

The third reason for using standard library string functions in a language like C or C++ is the most important: inline expansion. Many compilers recognize certain standard library function names and expand them inline to efficient machine code in place of the function call. This inline expansion can be many times faster than an explicit function call, especially if the function call contains several parameters. As a simple example, consider the following (almost trivial) C program:

```c
#include <string.h>
#include <stdio.h>

int main( int argc, char **argv )
{
    char localStr[256];

    strcpy( localStr, "Hello World" );
    printf( localStr );
    return 0;
}
```

2. It's worth pointing out that this code is not an exact replacement for the simplistic C code given in this section. The HLA code assumes that all strings are padded to a multiple of 4 bytes in length (a reasonable assumption in HLA). This isn't necessarily true for standard C strings. Also, this is far from being as efficient as it could be. Newer CPUs with SSE4.1 extensions can use certain SSE instructions to execute this operation even faster.

The corresponding 64-bit x86-64 assembly code that Visual C++ produces is quite interesting:

```
; Storage for the literal string appearing in the
; strcpy invocation:

_DATA   SEGMENT
$SG6874 DB  'Hello World', 00H
_DATA   ENDS


_TEXT   SEGMENT
localStr$ = 32
__$ArrayPad$ = 288
argc$ = 320
argv$ = 328
main    PROC
; File c:\users\rhyde\test\t\t\t.cpp
; Line 6
$LN4:
    sub rsp, 312                ; 00000138H
    mov rax, QWORD PTR __security_cookie
    xor rax, rsp
    mov QWORD PTR __$ArrayPad$[rsp], rax
; Line 9
    movsd   xmm0, QWORD PTR $SG6874
; Line 10
    lea rcx, QWORD PTR localStr$[rsp]
    mov eax, DWORD PTR $SG6874+8
    movsd   QWORD PTR localStr$[rsp], xmm0
    mov DWORD PTR localStr$[rsp+8], eax
    call    printf
; Line 11
    xor eax, eax
; Line 12
    mov rcx, QWORD PTR __$ArrayPad$[rsp]
    xor rcx, rsp
    call    __security_check_cookie
    add rsp, 312                ; 00000138H
    ret 0
main    ENDP
_TEXT   ENDS
```

The compiler recognizes what's going on and substitutes four inline instructions that copy the 12 bytes of the string from the literal constant in memory to the `localStr` variable (specifically, it copies 8 bytes using the XMM0 register and 4 bytes using the EAX register; note that this code uses RCX to pass the address of `localStr` to the `printf()` function). The overhead of a call and return to an actual `strcpy()` function will be more expensive than this (and that's without considering the work needed to copy the string data). This example demonstrates quite well why you should usually call standard library functions rather than writing your own "optimized" functions to do the same job.

10.1.1.2 When Not to Use Standard Library Functions

Although, as you've seen, it's usually better to call a standard library routine rather than writing your own version, there are some special situations when you should *not* rely on one or more library functions in the standard library.

Library functions work great when they perform exactly the function you need—no more and no less. One area where programmers get into trouble is when they misuse a library function and call it to do something that it wasn't really intended to do, or they need only part of the functionality it provides. For example, consider the C standard library `strcspn()` function:

```
size_t strcspn( char *source, char *cset );
```

This function returns the number of characters in the *`source`* string up to the first character it finds that also appears in the *`cset`* string. It's not at all uncommon to see calls to this function that look like this:

```
len = strcspn( SomeString, "a" );
```

The intent here is to return the number of characters in *`SomeString`* before the first occurrence of an `a` character in that string. That is, it attempts to do something like the following:

```
len = 0;
while
(
        SomeString[ len ] != '\0'
    &&  SomeString[ len ] != 'a'
){
    ++len;
}
```

Unfortunately, the call to the `strcspn()` function is probably a lot slower than this simple `while` loop implementation. That's because `strcspn()` actually does a lot more work than search for a single character within a string. It looks for any character from a set of characters within the source string. The generic implementation of this function might be something like:

```
len = 0;
for(;;) // Infinite loop
{
    ch = SomeString[ len ];
    if( ch == '\0' ) break;
    for( i=0; i<strlen( cset ); ++i )
    {
        if( ch == cset[i] ) break;
    }
    if( ch == cset[i] ) break;
    ++len;
}
```

With a little analysis (and noting that we have a pair of nested loops here), it's clear that this code is slower than the code given earlier, even if you pass in a cset string containing a single character. This is a classic example of calling a function that is more general than you need, because it searches for any of several termination characters rather than the special case of a single terminating character. When a function does exactly what you want, using the standard library's version of it is a good idea. However, when it does more than you need, using the standard library function can be expensive, and it's better to write your own version.

10.1.1.3 Why to Avoid Length Recomputing Data

The last example in the previous section demonstrates a common C programming mistake. Consider the coded fragment:

```
for( i=0; i<strlen( cset ); ++i )
{
    if( ch == cset[i] ) break;
}
```

On each iteration of this loop, the code tests the loop index to see if it is less than the length of the cset string. But because the loop body does not modify the cset string (and because, presumably, this is not a multithreaded application with another thread modifying the cset string), there's really no need to recompute the string length on each iteration of this loop. Look at the code that the Microsoft Visual C++ 32-bit compiler emits for this code fragment:

```
; Line 10
        mov     DWORD PTR i$1[rsp], 0 ;for(i = 0;...;...)
        jmp     SHORT $LN4@main

$LN2@main:
        mov     eax, DWORD PTR i$1[rsp] ;for(...;...;++i)
        inc     eax
        mov     DWORD PTR i$1[rsp], eax

$LN4@main: ;for(...; i < strlen(localStr);...)
        movsxd  rax, DWORD PTR i$1[rsp]
        mov     QWORD PTR tv65[rsp], rax
        lea     rcx, QWORD PTR localStr$[rsp]
        call    strlen
        mov     rcx, QWORD PTR tv65[rsp]
        cmp     rcx, rax
        jae     SHORT $LN3@main
; Line 12
        movsx   eax, BYTE PTR ch$[rsp]
        movsxd  rcx, DWORD PTR i$1[rsp]
        movsx   ecx, BYTE PTR localStr$[rsp+rcx]
        cmp     eax, ecx
        jne     SHORT $LN5@main
        jmp     SHORT $LN3@main
```

```
$LN5@main:
; Line 13
        jmp     SHORT $LN2@main
$LN3@main:
```

Again, the machine code recalculates the string's length on every iteration of the innermost for loop, but because the cset string's length never changes, this is totally unnecessary. We can easily rectify this problem by rewriting the code fragment this way:

```
slen = strlen( cset );
len = 0;
for(;;) // Infinite loop
{
    ch = SomeString[ len ];
    if( ch == '\0' ) break;
    for( i=0; i<slen; ++i )
    {
        if( ch == cset[i] ) break;
    }
    if( ch == cset[i] ) break;
    ++len;
}
```

On the plus side, recent versions of Microsoft's VC++ compiler will recognize this situation if you have optimizations turned on. As VC++ determines that the string length is a loop-invariant calculation (that is, its value does not change from one loop iteration to the next), VC++ will move the call to strlen() out of the loop. Unfortunately, VC++ can't catch this in every situation. For example, if you call some function that VC++ doesn't know about and you pass it the address of localStr as a (non-const) parameter, VC++ will have to assume that the string's length could change (even if it doesn't) and it won't be able to move the strlen() call out of the loop.

A fair number of string operations require the string's length before they can execute. Consider the strdup() function commonly found in many C libraries.[3] The following code is a common implementation of this function:

```
char *strdup( char *src )
{
    char *result;

    result = malloc( strlen( src ) + 1 );
    assert( result != NULL ); // Check malloc check
    strcpy( result, src );
    return result;
}
```

3. The strdup() function is not defined in the original C standard library, but it's very common for vendors to include it as an extension to the C standard library.

Fundamentally, nothing is wrong with this implementation of strdup(). If you know absolutely nothing about the string object you're passing as a parameter, then you must compute the string's length so you know how much memory to allocate for a copy of that string. Consider, however, the following code sequence that calls strdup():

```
len = strlen( someStr );
if( len == 0 )
{
    newStr = NULL;
}
else
{
    newStr = strdup( someStr );
}
```

The problem here is that you wind up calling strlen() twice: once for the explicit call to strlen() in this code fragment, and once for the call buried in the strdup() function. Worse, it isn't obvious that you're calling strlen() twice, so it's not even clear that you're wasting CPU cycles in this code. This is another example of calling a function that is more general than you need, causing the program to recompute the string's length (an inefficient process). One solution is to provide a less general version of strdup(), say strduplen(), that lets you pass it the length of the string you've already computed. You could implement strduplen() as follows:

```
char *strduplen( char *src, size_t len)
{
    char *result;

    // Allocate storage for new string:

    result = malloc( len + 1 );
    assert( result != NULL );

    // Copy the source string and
    // 0 byte to the new string:

    memcpy( result, src, len+1 );
    return result;
}
```

Notice the use of memcpy() rather than strcpy() (or, better yet, strncpy()). Again, we already know the length of the string, so there's no need to execute any code looking for the 0 terminating byte (as both strcpy() and strncpy() will do). Of course, this function implementation assumes that the caller passes the correct length, but that's a standard C assumption for most string and array operations.

10.1.1.4 Why to Avoid Copying Data

Copying strings, especially long strings, can be a time-consuming process on a computer. Most programs maintain string data in memory, and memory is much slower than the CPU (often by an order of magnitude or more). Although cache memory can help mitigate this problem, processing a lot of string data can eliminate other data from the cache and lead to thrashing problems if you don't frequently reuse all the string data you move through the cache. It's not always possible to avoid moving string data around, but many programs needlessly copy data, and that can hamper program performance.

A better solution is to pass around *pointers* to zero-terminated strings rather than copying those strings from string variable to string variable. Pointers to zero-terminated strings can fit in registers and don't consume much memory when you use memory variables to hold them. Therefore, passing pointers has far less impact on cache and CPU performance than copying string data among string variables.

As you've seen in this section, zero-terminated string functions are generally less efficient than functions that manipulate other types of strings. Furthermore, programs that utilize zero-terminated strings tend to make mistakes, such as calling `strlen()` multiple times or abusing generic functions to achieve specific goals. Fortunately, designing and using a more efficient string format is easy enough in languages whose native string format is the zero-terminated string.

10.1.2 Length-Prefixed Strings

A second common string format, *length-prefixed strings*, overcomes some of the problems with zero-terminated strings. Length-prefixed strings are common in languages like Pascal; they generally consist of a single byte that specifies the length of the string, followed by zero or more 8-bit character codes (see Figure 10-2). In a length-prefixed scheme, the string `"String"` would consist of 4 bytes: the length byte (`6`), followed by the characters `S`, `t`, `r`, `i`, `n`, and `g`.

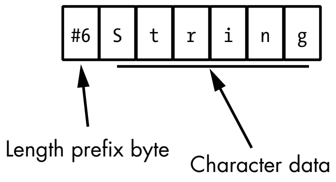


Figure 10-2: Length-prefixed string format

Length-prefixed strings solve two of the problems associated with zero-terminated strings:

- NUL characters can be represented in length-prefixed strings.
- String operations are more efficient.

Another advantage to length-prefixed strings is that the length is usually located at position `0` in the string (if we view the string as an array of characters), so the first character of the string begins at index `1` in the array representation of the string. For many string functions, having a `1`-based index into the character data is much more convenient than a `0`-based index (which zero-terminated strings use).

Length-prefixed strings do suffer from their own drawbacks, the principal one being that they're limited to a maximum of 255 characters in length (assuming a 1-byte length prefix). You can remove this limitation by using a 2- or 4-byte length value, but doing so increases the amount of overhead data from 1 to 2 or 4 bytes. It also changes the starting index of the string from 1 to either 2 or 4, eliminating the `1`-based index feature. While there are ways to overcome this problem, they entail even more overhead.

Many string functions are much more efficient with length-prefixed strings. Obviously, computing the length of a string is a trivial operation—it's just a memory access—but other string functions that ultimately need the string's length (such as concatenation and assignment) are usually more efficient than similar functions for zero-terminated strings. Furthermore, you don't have to worry about recomputing the string's length every time you call a string function that is built into the language's standard library.

Despite these advantages, don't get the impression that programs using length-prefixed string functions are always going to be efficient. You can still waste many CPU cycles by needlessly copying data. As with zero-terminated strings, if you use only a subset of a string function's capabilities, you can waste lots of CPU cycles performing unnecessary tasks.

When using length-prefixed string functions, keep the following points in mind:

- Try to use the language's runtime library functions rather than attempting to code comparable functions yourself. Most compiler vendors provide highly optimized versions of their string functions that will probably run many times faster than code you would write yourself.
- Although computing the string length when using the length-prefixed string format is fairly trivial, many (Pascal) compilers actually emit a function call to extract the length value from the string's data. The function call and return is far more expensive than retrieving the length value from a variable. So, once you compute the string's length, consider saving that length in a local variable if you intend to use that value again. Of course, if a compiler is smart enough to replace a call to the length function with a simple data fetch from the string's data structure, this "optimization" won't buy you much.
- Avoid copying string data from one string variable to another. Doing so is one of the more expensive operations in programs using length-prefixed strings. Passing around pointers to strings has the same benefit as for zero-terminated strings.

10.1.3 Seven-Bit Strings

The 7-bit string format is an interesting option that works for 7-bit encodings like ASCII. It uses the (normally unused) higher-order bit of the characters in the string to indicate the end of the string. All but the last character code in the string has its HO bit clear, and the last character in the string has its HO bit set (see Figure 10-3).

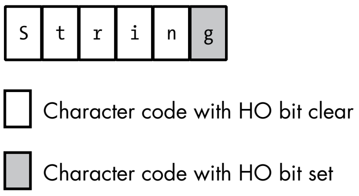

S t r i n g
Character code with HO bit clear
Character code with HO bit set

Figure 10-3: Seven-bit string format

This 7-bit string format has several disadvantages:

- You have to scan the entire string in order to determine the length of the string.
- You cannot have zero-length strings in this format.
- Few languages provide literal string constants for 7-bit strings.
- You're limited to a maximum of 128 character codes, although this is fine when you are using plain ASCII.

However, the big advantage of 7-bit strings is that they don't require any overhead bytes to encode the length. Assembly language (using a macro to create literal string constants) is probably the best language to use when dealing with 7-bit strings. Because the benefit of 7-bit strings is that they're compact and assembly language programmers tend to worry most about compactness, this is a good match. Here's an HLA macro that converts a literal string constant to a 7-bit string:

```
#macro sbs( s );

    // Grab all but the last character of the string:

    (@substr( s, 0, @length(s) - 1) +

        // Concatenate the last character
        // with its HO bit set:

        char
        (
            uns8
            (
               char( @substr( s, @length(s) - 1, 1))
            ) | $80
        )
    )
```

```
#endmacro
    .
    .
    .
byte sbs( "Hello World" );
```

Because few languages provide support for 7-bit strings, the first suggestion that applied to zero-terminated and length-prefixed strings doesn't apply to 7-bit strings: you'll probably have to write your own string-handling functions. Computing lengths and copying data are expensive operations even with 7-bit strings, however, so these two suggestions still apply:

- Once you've computed the length of a string by scanning the entire string, save that length for future use (rather than recomputing it every time you need it).
- Avoid copying string data from one string variable to another. Doing so is one of the more expensive operations in programs using 7-bit strings.

10.1.4 HLA Strings

As long as you're not too concerned about a few extra bytes of overhead per string, you can create a string format that combines the advantages of both length-prefixed and zero-terminated strings without their respective disadvantages. The High-Level Assembly language has done this with its native string format.[4]

The biggest drawback to the HLA character string format is the amount of overhead required for each string (which can be significant, percentage-wise, if you're in a memory-constrained environment and you process many small strings). HLA strings contain a length prefix and a zero-terminating byte, as well as some other information, totaling 9 bytes of overhead per string.[5]

The HLA string format uses a 4-byte length prefix, allowing character strings to be just over 4 billion characters long (far more than any practical application will use). HLA also appends a `0` byte to the character string data, so HLA strings are compatible with string functions that reference (but do not change the length of) zero-terminated strings. The remaining 4 bytes of overhead in an HLA string contain the maximum legal length for that string (plus a `0` terminating byte). Having this extra field allows HLA string functions to check for string overflow, if necessary. In memory, HLA strings take the form shown in Figure 10-4.

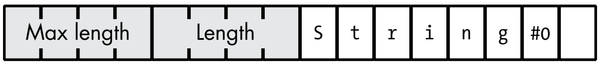


Figure 10-4: HLA string format

4. Note that HLA is an assembly language, so it's perfectly possible—and easy in fact—to support any reasonable string format. HLA's native string format is what it uses for literal string constants, and what most of the routines in the HLA standard library support.

5. Actually, because of memory alignment restrictions, there can be up to 12 bytes of overhead, depending on the string.

The 4 bytes immediately before the first character of the string contain the current string length. The 4 bytes preceding the current string length contain the maximum string length. Immediately following the character data is a 0 byte. Finally, HLA always ensures that the string data structure's length is a multiple of 4 bytes for performance reasons, so there may be up to 3 additional bytes of padding at the end of the object in memory. (Note that the string shown in Figure 10-4 requires only 1 byte of padding to ensure that the data structure is a multiple of 4 bytes in length.)

HLA string variables are actually pointers that contain the byte address of the first character in the string. To access the length fields, you load the value of the string pointer into a 32-bit register, then access the length field at offset –4 from the register and the maximum length field at offset –8 from the register. Here's an example:

```
static
    s :string := "Hello World";
        .
        .
        .
// Move the address of 'H' in
// "Hello World" into esi.

mov( s, esi );

// Puts length of string
// (11 for "Hello World") into ECX.

mov( [esi-4], ecx );
        .
        .
        .
mov( s, esi );

// See if value in ECX exceeds the
// maximum string length.

cmp( ecx, [esi-8] );
jae StringOverflow;
```

As noted earlier, the amount of memory reserved to hold an HLA string's character data (including the 0 byte) is always a multiple of 4 bytes. Therefore, it's always guaranteed that you can move data from one HLA string to another by copying double words rather than individual bytes. This allows string copy routines to run up to four times faster, because you execute one-fourth the number of loop iterations copying a string of double words as you would copying the string a byte at a time. For example, here's the highly modified version of the pertinent code in the HLA `str.cpy()` function that copies one string to another:

```
// Get the source string pointer into ESI,
// and the destination pointer into EDI.
```

```
    mov( dest, edi );
    mov( src, esi );

    // Get the length of the source string
    // and make sure that the source string
    // will fit in the destination string.

    mov( [esi-4], ecx );

    // Save as the length of the destination string.

    mov( ecx, [edi-4] );


    // Add 1 byte to the length so we will
    // copy the 0 byte. Also compute the
    // number of dwords to copy (rather than bytes).
    // Then copy the data.

    add( 4, ecx );  // Adds one, after division by 4.
    shr( 2, ecx );  // Divides length by 4
    rep.movsd();    // Moves length/4 dwords
```

The HLA `str.cpy()` function also checks for string overflows and `NULL` pointer references (for clarity, that code does not appear in this example). However, the takeaway here is that HLA copies the strings as double words in order to improve performance.

One nice thing about HLA string variables is that (as read-only objects) HLA strings are compatible with zero-terminated strings. For example, if you have a function written in C or some other language that expects you to pass a zero-terminated string to it, you can call that function and pass an HLA string variable to it, like this:

```
someCFunc( hlaStringVar );
```

The only catch is that the C function must not make any changes to the string that would affect its length (because the C code won't update the `Length` field of the HLA string). Of course, you can always call a C `strlen()` function upon returning to update the length field yourself, but generally, it's best not to pass HLA strings to a function that modifies zero-terminated strings.

The comments on length-prefixed strings generally apply to HLA strings, specifically:

- Try to use the HLA standard library functions rather than attempting to code comparable functions yourself. While you might want to check out the library function's source code (available with HLA), most of the string functions do a good job on generic string data.
- Although, in theory, you shouldn't count on the explicit length field appearing in the HLA string data format, most programs simply grab the length from the 4 bytes immediately preceding the string data, so

there's generally no need to save the length. Careful HLA programmers will actually call the str.len() function in the HLA standard library and simply save this value in a local variable for future use. However, accessing the length directly is probably safe.

- Avoid copying string data from one string variable to another. Doing so is one of the more expensive operations in programs using HLA strings.

10.1.5 Descriptor-Based Strings

The string formats we've considered up to this point have kept the attribute information (that is, the lengths and terminating bytes) for a string in memory along with the character data. A slightly more flexible scheme is to maintain such information in a record structure, known as a *descriptor*, that also contains a pointer to the character data. Consider the following Pascal/Delphi data structure (see Figure 10-5):

```
type
    dString = record
        curLength  :integer;
        strData    :^char;
    end;
```

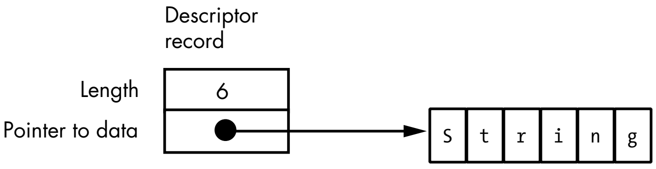


Figure 10-5: String descriptors

Note that this data structure does not hold the actual character data. Instead, the strData pointer contains the address of the first character of the string. The curLength field specifies the current length of the string. You could add any other fields you like to this record, such as a maximum length field, although a maximum length isn't usually necessary because most string formats employing a descriptor are dynamic (as the next section will discuss).

An interesting attribute of a descriptor-based string system is that the actual character data associated with a string could be part of a larger string. Because no length or terminating bytes are in the actual character data, it's possible to have the character data for two strings overlap (see Figure 10-6).

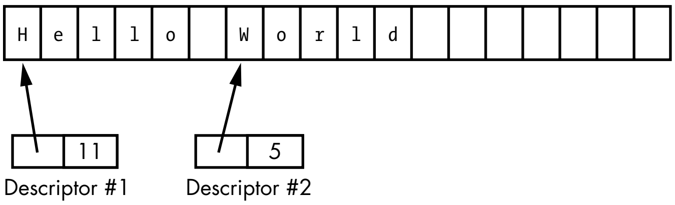


Figure 10-6: Overlapping strings using descriptors

This example shows two strings—`"Hello World"` and `"World"`—that overlap. This can save memory and make certain functions, like `substring()`, very efficient. Of course, when strings overlap as these do, you can't modify the string data because that could wipe out part of some other string.

The suggestions given for other string formats don't apply as strongly to descriptor-based strings. Certainly, if standard libraries are available, you should call those functions because they're probably more efficient than the ones you would write yourself. There is no need to save the length, because extracting the length field from the string's descriptor is usually a minor task. Also, many descriptor-based string systems use *copy on write* (see *WGC1* and the section "Dynamic Strings" on page 317) to reduce string copy overhead. In a string descriptor system, you should avoid making changes to a string, because the copy-on-write semantics generally require the system to make a complete copy of the string whenever you change a single character (something that isn't necessary with other string formats).

10.2 Static, Pseudo-Dynamic, and Dynamic Strings

Having covered the various string data formats, it's time to consider where to store string data in memory. Strings can be classified according to when and where the system allocates storage for them. There are three categories: static strings, pseudo-dynamic strings, and dynamic strings.

10.2.1 Static Strings

Pure *static strings* are those whose maximum size a programmer chooses when writing the program. Pascal strings and Delphi *short strings* fall into this category. Arrays of characters that you use to hold zero-terminated strings in C/C++ also fall into this category, as do fixed-length arrays of characters. Consider the following declaration in Pascal:

```
(* Pascal static string example *)

var
    //Max length will always be 255 characters.

    pascalString :string[255];
```

And here's an example in C/C++:

```
// C/C++ static string example:

//Max length will always be 255 characters (plus 0 byte).

char cString[256];
```

While the program is running, there's no way to increase the maximum sizes of these static strings. Nor is there any way to reduce the storage they will use; these string objects will consume 256 bytes at runtime, period.

One advantage to pure static strings is that the compiler can determine their maximum length at compile time and implicitly pass this information to a string function so it can test for bounds violations at runtime.

10.2.2 Pseudo-Dynamic Strings

A pseudo-dynamic string is one whose length the system sets at runtime by calling a memory management function like `malloc()` to allocate storage for it. However, once the system allocates storage for the string, the maximum length of the string is fixed. HLA strings generally fall into this category.[6] An HLA programmer typically calls the `stralloc()` function to allocate storage for a string variable, after which that particular string object has a fixed length that cannot change.[7]

10.2.3 Dynamic Strings

Dynamic string systems, which typically use a descriptor-based format, automatically allocate sufficient storage for a string object whenever you create a new string or otherwise do something that affects an existing string. Operations like string assignment and substring extraction are relatively trivial in dynamic string systems—generally they copy only the string descriptor data, so these operations are fast. However, as noted in the section "Descriptor-Based Strings" on page 315, when using strings this way, you cannot store data back into a string object, because it could modify data that is part of other string objects in the system.

The solution to this problem is to use the copy-on-write technique. Whenever a string function needs to change characters in a dynamic string, the function first makes a copy of the string and then makes the necessary modifications to that copy. Research suggests that copy-on-write semantics can improve the performance of many typical applications, because operations like string assignment and substring extraction (which is just a partial string assignment) are far more common than the modification of character data within strings. The only drawback to this approach is that after several modifications to string data in memory, there may be sections of the string heap area that contain character data that's no longer in use. To avoid a memory leak, dynamic string systems employing copy on write usually provide garbage collection code, which scans the string heap area looking for stale character data in order to recover that memory for other purposes. Unfortunately, depending on the algorithms in use, garbage collection can be quite slow.

NOTE *See Chapter 9 for more information on memory leaks and garbage collection.*

6. Though, being assembly language, it's possible to create static strings and pure dynamic strings in HLA as well.

7. Actually, you could call `strrealloc()` to change the size of an HLA string, but dynamic string systems generally do this automatically. Existing HLA string functions will not do this for you if they detect a string overflow.

10.3 Reference Counting for Strings

Consider the case where you have two string descriptors (or pointers) pointing at the same string data in memory. Clearly, you can't deallocate the storage associated with one pointer while the program is still using the other pointer to access the same data. One common solution is to make the programmer responsible for keeping track of such details. Unfortunately, as applications become more complex, this approach often leads to dangling pointers, memory leaks, and other pointer-related problems in the software. A better solution is to allow the programmer to deallocate the storage for the character data in the string and to have the actual deallocation process hold off until the programmer releases the last pointer referencing that data. To accomplish this, a string system can use *reference counters*, which track the pointers and their associated data.

A reference counter is an integer that counts the number of pointers that reference a string's character data in memory. Every time you assign the address of the string to some pointer, you increment the reference counter by 1. Likewise, whenever you want to deallocate the storage associated with the character data for the string, you decrement the reference counter. Deallocation of the storage for the actual character data doesn't happen until the reference counter decrements to 0.

Reference counting works great when the language handles the details of string assignment automatically for you. If you try to implement reference counting manually, you must be sure to always increment the reference counter when you assign a string pointer to some other pointer variable. The best way to do this is to never assign pointers directly, but rather handle all string assignments via some function (or macro) call that updates the reference counters in addition to copying the pointer data. If your code fails to update the reference counter properly, you'll wind up with dangling pointers or memory leaks.

10.4 Delphi Strings

Although Delphi provides a "short string" format that is compatible with the length-prefixed strings in earlier versions of Delphi and Turbo Pascal, later versions of Delphi (v4.0 and later) use dynamic strings for their native string format. While this string format is unpublished (and, therefore, subject to change), indications are that Delphi's string format is very similar to HLA's. Delphi uses a zero-terminated sequence of characters with a leading string length and a reference counter (rather than a maximum length as HLA uses). Figure 10-7 shows the layout of a Delphi string in memory.

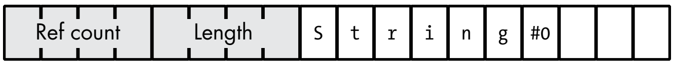


Figure 10-7: Delphi string data format

As with HLA, Delphi string variables are pointers holding the address of the first character of the actual string data. To access the length and reference counter fields, the Delphi string routines use a negative offset of –4 and –8 from the character data's base address. However, because this string format is not published, applications should never access the length or reference counter fields directly (for example, these fields could be 64-bit values one day). Delphi provides a length function that extracts the string length for you, and there's really no need for your applications to access the reference counter field because the Delphi string functions maintain it automatically.

10.5 Using Strings in a High-Level Language

Strings are a very common data type in high-level programming languages. Because applications often make extensive use of string data, many HLLs provide libraries with lots of complex string manipulation routines that hide considerable complexity from the programmer. Unfortunately, it's easy to forget the amount of work involved in a typical string operation when you execute a statement like this:

```
aLengthPrefixedString := 'Hello World';
```

In a typical Pascal implementation, this assignment statement calls a function that winds up copying each character from the string literal to the storage reserved for the `aLengthPrefixedString` variable. That is, this statement roughly expands to the following:

```
(* Copy the characters in the string *)

    for i:= 1 to length( HelloWorldLiteralString ) do begin

        aLengthPrefixedString[ i ] :=
            HelloWorldLiteralString[ i ];

    end;

    (* Set the string's length *)

    aLengthPrefixedString[0] :=
        char( length( HelloWorldLiteralString ));
```

This code doesn't even include the overhead of the procedure call, return, and parameter passing. As noted throughout the chapter, copying string data is one of the more expensive operations programs commonly do. This is why many HLLs have switched to dynamic strings and copy-on-write semantics—string assignments are far more efficient when you copy only a pointer rather than all of the character data. This is not to suggest that copy on write is always better, but for many string operations—such as assignment, substring, and other operations that do not change the string's character data—it can be very efficient.

Although few programming languages give you the option of choosing which string format you want to use, many do let you create pointers to strings, so you can manually support copy on write. If you're willing to write your own string-handling functions, you can create some very efficient programs by avoiding the use of your language's built-in string-handling capabilities. For example, the substring operation in C is usually handled by the `strncpy()` function and is often implemented like so:[8]

```
char *
strncpy( char* dest, char *src, int max )
{
    char *result = dest;
    while( max > 0 )
    {
        *dest = *src++;
        if( *dest++ == '\0') break;
        --max;
    }
    return result;
}
```

A typical "substring" operation might use `strncpy()` as follows:

```
strncpy( substring, fullString + start, length );
substring[ length ] = '\0';
```

where *`substring`* is the destination string object, *`fullString`* is the source string, *`start`* is the starting index of the substring to copy, and *`length`* is the length of the substring to copy.

If you create a descriptor-based string format in C using a `struct`, similar to the HLA record in "Descriptor-Based Strings" on page 315, you could do a substring operation with the following two statements in C:

```
// Assumption: ".strData" field is char*

    substring.strData = fullString.strData + start;
    substring.curLength = length;
```

This code executes much faster than the `strncpy()` version.

Sometimes, a particular programming language won't provide access to the underlying string data representation it supports, and you'll have to live with the performance loss, switch languages, or write your own string-handling code in assembly language. Generally, though, there are alternatives to copying string data in your applications, such as using a string descriptor as in the example just given.

8. Most real-world `strncpy()` routines are often more efficient than this example. In fact, many are written in assembly language, but we'll ignore that here.

10.6 Unicode Character Data in Strings

Up to this point, we've assumed that each character in a string consumes exactly 1 byte of storage. We've also assumed the use of the 7-bit ASCII character set when discussing the character data appearing in a string. Traditionally, this has been the way programming languages have represented a string's character data. Today, however, the ASCII character set is too limited for worldwide use, and several new character sets have risen in popularity, including the Unicode variants: UTF-8, UTF-16, UTF-32, and UTF-7. Because these character formats can have a big impact on the efficiency of string functions that operate upon them, we'll spend some time covering them.

10.6.1 The Unicode Character Set

A few decades back, engineers at Aldus, NeXT, Sun, Apple Computer, IBM, Microsoft, the Research Library Group, and Xerox realized that their new computer systems with bitmaps and user-selectable fonts could display far more than 256 different characters at one time. At the time, *double-byte character sets (DBCSs)* were the most common solution. DBCSs had a couple of issues, however. First, as they were typically variable-length encodings, DBCSs required special library code; common character/string algorithms that depended upon fixed-length character encodings would not work properly with them. Second, there was no consistent standard—different DBCSs used the same encoding for different characters. So, wanting to avoid these compatibility problems, the engineers sought a different route.

The solution they came up with was the Unicode character set. The engineers who originally developed Unicode chose a 2-byte character size. Like DBCSs, this approach still required special library code (existing single-byte string functions would not always work with 2-byte characters), but other than changing the size of a character, most existing string algorithms would still work with 2-byte characters. The Unicode definition included all of the (known/living) character sets at the time, giving each character a unique encoding, to avoid the consistency problems that plagued differing DBCSs.

The original Unicode standard used a 16-bit word to represent each character. Therefore, Unicode supported up to 65,536 different character codes—a huge advance over the 256 possible codes that are representable with an 8-bit byte. Furthermore, Unicode is upward compatible from ASCII. If the HO 9 bits[9] of a Unicode character's binary representation contain `0`, then the LO 7 bits use the standard ASCII code. If the HO 9 bits contain some nonzero value, then the 16 bits form an extended character code (extended from ASCII, that is). If you're wondering why so many different character codes are necessary, note that, at the time, certain Asian character sets contained 4,096 characters. The Unicode character set even provided a set of codes you could use to create an application-defined character set.

9. ASCII is a 7-bit code. If the HO 9 bits of a 16-bit Unicode value are all `0`, the remaining 7 bits are an ASCII encoding for a character.

Approximately half of the 65,536 possible character codes have been defined, and the remaining character encodings are reserved for future expansion.

Today, Unicode is a universal character set, long replacing ASCII and older DBCSs. All modern operating systems (including macOS, Windows, Linux, iOS, Android, and Unix), web browsers, and most modern applications provide Unicode support. Unicode Consortium, a nonprofit corporation, maintains the Unicode standard. By maintaining the standard, Unicode, Inc. (*https://home.unicode.org*), helps guarantee that a character you write on one system will display as you expect on a different system or application.

10.6.2 Unicode Code Points

Alas, as well thought-out as the original Unicode standard was, it couldn't have anticipated the explosion in characters that would occur. Emojis, astrological symbols, arrows, pointers, and a wide variety of symbols introduced for the internet, mobile devices, and web browsers have greatly expanded the Unicode symbol repertoire (along with a desire to support historic, obsolete, and rare scripts). In 1996, systems engineers discovered that 65,536 symbols were insufficient. Rather than require 3 or 4 bytes for each Unicode character, those in charge of the Unicode definition gave up on trying to create a fixed-size representation of characters and allowed for opaque (and multiple) encodings of Unicode characters. Today, Unicode defines 1,112,064 code points, far exceeding the 2-byte capacity originally set aside for Unicode characters.

A Unicode *code point* is simply an integer value that Unicode associates with a particular character symbol; you can think of it as the Unicode equivalent of the ASCII code for a character. The convention for Unicode code points is to specify the value in hexadecimal with a `U+` prefix; for example, `U+0041` is the Unicode code point for the letter *A*.

NOTE *See* https://en.wikipedia.org/wiki/Unicode#General_Category_property *for more details on code points.*

10.6.3 Unicode Code Planes

Because of its history, blocks of 65,536 characters are special in Unicode—they are known as a *multilingual plane*. The first multilingual plane, `U+000000` to `U+00FFFF`, roughly corresponds to the original 16-bit Unicode definition; the Unicode standard calls this the *Basic Multilingual Plane (BMP)*. Planes 1 (`U+010000` to `U+01FFFF`), 2 (`U+020000` to `U+02FFFF`), and 14 (`U+0E0000` to `U+0EFFFF`) are supplementary planes. Unicode reserves planes 3 through 13 for future expansion and planes 15 and 16 for user-defined character sets.

The Unicode standard defines code points in the range `U+000000` to `U+10FFFF`. Note that `0x10ffff` is 1,114,111, which is where most of the 1,112,064 characters in the Unicode character set come from; the remaining 2,048 code points are reserved for use as *surrogates*, which are Unicode extensions. *Unicode scalar* is another term you might hear; this is a value from the set of all Unicode code points *except* the 2,048 surrogate code points. The HO two hexadecimal digits of the six-digit code point value specify the multilingual plane. Why 17 planes? The reason, as you'll see in a moment, is that Unicode uses special multiword entries to encode code points beyond `U+FFFF`. Each of the two possible extensions encodes 10 bits, for a total of 20 bits; 20 bits gives you 16 multilingual planes, which, plus the original BMP, produces 17 multilingual planes. This is also why code points fall in the range `U+000000` to `U+10FFFF`: it takes 21 bits to encode the 16 multilingual planes plus the BMP.

10.6.4 Surrogate Code Points

As noted earlier, Unicode began life as a 16-bit (2-byte) character set encoding. When it became apparent that 16 bits were insufficient to handle all the possible characters that existed at the time, an expansion was necessary. As of Unicode v2.0, the Unicode, Inc., organization extended the definition of Unicode to include multiword characters. Now Unicode uses surrogate code points (`U+D800` through `U+DFFF`) to encode values larger than `U+FFFF`. Figure 10-8 shows the encoding.

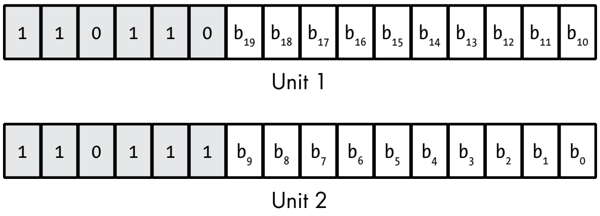

1 1 0 1 1 0 b19 b18 b17 b16 b15 b14 b13 b12 b11 b10
Unit 1
1 1 0 1 1 1 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
Unit 2

Figure 10-8: Surrogate code point encoding for Unicode planes 1 through 16

Note that the two words (unit 1/high surrogate and unit 2/low surrogate) always appear together. The unit 1 value (with HO bits `%110110`) specifies the upper 10 bits ($b_{10}..b_{19}$) of the Unicode scalar, and the unit 2 value (with HO bits `%110111`) specifies the lower 10 bits ($b_0..b_9$) of the Unicode scalar. Therefore, the value of bits $b_{16}..b_{19}$ plus 1 specifies Unicode plane 1 through 16. Bits $b_0..b_{15}$ specify the Unicode scalar value within the plane.

Note that surrogate codes only appear in the BMP. None of the other multilingual planes contain surrogate codes. Bits $b_0..b_{19}$ extracted from the unit 1 and 2 values always specify a Unicode scalar value (even if the values fall in the range `U+D800` through `U+DFFF`).

10.6.5 Glyphs, Characters, and Grapheme Clusters

Each Unicode code point has a unique name. For example, `U+0045` has the name "LATIN CAPITAL LETTER A." Note that the symbol *A* is *not* the name of the character. *A* is a *glyph*—a series of strokes (one horizontal and two slanted strokes) that a device draws in order to represent the character.

There are many different glyphs for the single Unicode character "LATIN CAPITAL LETTER A." For example, a Times Roman A and a Times Roman Italic *A* have different glyphs, but Unicode doesn't differentiate between them (or between the *A* character in any two different fonts).

The character "LATIN CAPITAL LETTER A" remains `U+0045` regardless of the font or style you use to draw it.

As an interesting side note, if you have access to the Swift programming language, you can print the name of any Unicode character using the following code:

```
import Foundation
let charToPrintName  :String = "A"      // Print name of this character

let unicodeName =
    String(charToPrintName).applyingTransform(
        StringTransform(rawValue: "Any-Name"),
        reverse: false
    )! // Forced unwrapping is legit here because it always succeeds.
print( unicodeName )

Output from program:
\N{LATIN CAPITAL LETTER A}
```

So, what exactly is a character in Unicode? Unicode scalars are Unicode characters, but there's a difference between what you'd normally call a character and the definition of a Unicode character (scalar). For example, is *é* one character or two? Consider the following Swift code:

```
import Foundation
let eAccent  :String = "e\u{301}"
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

`"\u{301}"` is the Swift syntax for specifying a Unicode scalar value within a string; in this particular case `301` is the hexadecimal code for the *combining acute accent* character.

The first `print` statement:

```
print( eAccent )
```

prints the character (producing `é` on the output, as we expect).

The second `print` statement prints the number of characters Swift determines are present in the string:

```
print( "eAccent.count=\(eAccent.count)" )
```

This prints `1` to the standard output.

The third `print` statement prints the number of elements (UTF-16 elements[10]) in the string:

```
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

10. See "Unicode Encodings" on page 327 for a discussion of UTF-16 encoding.

This prints `2` on the standard output, because the string holds 2 words of UTF-16 data.

So, again, is this one character or two? Internally (assuming UTF-16 encoding), the computer sets aside 4 bytes of memory for this single character (two 16-bit Unicode scalar values).[11] On the screen, however, the output takes only one character position and looks like a single character to the user. When this character appears within a text editor and the cursor is immediately to the right of the character, the user expects that pressing the backspace key will delete it. From the user's perspective, then, this is a single character (as Swift reports when you print the `count` attribute of the string).

In Unicode, however, a character is largely equivalent to a code point. This is not what people normally think of as a character. In Unicode terminology, a *grapheme cluster* is what people normally call a character—it's a sequence of one or more Unicode code points that combine to form a single language element (that is, a single character). So, when we talk about characters with respect to symbols that an application displays to an end user, we're really talking about grapheme clusters.

Grapheme clusters can make life miserable for software developers. Consider the following Swift code (a modification of the earlier example):

```
import Foundation
let eAccent  :String = "e\u{301}\u{301}"
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

This code produces the same `é` and `1` outputs from the first two `print` statements. The following produces `é`:

```
print( eAccent )
```

and this `print` statement produces `1`.

```
print( "eAccent.count=\(eAccent.count)" )
```

However, the third `print` statement:

```
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

displays `3` rather than `2` (as in the original example).

There are definitely three Unicode scalar values in this string (`U+0065`, `U+0301`, and `U+0301`). When printing, the operating system combines the `e` and the two acute accent combining characters to form the single character `é` and then outputs the character to the standard output device. Swift is smart enough to know that this combination creates a single output symbol on the display, so printing the result of the `count` attribute continues to

11. Swift 5 switches the preferred encoding of strings from UTF-16 to UTF-8; see *https://swift.org/blog/utf8-string/*.

output 1. However, there are (undeniably) three Unicode code points in this string, so printing utf16.count produces 3 on output.

10.6.6 Unicode Normals and Canonical Equivalence

The Unicode character *é* actually existed on personal computers long before Unicode came along. It's part of the original IBM PC character set and also part of the Latin-1 character set (used, for example, on old DEC terminals). As it turns out, Unicode uses the Latin-1 character set for the code points in the range U+00A0 to U+00FF, and U+00E9 just happens to correspond to the *é* character. Therefore, we can modify the earlier program as follows:

```
import Foundation
let eAccent  :String = "\u{E9}"
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

And here are the outputs from this program:

```
é
1
1
```

Ouch! Three different strings all producing é but containing a different number of code points. Imagine how this complicates programming strings containing Unicode characters. For example, if you have the following three strings (Swift syntax) and you try to compare them, what will the result be?

```
let eAccent1 :String = "\u{E9}"
let eAccent2 :String = "e\u{301}"
let eAccent3 :String = "e\u{301}\u{301}"
```

To the user, all three strings look the same on the screen. However, they clearly contain different values. If you compare them to see if they are equal, will the result be true or false?

Ultimately, that depends upon whose string libraries you're using. Most current string libraries would return false if you compared these strings for equality. Interestingly enough, Swift will claim that eAccent1 is equal to eAccent2, but it isn't smart enough to report that eAccent1 is equal to eAccent3 or that eAccent2 is equal to eAccent3—despite the fact that it displays the same symbol for all three strings. Many languages' string libraries simply report that all three strings are unequal.

The three Unicode/Swift strings "\u{E9}", "e\u{301}", and "e\u{301}\u{301}" all produce the same output on the display. Therefore, they are canonically equivalent according to the Unicode standard. Some string libraries won't report any of these strings as being equivalent. Some, like the one accompanying Swift, will handle small canonical equivalences (such as "\u{E9}" == "e\u{301}") but not arbitrary sequences that should be equivalent (probably a good balance of correctness versus efficiency; it can be computationally expensive to handle all the weird cases that won't normally happen, such as `"e\u{301}\u{301}"`).

Unicode defines *normal forms* for Unicode strings. One aspect of normal form is to replace canonically equivalent sequences with an equivalent sequence—for example, replace `"e\u{309}"` by `"\u{E9}"` or replace `"\u{E9}"` by `"e\u{309}"` (usually, the shorter form is preferable). Some Unicode sequences allow multiple combining characters. Often, the order of the combining characters is irrelevant to producing the desired grapheme cluster. However, it's easier to compare two such strings if the combining characters are in a specified order. Normalizing Unicode strings may also produce results whose combining characters always appear in a fixed order (thereby improving efficiency of string comparisons).

10.6.7 Unicode Encodings

As of Unicode v2.0, the standard supports a 21-bit character space capable of handling over a million characters (though most of the code points remain reserved for future use). Rather than use a fixed-size 3-byte (or worse, 4-byte) encoding to allow the larger character set, Unicode, Inc., allows different encodings—UTF-32, UTF-16, and UTF-8—each with its own advantages and disadvantages.[12]

UTF-32 uses 32-bit integers to hold Unicode scalars. The advantage to this scheme is that a 32-bit integer can represent every Unicode scalar value (which requires only 21 bits). Programs that require random access to characters in strings—without having to search for surrogate pairs—and other constant-time operations are (mostly) possible when using UTF-32. The obvious drawback to UTF-32 is that each Unicode scalar value requires 4 bytes of storage—twice that of the original Unicode definition and four times that of ASCII characters. It may seem that using two or four times as much storage (over ASCII and the original Unicode) is a small price to pay. After all, modern machines have several orders of magnitude more storage than they did when Unicode first appeared. However, that extra storage has a huge impact on performance, because those additional bytes quickly consume cache storage. Furthermore, modern string processing libraries often operate on character strings 8 bytes at a time (on 64-bit machines). With ASCII characters, that means a given string function can process up to eight characters concurrently; with UTF-32, that same string function can operate on only two characters concurrently. As a result, the UTF-32 version will run four times slower than the ASCII version. Ultimately, even Unicode scalar values are insufficient to represent all Unicode characters (that is, many Unicode characters require a sequence of Unicode scalars), so using UTF-32 doesn't solve the problem.

The second encoding format the Unicode supports is UTF-16. As the name suggests, UTF-16 uses 16-bit (unsigned) integers to represent Unicode values. To handle scalar values greater than `0xFFFF`, UTF-16 uses

12. *UTF* stands for *Unicode Transformational Format.*

the surrogate pair scheme to represent values in the range `0x010000` to `0x10FFFF` (see "Surrogate Code Points" on page 323). Because the vast majority of useful characters fit into 16 bits, most UTF-16 characters require only 2 bytes. For those rare cases where surrogates are necessary, UTF-16 requires 2 words (32 bits) to represent the character.

The last encoding, and unquestionably the most popular, is UTF-8. The UTF-8 encoding is forward compatible from the ASCII character set. In particular, all ASCII characters have a single-byte representation (their original ASCII code, where the HO bit of the byte containing the character contains a `0` bit). If the UTF-8 HO bit is `1`, then UTF-8 requires between 1 and 3 additional bytes to represent the Unicode code point. Table 10-1 provides the UTF-8 encoding schema.

Table 10-1: UTF Encoding

Bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+00	U+7F	0*xxxxxxx*			
2	11	U+80	U+7FF	110*xxxxx*	10*xxxxxx*		
3	16	U+800	U+FFFF	1110*xxxx*	10*xxxxxx*	10*xxxxxx*	
4	21	U+10000	U+10FFFF	11110*xxx*	10*xxxxxx*	10*xxxxxx*	10*xxxxxx*

The "*xxx*..." bits are the Unicode code point bits. For multibyte sequences, Byte 1 contains the HO bits, Byte 2 contains the next HO bits (LO bits compared to byte 1), and so on. For example, the 2-byte sequence (`%11011111`, `%10000001`) corresponds to the Unicode scalar `%0000_0111_1100_0001` (`U+07C1`).

UTF-8 encoding is probably the most common encoding in use. Most web pages use it. Most C standard library string functions will operate on UTF-8 text without modification (although some C standard library functions can produce malformed UTF-8 strings if the programmer isn't careful with them).

Different languages and operating systems use different encodings as their default. For example, macOS and Windows tend to use UTF-16 encoding, whereas most Unix systems use UTF-8. Some variants of Python use UTF-32 as their native character format. By and large, though, most programming languages use UTF-8 because they can continue to use older ASCII-based character processing libraries to process UTF-8 characters. Apple's Swift is one of the first programming languages that attempts to do Unicode right (though there is a huge performance hit for doing so).

10.6.8 Unicode Combining Characters

Although UTF-8 and UTF-16 encodings are much more compact than UTF-32, the CPU overhead and algorithmic complexities of dealing with multibyte (or multiword) characters sets complicates their use (introducing bugs and performance issues). Despite the issues of wasting memory (especially in the cache), why not simply define characters as 32-bit entities and be done with it? This seems like it would simplify string processing algorithms, improving performance and reducing the likelihood of defects in the code.

The problem with this theory is that you cannot represent all possible grapheme clusters with only 21 bits (or even 32 bits) of storage. Many grapheme clusters consist of several concatenated Unicode code points. Here's an example from Chris Eidhof and Ole Begemann's *Advanced Swift* (CreateSpace, 2017):

```
let chars: [Character] = [
    "\u{1ECD}\u{300}",
    "\u{F2}\u{323}",
    "\u{6F}\u{323}\u{300}",
    "\u{6F}\u{300}\u{323}"
]
```

Each of these Unicode grapheme clusters produces an identical character: an ó with a dot underneath the character (this is a character from the Yoruba character set). The character sequence (`U+1ECD`, `U+300`) is an `o` with a dot under it followed by a combining acute. The character sequence (`U+F2`, `U+323`) is an `ó` followed by a combining dot. The character sequence (`U+6F`, `U+323`, `U+300`) is an `o` followed by a combining dot, followed by a combining acute. Finally, the character sequence (`U+6F`, `U+300`, `U+323`) is an `o` followed by a combining acute, followed by a combining dot. All four strings produce the same output. Indeed, the Swift string comparisons treat all four strings as equal:

```
print("\u{1ECD} + \u{300} = \u{1ECD}\u{300}")
print("\u{F2} + \u{323} = \u{F2}\u{323}")
print("\u{6F} + \u{323} + \u{300} = \u{6F}\u{323}\u{300}")
print("\u{6F} + \u{300} + \u{323} = \u{6F}\u{300}\u{323}")
print( chars[0] == chars[1] ) // Outputs true
print( chars[0] == chars[2] ) // Outputs true
print( chars[0] == chars[3] ) // Outputs true
print( chars[1] == chars[2] ) // Outputs true
print( chars[1] == chars[3] ) // Outputs true
print( chars[2] == chars[3] ) // Outputs true
```

Note that there is not a single Unicode scalar value that will produce this character. You must combine at least two Unicode scalars (or as many as three) to produce this grapheme cluster on the output device. Even if you used UTF-32 encoding, it would still require two (32-bit) scalars to produce this particular output.

Emojis present another challenge that can't be solved using UTF-32. Consider the Unicode scalar `U+1F471`. This prints an emoji of a person with blond hair. If we add a skin color modifier to this, we obtain (`U+1F471`, `U+1F3FF`), which produces a person with a dark skin tone (and blond hair). In both cases we have a single character displaying on the screen. The first example uses a single Unicode scalar value, but the second example requires two. There is no way to encode this with a single UTF-32 value.

The bottom line is that certain Unicode grapheme clusters will require multiple scalars, no matter how many bits we assign to the scalar (it's possible to combine 30 or 40 scalars into a single grapheme cluster, for example). That means we're stuck dealing with multiword sequences to represent a single "character" regardless of how hard we try to avoid it. This is why UTF-32 has never really taken off. It doesn't solve the problem of random access into a string of Unicode characters. If you've got to deal with normalizing and combining Unicode scalars, it's more efficient to use UTF-8 or UTF-16 encodings.

Again, most languages and operating systems today support Unicode in one form or another (typically using UTF-8 or UTF-16 encoding). Despite the obvious problems with dealing with multibyte character sets, modern programs need to deal with Unicode strings rather than simple ASCII strings. Swift, which is almost "pure Unicode," doesn't even offer much in the way of standard ASCII character support.

10.7 Unicode String Functions and Performance

Unicode strings have one fundamental problem: because Unicode is a multibyte character set, the number of bytes in a character string is not equal to the number of characters (or, more importantly, the number of glyphs) in the string. Unfortunately, the only way to determine the length of a string is to scan all bytes in the string (from the beginning to the end) and count those characters. In this respect, the performance of a Unicode string length function will be proportional to the size of the string, just as it is for zero-terminated strings.

Worse still, the only way to compute the index of a character position in a string (that is, the offset in bytes from the beginning of the string) is to scan from the beginning of the string and count off the desired number of characters. Even zero-terminated (ASCII) strings don't suffer from this problem. In Unicode, functions like substring or insert/delete characters in a string can be very expensive.

The Swift standard library's string function performance suffers as a result of the language's Unicode purity. Swift programmers have to exercise caution when processing strings because operations that would normally be fast in C/C++ or other languages can be a source of performance problems in Swift's Unicode environment.

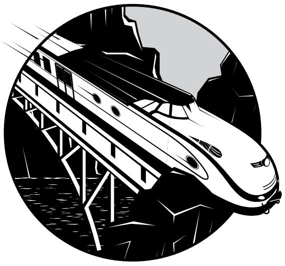
10.8 For More Information

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

———. *Write Great Code, Volume 1: Understanding the Machine*. 2nd ed. San Francisco: No Starch Press, 2020.

11

RECORD, UNION, AND CLASS DATA TYPES



Records, unions, and classes are popular composite data types found in many modern programming languages. Incorrectly used, these data types can have a very negative impact on the performance of your software. Correctly used, however, they can actually improve the performance of your applications (compared with using alternative data structures). In this chapter we'll explore how you can make the most of these data types to maximize the efficiency of your programs. The topics this chapter covers include:

- Definitions for the record, union, and class data types
- Declaration syntax for records, unions, and classes in various languages
- Record variables and instantiation
- Compile-time initialization of records
- Memory representation of record, union, and class data
- Using records to improve runtime memory performance

- Dynamic record types
- Namespaces
- Variant data types and their implementation as a union
- Virtual method tables for classes and their implementation
- Inheritance and polymorphism in classes
- The performance cost associated with classes and objects

Before we get into the details of how you can implement these data types to produce code that is more efficient, easier to read, and easier to maintain, let's begin with some definitions.

11.1 Records

The Pascal *record* and the C/C++ *structure* are terms used to describe comparable composite data structures. Language design textbooks sometimes refer to these types as *Cartesian products* or *tuples*. The Pascal terminology is probably best, because it avoids confusion with the term *data structure*, so we'll use *record* here. Regardless of what you call them, records are a great tool for organizing your application data, and a good understanding of how languages implement them will help you write more efficient code.

An array is *homogeneous*, meaning that its elements are all of the same type. A record, on the other hand, is *heterogeneous*—its elements can have differing types. The purpose of a record is to let you encapsulate logically related values into a single object.

Arrays let you select a particular element via an integer index. With records, you must select an element, known as a *field*, by the field's name. Each of the field names within the record must be unique; that is, you can't use the same name more than once in the same record. However, all field names are local to their record, so you may reuse those names elsewhere in the program.[1]

11.1.1 Declaring Records in Various Languages

Before discussing how various languages implement record data types, we'll take a quick look at the declaration syntax for some of them, including Pascal, C/C++/C#, Swift, and HLA.

11.1.1.1 Record Declarations in Pascal/Delphi

Here's a typical record declaration for a `student` data type in Pascal/Delphi:

```
type
    student =
        record
            Name:     string [64];
```

1. Technically, nested records may reuse field names internally, but those are different record structures, so the basic rule remains true.

```
        Major:    smallint;    // 2-byte integer in Delphi
        SSN:      string[11];
        Mid1:     smallint;
        Mid2:     smallint;
        Final:    smallint;
        Homework: smallint;
        Projects: smallint;
    end;
```

A record declaration consists of the keyword `record`, followed by a sequence of *field declarations*, and ending with the keyword `end`. The field declarations are syntactically identical to variable declarations in the Pascal language.

Many Pascal compilers allocate all of the fields in contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name,[2] the next 2 bytes hold the major code, the next 12 bytes the Social Security number, and so on.

11.1.1.2 Record Declarations in C/C++

Here's the same declaration in C/C++:

```
typedef
    struct
    {
        // Room for a 64-character zero-terminated string:

        char Name[65];

        // Typically a 2-byte integer in C/C++:

        short Major;

        // Room for an 11-character zero-terminated string:

        char SSN[12];

        short Mid1;
        short Mid2;
        short Final;
        short Homework;
        short Projects;

    } student;
```

Record (structure) declarations in C/C++ begin with the keyword `typedef` followed by the `struct` keyword, a set of *field declarations* enclosed by a pair of braces, and a structure name. As with Pascal, most C/C++ compilers assign memory offsets to the fields in the order of their declaration in the record.

2. Pascal strings usually require an extra byte, in addition to all the characters in the string, to encode the length.

11.1.1.3 Record Declarations in C#

C# structure declarations are very similar to C/C++:

```
struct student
 {
    // Room for a 64-character zero-terminated string:

    public char[] Name;

    // Typically a 2-byte integer in C/C++:

    public short Major;

    // Room for an 11-character zero-terminated string:

    public char[] SSN;

    public short Mid1;
    public short Mid2;
    public short Final;
    public short Homework;
    public short Projects;

 };
```

Record (structure) declarations in C# begin with the keyword `struct`, a structure name, and a set of *field declarations* enclosed by a pair of braces. As with Pascal, most C# compilers assign memory offsets to the fields in the order of their declaration in the record.

This example defines the `Name` and `SSN` fields as arrays of characters in order to match the other record declaration examples in this chapter. In an actual C# program you'd probably want to use the `string` data type rather than an array of characters for these fields. However, keep in mind that C# uses dynamically allocated arrays; thus, the memory layout for the C# structure will differ from those for C/C++, Pascal, and HLA.

11.1.1.4 Record Declarations in Java

Java doesn't support a pure record, but class declarations with only data members serve the same purpose (see the section "Class Declarations in C# and Java" on page 366).

11.1.1.5 Record Declarations in HLA

In HLA, you can create record types using the `record/endrecord` declaration. You would encode the record from the previous sections as follows:

```
type
    student:
        record
            sName:    char[65];
            Major:    int16;
            SSN:      char[12];
            Mid1:     int16;
            Mid2:     int16;
            Final:    int16;
            Homework: int16;
            Projects: int16;
        endrecord;
```

As you can see, the HLA declaration is very similar to the Pascal declaration. Note that, to stay consistent with the Pascal declaration, this example uses character arrays rather than strings for the `sName` and `SSN` (Social Security number) fields. In a typical HLA record declaration, you'd probably use a `string` type for at least the `sName` field (keeping in mind that a string variable is only a 4-byte pointer).

11.1.1.6 Record (Tuple) Declarations in Swift

Although Swift does not support the concept of a record, you can simulate one using a Swift *tuple*. Tuples are a useful construct for creating a composite/aggregate data type without the overhead of a class. (Note, however, that Swift does not store record/tuple elements in memory in the same manner as other programming languages.)

A Swift tuple is simply a list of values. Syntactically, a tuple takes the following form:

$(value_1, value_2, \ldots, value_n)$

The types of the values within the tuple don't have to be identical.

Swift typically uses tuples to return multiple values from functions. Consider the following short Swift code fragment:

```
func returns3Ints()->(Int, Int, Int )
{
    return(1, 2, 3)
}
var (r1, r2, r3) = returns3Ints();
print( r1, r2, r3 )
```

The `returns3Ints` function returns three values (`1`, `2`, and `3`). The statement

```
var (r1, r2, r3) = returns3Ints();
```

stores those three integer values into `r1`, `r2`, and `r3`, respectively.

You can also assign tuples to a single variable and access "fields" of the tuple using integer indexes as the field names:

```
let rTuple = ( "a", "b", "c" )
print( rTuple.0, rTuple.1, rTuple.2 ) // Prints "a b c"
```

Using field names like `.0` is inadvisable, as it results in hard-to-maintain code. You can create records out of tuples, but referring to the fields using integer indices is rarely suitable in real-world programs.

Fortunately, Swift allows you to assign labels to tuple fields and refer to those fields by the label name rather than an integer index, via the `typealias` keyword:

```
typealias record = ( field1:Int, field2:Int, field3:Float64 )

var r = record(1, 2, 3.0 )
print( r.field1, r.field2, r.field3 )  // prints "1 2 3.0"
```

Keep in mind that the storage of the tuple in memory might not map to the same layout as a record or structure in other languages. Like arrays in Swift, tuples are an opaque type, without a guaranteed definition of how Swift will store them in memory.

11.1.2 Instantiating a Record

Generally, a record declaration does not reserve storage for a record object; instead, it specifies a data type that you can use as a template when declaring record variables. *Instantiation* refers to this process of using a record template, or type, to create a record variable.

Consider the HLA type declaration for `student` from the previous section. This type declaration doesn't allocate any storage for a record variable; it simply provides the structure for the record object to use. To create an actual `student` variable, you must set aside some storage for the record variable, either at compile time or at runtime. In HLA, you can set aside storage for a `student` object at compile time by using variable declarations such as:

```
var
    automaticStudent :student;

static
    staticStudent :student;
```

The `var` declaration tells HLA to reserve sufficient storage for a `student` object in the current activation record when the program enters the current procedure. The `static` statement tells HLA to reserve sufficient storage for a `student` object in the static data section; this is done at compilation time.

You can also allocate storage for a record object dynamically using memory allocation functions. For example, in the C language you can use `malloc()`to allocate storage for a `student` object like so:

```
student *ptrToStudent;
        .
        .
        .
    ptrToStudent = malloc( sizeof( student ));
```

A record is simply a collection of (otherwise) unrelated variables. So why not just create separate variables? In C, for example, why not just write:

```
// Room for a 64-character zero-terminated string:

char someStudent_Name[65];

// Typically a 2-byte integer in C/C++:

short someStudent_Major;

// Room for an 11-character zero-terminated string:

char someStudent_SSN[12];

short someStudent_Mid1;
short someStudent_Mid2;
short someStudent_Final;
short someStudent_Homework;
short someStudent_Projects;
```

There are several reasons why this approach isn't ideal. On the software engineering side of things, there are maintenance issues to consider. For example, what happens if you create several sets of `student` variables and then decide you want to add a field? Now you've got to go back and edit every set of declarations you've created—not a pretty sight. With structure/record declarations, however, you only need to make one change to the type declaration, and all the variable declarations automatically get the new field. Also, consider what happens if you want to create an array of `student` objects.

Software engineering issues aside, collecting disparate fields into a record is a good idea for efficiency reasons. Many compilers allow you to treat a whole record as a single object for the purposes of assignment, parameter passing, and so on. In Pascal, for example, if you have two variables, `s1` and `s2`, of type `student`, you can assign all the values of one `student` object to the other with a single assignment statement like this:

```
s2 := s1;
```

Not only is this more convenient than assigning the individual fields, but the compiler can often generate better code by using a block move operation. Consider the following C++ code and the associated x86 assembly language output:

```
#include <stdio.h>

// A good-sized but otherwise arbitrary structure that
// demonstrates how a C++ compiler can handle structure
// assignments.

typedef struct
{
    int x;
    int y;
    char *z;
    int a[16];
}aStruct;


int main( int argc, char **argv )
{
    static aStruct s1;
    aStruct s2;
    int i;

    // Give s1 some nonzero values so
    // that the optimizer doesn't simply
    // substitute zeros everywhere fields
    // of s1 are referenced:

    s1.x = 5;
    s1.y = argc;
    s1.z = *argv;

    // Do a whole structure assignment
    // (legal in C++!)

    s2 = s1;

    // Make an arbitrary change to S2
    // so that the compiler's optimizer
    // won't eliminate the code to build
    // s2 and just use s1 because s1 and
    // s2 have the same values.

    s2.a[2] = 2;

    // The following loop exists, once again,
    // to thwart the optimizer from eliminating
    // s2 from the code:

    for( i=0; i<16; ++i)
    {
        printf( "%d\n", s2.a[i] );
    }

    // Now demonstrate a field-by-field assignment
    // so we can see the code the compiler generates:

    s1.y = s2.y;
    s1.x = s2.x;
    s1.z = s2.z;
    for( i=0; i<16; ++i )
    {
        s1.a[i] = s2.a[i];
    }
    for( i=0; i<16; ++i)
    {
        printf( "%d\n", s2.a[i] );
    }
    return 0;
}
```

Here's the relevant portion of the x86-64 assembly code that Microsoft's Visual C++ compiler produces (with the /O2 optimization option):

```
; Storage for the s1 array in the BSS segment:

_BSS    SEGMENT
?s1@?1??main@@9@9 DB 050H DUP (?)                       ; `main'::`2'::s1
_BSS    ENDS
;
s2$1$ = 32
s2$2$ = 48
s2$3$ = 64
s2$ = 80
__$ArrayPad$ = 160
argc$ = 192
argv$ = 200

; Note: on entry to main, rcx = argc, rdx = argv

main    PROC                                            ; COMDAT
; File c:\users\rhyde\test\t\t\t.cpp
; Line 20
$LN27:
        mov     r11, rsp
        mov     QWORD PTR [r11+24], rbx
        push    rdi
;
; Allocate storage for the local variables
; (including s2):

        sub     rsp, 176                                ; 000000b0H
        mov     rax, QWORD PTR __security_cookie
        xor     rax, rsp
        mov     QWORD PTR __$ArrayPad$[rsp], rax

        xor     ebx, ebx   ; ebx = 0
        mov     edi, ebx  ; edi = 0

     ; s1.z = *argv
        mov     rax, QWORD PTR [rdx] ;rax = *argv
        mov     QWORD PTR ?s1@?1??main@@9@9+8, rax

     ; s1.x = 5
        mov     DWORD PTR ?s1@?1??main@@9@9, 5

     ;s1.y = argc
        mov     DWORD PTR ?s1@?1??main@@9@9+4, ecx
```

```
;     s2 = s1;
;
;       xmm1=s1.a[0..1]
        movaps  xmm1, XMMWORD PTR ?s1@?1??main@@9@9+16
        movaps  XMMWORD PTR s2$[rsp+16], xmm1 ;s2.a[0..1] = xmm1
        movaps  xmm0, XMMWORD PTR ?s1@?1??main@@9@9
        movaps  XMMWORD PTR s2$[rsp], xmm0
        movaps  xmm0, XMMWORD PTR ?s1@?1??main@@9@9+32
        movaps  XMMWORD PTR s2$[rsp+32], xmm0
        movups  XMMWORD PTR s2$1$[rsp], xmm0
        movaps  xmm0, XMMWORD PTR ?s1@?1??main@@9@9+48
        movaps  XMMWORD PTR [r11-56], xmm0
        movups  XMMWORD PTR s2$2$[rsp], xmm0
        movaps  xmm0, XMMWORD PTR ?s1@?1??main@@9@9+64
        movaps  XMMWORD PTR [r11-40], xmm0
        movups  XMMWORD PTR s2$3$[rsp], xmm0

    ; s2.a[2] = 2

        mov     DWORD PTR s2$[rsp+24], 2
        npad    14

;    for (i = 0; i<16; ++i)
;    {

$LL4@main:
; Line 53
        mov     edx, DWORD PTR s2$[rsp+rdi*4+16]
        lea     rcx, OFFSET FLAT:??_C@_03PMGGPEJJ@?$CFd?6?$AA@
        call    printf
        inc     rdi
        cmp     rdi, 16
        jl      SHORT $LL4@main

.;     } //endfor

; Line 59 // s1.y = s2.y
        mov     eax, DWORD PTR s2$[rsp+4]
        mov     DWORD PTR ?s1@?1??main@@9@9+4, eax

       ;s1.x = s2.x
        mov     eax, DWORD PTR s2$[rsp]
        mov     DWORD PTR ?s1@?1??main@@9@9, eax

       ; s1.z = s2.z
        mov     rax, QWORD PTR s2$[rsp+8]
        mov     QWORD PTR ?s1@?1??main@@9@9+8, rax

;    for (i = 0; i<16; ++i)
;    {
;        printf("%d\n", s2.a[i]);
;    }
; Line 64
        movups  xmm1, XMMWORD PTR s2$1$[rsp]
        movaps  xmm0, XMMWORD PTR s2$[rsp+16]
        movups  XMMWORD PTR ?s1@?1??main@@9@9+32, xmm1
        movups  xmm1, XMMWORD PTR s2$3$[rsp]
        movups  XMMWORD PTR ?s1@?1??main@@9@9+16, xmm0
        movups  xmm0, XMMWORD PTR s2$2$[rsp]
        movups  XMMWORD PTR ?s1@?1??main@@9@9+64, xmm1
        movups  XMMWORD PTR ?s1@?1??main@@9@9+48, xmm0
        npad    7

$LL10@main:
; Line 68
        mov     edx, DWORD PTR s2$[rsp+rbx*4+16]
        lea     rcx, OFFSET FLAT:??_C@_03PMGGPEJJ@?$CFd?6?$AA@
        call    printf
        inc     rbx
        cmp     rbx, 16
        jl      SHORT $LL10@main

; Return 0
; Line 70
        xor     eax, eax
; Line 71
        mov     rcx, QWORD PTR __$ArrayPad$[rsp]
        xor     rcx, rsp
        call    __security_check_cookie
        mov     rbx, QWORD PTR [rsp+208]
        add     rsp, 176                              ; 000000b0H
        pop     rdi
        ret     0
main    ENDP
```

The important thing to note in this example is that the Visual C++ compiler emits a sequence of movaps and movups instructions whenever you assign whole structures. However, it may degenerate to a sequence of individual mov instructions for each of the fields when you do a field-by-field assignment of two structures. Likewise, if you had not encapsulated all the fields into a structure, then assigning the variables associated with your "structure" via a block copy operation wouldn't have been possible.

Combining fields together into a record has many advantages, including:

- It is much easier to maintain the record structure (that is, add, remove, rename, and change fields).
- Compilers can do additional type and semantic checking on records, thereby helping catch logic errors in your programs when you use a record improperly.
- Compilers can treat records as monolithic objects, generating more efficient code (for example, movsd and movaps instructions) than they can when working with individual field variables.

- Most compilers respect the order of declaration in a record, allocating successive fields to consecutive memory locations. This is important when interfacing data structures from two different languages. There is no guarantee for the organization of separate variables in memory in most languages.
- You can use records to improve cache memory performance and reduce virtual memory thrashing (as you'll soon see).
- Records can contain pointer fields that contain the address of other (like-typed) record objects. This isn't possible when you use bulk variables in memory.

You'll see some other advantages of records in the following sections.

11.1.3 Initializing Record Data at Compile Time

Some languages—for example, C/C++ and HLA—allow you to initialize record variables at compile time. For static objects, this spares your application the code and time needed to manually initialize each field of a record. For example, consider the following C code, which provides initializers for both static and automatic structure variables:

```
#include <stdlib.h>

// Arbitrary structure that consumes a nontrival
// amount of space:

typedef struct
{
    int x;
    int y;
    char *z;
    int a[4];
}initStruct;

// The following exists just to thwart
// the optimizer and make it think that
// all the fields of the structure are
// needed.

extern void thwartOpt( initStruct *i );

int main( int argc, char **argv )
{
    static initStruct staticStruct = {1,2,"Hello", {3,4,5,6}};
    initStruct autoStruct = {7,8,"World", {9,10,11,12}};

    thwartOpt( &staticStruct );
    thwartOpt( &autoStruct );
    return 0;

}
```

When compiled with Visual C++ using the `/O2` and `/Fa` command-line options, this example emits the following x86-64 machine code (edited manually to eliminate irrelevant output):

```
; Static structure declaration.
; Note how each of the fields is
; initialized with the initial values
; specified in the C source file:

; String used in static initStruct:

CONST   SEGMENT
??_C@_05COLMCDPH@Hello?$AA@ DB 'Hello', 00H      ; `string'
CONST   ENDS

_DATA   SEGMENT
; `main'::`2'::staticStruct
?staticStruct@?1??main@@9@9 DD 01H ;x field
        DD      02H ;y field
        DQ      FLAT:??_C@_05COLMCDPH@Hello?$AA@  ; z field
        DD      03H ;a[0] field
        DD      04H ;a[1] field
        DD      05H ;a[2] field
        DD      06H ;a[3] field
_DATA   ENDS

; String used to initialize autoStruct:

CONST   SEGMENT
??_C@_05MFLOHCHP@World?$AA@ DB 'World', 00H      ; `string'
CONST   ENDS
;
_TEXT   SEGMENT
autoStruct$ = 32
__$ArrayPad$ = 64
argc$ = 96
argv$ = 104
main    PROC                                     ; COMDAT
; File c:\users\rhyde\test\t\t\t.cpp
; Line 26
$LN9: ;Main program startup code:
        sub     rsp, 88                          ; 00000058H
        mov     rax, QWORD PTR __security_cookie
        xor     rax, rsp
        mov     QWORD PTR __$ArrayPad$[rsp], rax

; Line 28
;
; Initialize autoStruct:

        lea     rax, OFFSET FLAT:??_C@_05MFLOHCHP@World?$AA@
        mov     DWORD PTR autoStruct$[rsp], 7 ;autoStruct.x
        mov     QWORD PTR autoStruct$[rsp+8], rax
        mov     DWORD PTR autoStruct$[rsp+4], 8 ;autoStruct.y
        lea     rcx, QWORD PTR autoStruct$[rsp+16] ;autoStruct.a
        mov     eax, 9
        lea     edx, QWORD PTR [rax-5] ;edx = 4
$LL3@main:
; autoStruct.a[0] = 9, 10, 11, 12 (this is a loop)
        mov     DWORD PTR [rcx], eax
        inc     eax

; point RCX at next element of autoStruct.a
        lea     rcx, QWORD PTR [rcx+4]
        sub     rdx, 1
        jne     SHORT $LL3@main

; Line 30
; thwartOpt(&staticStruct );

        lea     rcx, OFFSET FLAT:?staticStruct@?1??main@@9@9
        call    thwartOpt

; Line 31
; thwartOpt( &autoStruct );

        lea     rcx, QWORD PTR autoStruct$[rsp]
        call    thwartOpt
; Line 32
; Return 0
        xor     eax, eax ;EAX = 0
; Line 34
        mov     rcx, QWORD PTR __$ArrayPad$[rsp]
        xor     rcx, rsp
        call    __security_check_cookie
        add     rsp, 88                                 ; 00000058H
        ret     0
main    ENDP
_TEXT   ENDS
        END
```

Look carefully at the machine code the compiler emits for the initialization of the autoStruct variable. In contrast to static initialization, the compiler cannot initialize memory at compile time because it doesn't know the addresses of the various fields of the automatic record that the system allocates at runtime. Unfortunately, this particular compiler generates a field-by-field sequence of assignments to initialize the fields of the structure. While this is relatively fast, it can consume quite a bit of memory, especially if you've got a large structure. If you want to reduce the size of the automatic structure variable initialization, one possibility is to create an initialized static structure and assign it to the automatic variable upon each entry into the function in which you've declared the automatic variable. Consider the following C++ and 80x86 assembly code:

```
#include <stdlib.h>
typedef struct
{
    int x;
    int y;
    char *z;
    int a[4];
}initStruct;

// The following exists just to thwart
// the optimizer and make it think that
// all the fields of the structure are
// needed.

extern void thwartOpt( initStruct *i );

int main( int argc, char **argv )
{
    static initStruct staticStruct = {1,2,"Hello", {3,4,5,6}};

    // initAuto is a "readonly" structure used to initialize
    // autoStruct upon entry into this function:

    static initStruct initAuto = {7,8,"World", {9,10,11,12}};

    // Allocate autoStruct on the stack and assign the initial
    // values kept in initAuto to this new structure:

    initStruct autoStruct = initAuto;

    thwartOpt( &staticStruct );
    thwartOpt( &autoStruct );
    return 0;

}
```

Here's the corresponding x86-64 assembly code that Visual C++ emits:

```
; Static initialized data for the staticStruct structure:

_DATA   SEGMENT

; Initialized data for staticStruct:

?staticStruct@?1??main@@9@9 DD 01H                     ;
`main'::`2'::staticStruct
        DD      02H
        DQ      FLAT:??_C@_05COLMCDPH@Hello?$AA@
        DD      03H
        DD      04H
        DD      05H
        DD      06H

; Initialization data to be copied to autoStruct:

?initAuto@?1??main@@9@9 DD 07H                         ;
`main'::`2'::initAuto
```

```
        DD      08H
        DQ      FLAT:??_C@_05MFLOHCHP@World?$AA@
        DD      09H
        DD      0aH
        DD      0bH
        DD      0cH
_DATA   ENDS

_TEXT   SEGMENT
autoStruct$ = 32
__$ArrayPad$ = 64
argc$ = 96
argv$ = 104
main    PROC                                ; COMDAT
; File c:\users\rhyde\test\t\t\t.cpp
; Line 23
$LN4:
; Main startup code:

        sub     rsp, 88                     ; 00000058H
        mov     rax, QWORD PTR __security_cookie
        xor     rax, rsp
        mov     QWORD PTR __$ArrayPad$[rsp], rax
; Line 34
; Initialize autoStruct by copying the data from the static
; initializer to the automatic variable:

        movups  xmm0, XMMWORD PTR ?initAuto@?1??main@@9@9
        movups  xmm1, XMMWORD PTR ?initAuto@?1??main@@9@9+16
        movups  XMMWORD PTR autoStruct$[rsp], xmm0
        movups  XMMWORD PTR autoStruct$[rsp+16], xmm1

; thwartOpt( &staticStruct );

        lea     rcx, OFFSET FLAT:?staticStruct@?1??main@@9@9
        call    thwartOpt  ; Arg is passed in RCX.

; thwartOpt( &autoStruct );

        lea     rcx, QWORD PTR autoStruct$[rsp]
        call    thwartOpt

; Return 0;
        xor     eax, eax
; Line 40
        mov     rcx, QWORD PTR __$ArrayPad$[rsp]
        xor     rcx, rsp
        call    __security_check_cookie
        add     rsp, 88                     ; 00000058H
        ret     0
main    ENDP
_TEXT   ENDS
        END
```

As you can see in this assembly code, it takes only a four-instruction sequence to copy the data from the statically initialized record into the automatically allocated record. This code is quite a bit shorter. Note, however, that it isn't necessarily faster. Copying data from one structure to another involves memory-to-memory moves, which can be quite slow if all the memory locations are not currently cached. Moving immediate constants directly to the individual fields is often faster, though it may take many instructions to accomplish this.

This example should remind you that if you attach an initializer to an automatic variable, the compiler will have to emit some code to handle that initialization at runtime. Unless your variables need to be reinitialized on each entry to your function, consider using static record objects instead.

11.1.4 Storing Records in Memory

The following Pascal example demonstrates a typical `student` record variable declaration:

```
var
    John: student;
```

Given the earlier declaration for the Pascal `student` data type, this allocates 81 bytes of storage laid out in memory as shown in Figure 11-1. If the label `John` corresponds to the *base address* of this record, then the `Name` field is at offset `John+0`, the `Major` field is at offset `John+65`, the `SSN` field is at offset `John+67`, and so on.

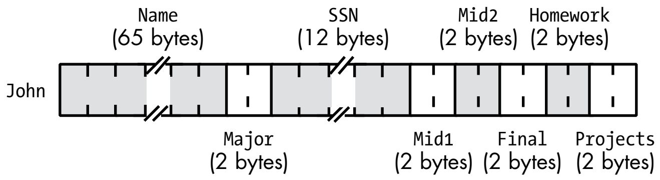


Figure 11-1: Student data structure storage in memory

Most programming languages let you refer to a record field by its name rather than by its numeric offset into the record (indeed, only a few low-end assemblers require that you reference fields by numeric offset; it's safe to say that such assemblers don't really support records). The typical syntax for a field access uses the *dot operator* to select a field from a record variable. Given the variable `John` from the previous example, here's how you could access various fields in this record:

```
John.Mid1 = 80;           // C/C++ example
John.Final := 93;         (* Pascal example *)
mov( 75, John.Projects ); // HLA example
```

Figure 11-1 suggests that all fields of a record appear in memory in the order of their declaration, and this is usually the case (although in theory, a compiler can freely place the fields anywhere in memory that it chooses). The first field usually appears at the lowest address in the record, the second field appears at the next-highest address, the third field follows the second field in memory, and so on.

Figure 11-1 also suggests that compilers pack the fields into adjacent memory locations with no gaps between the fields. While this is true for many languages, it's certainly not the most common memory organization for a record. For performance reasons, most compilers align the fields of a record on appropriate memory boundaries. The exact details vary by language, compiler implementation, and CPU, but a typical compiler places fields at an offset within the record's storage area that is "natural" for that particular field's data type. On the 80x86, for example, compilers that follow the Intel ABI (application binary interface) allocate single-byte objects at any offset within the record, words only at even offsets, and double word or larger objects on double word boundaries. Although not all 80x86 compilers support the Intel ABI, most do, which allows records to be shared among functions and procedures written in different languages on the 80x86. Other CPU manufacturers provide their own ABI for their processors, and programs that adhere to an ABI can share binary data at runtime with other programs that adhere to the same ABI.

In addition to aligning the fields of a record at reasonable offset boundaries, most compilers also ensure that the length of the entire record is a multiple of 2, 4, 8, or 16 bytes. As you've seen in previous chapters, they accomplish this by adding padding bytes at the end of the record to fill out the record's size. This ensures that the record's length is a multiple of the largest scalar (nonarray/nonrecord) object in the record.[3] For example, if a record has fields whose lengths are 1, 2, 4, and 8 bytes long, then an 80x86 compiler will generally pad the record's length so that it is a multiple of 8. This allows you to create an array of records and be assured that each record in the array starts at a reasonable address in memory.

Although some CPUs don't allow access to objects in memory at misaligned addresses, many compilers allow you to disable the automatic alignment of fields within a record. Generally, the compiler will have an option you can use to globally disable this feature. Many of these compilers also provide a `pragma`, `alignas`, or `packed` keyword that lets you turn off field alignment on a record-by-record basis. Disabling the automatic field alignment feature may allow you to save some memory by eliminating the padding bytes between the fields (and at the end of the record)—again, provided that field misalignment is acceptable on your CPU. The cost, of course, is that the program may run a little more slowly when it needs to access misaligned values in memory.

One reason to use a packed record is to gain manual control over the alignment of the record's fields. For example, suppose you have a couple of

3. Or a multiple of the CPU's maximum boundary size, if it is smaller than the size of the largest field in the record.

functions written in two different languages, and both of these functions need to access some data in a record. Further, suppose that the two compilers for these functions do not use the same field alignment algorithm. A record declaration like the following (in Pascal) may not be compatible with the way both functions access the record data:

```
type
    aRecord = record

        (* assume Pascal compiler supports a
        ** byte, word, and dword type
        *)

        bField : byte;
        wField : word;
        dField : dword;

    end; (* record *)
```

The problem here is that the first compiler could use the offsets 0, 2, and 4 for the `bField`, `wField`, and `dField` fields, respectively, while the second compiler might use offsets 0, 4, and 8.

Suppose, however, that the first compiler allows you to specify the `packed` keyword before the `record` keyword, causing the compiler to store each field immediately following the previous one. Although using the `packed` keyword doesn't make the records compatible with both functions, it does allow you to manually add padding fields to the record declaration, as follows:

```
type
    aRecord = packed record
        bField   :byte;  (* Offset 0 *)

        (* add padding to dword align wField *)

        padding0 :array[0..2] of byte;

        wField   :word; (* offset 4 *)

        (* add padding to dword align dField *)

        padding1 :word;

        dField   :dword;  (* offset 8 *)

    end; (* record *)
```

Manually adding padding can make maintaining your code a real chore. However, if incompatible compilers need to share data, it's a trick worth knowing. For the exact details on packed records, consult your language's reference manual.

11.1.5 Using Records to Improve Memory Performance

For someone who wants to write great code, records provide an important benefit: the ability to control variable placement in memory. This capability enables you to better control cache usage by those variables, which in turn can help you write code that executes much faster.

Consider, for a moment, the following C global/static variable declarations:

```
int i;
int j = 5;
int cnt = 0;
char a = 'a';
char b;
```

You might think that the compiler would allocate storage for these variables in consecutive memory locations. However, few (if any) languages guarantee this. C certainly doesn't and, in fact, C compilers like Microsoft's Visual C++ compiler don't allocate these variables in sequential memory locations. Consider the Visual C++ assembly language output for the preceding variable declarations:

```
PUBLIC  j
PUBLIC  cnt
PUBLIC  a
_DATA   SEGMENT
COMM    i:DWORD
_DATA   ENDS
_BSS    SEGMENT
cnt     DD      01H DUP (?)
_BSS    ENDS
_DATA   SEGMENT
COMM    b:BYTE
_DATA   ENDS
_DATA   SEGMENT
j       DD      05H
a       DB      061H
_DATA   ENDS
```

Even if you don't understand the purpose of all the directives here, it's clear that Visual C++ has rearranged all the variable declarations in memory. Therefore, you cannot count on adjacent declarations in your source file yielding adjacent storage cells in memory. Indeed, there is nothing to stop the compiler from allocating one or more variables in a machine register.

You might be wondering why you'd be concerned about the placement of variables in memory. After all, one of the main reasons for using named variables as an abstraction for memory is to avoid having to think about low-level memory allocation strategies. There are times, however, when being able to control variable placement in memory is important. For example, if you want to maximize program performance, you should try to place sets of variables that you access together in adjacent memory locations. This way, those variables will tend to sit in the same cache line, and you won't pay a heavy latency cost for accessing variables not currently held in the cache. Furthermore, by placing variables you use together adjacent to one another in memory, you'll use fewer cache lines and, therefore, have less thrashing.

Universally, programming languages that support the traditional notion of records maintain the fields of their records in adjacent memory locations; therefore, if you have some reason to keep different variables in adjacent memory locations (so that they share cache lines as much as possible), putting your variables into a record is a reasonable approach. However, the key word here is *traditional*—if your language uses a dynamic record type, you'll need a different approach.

11.1.6 Working with Dynamic Record Types and Databases

Some dynamic languages employ a dynamic type system, and object types can change at runtime. We'll explore dynamic types a little later in this chapter, but suffice it to say that if your language uses a dynamic type record structure, then all bets are off concerning the placement of fields in memory. Chances are pretty good that the fields will not be sitting in adjacent memory locations. Then again, if you're using a dynamic language, the fact that you're sacrificing a little performance because you're not getting maximal benefit from your cache will be the least of your worries.

A classic example of a dynamic record is the data you read from a database engine. The engine itself has no preconceived (that is, compile time) notion of what structure the database records will take. Instead, the database itself provides metadata that tells the database the record structure. The database engine reads this metadata from the database, uses it to organize the field data into a single record, and then returns this data to the database application. In a dynamic language, the actual field data is typically spread out across memory, and the database application references that data indirectly.

Of course, if you're using a dynamic language, you have much greater concerns about performance than the placement or organization of your record fields in memory. Dynamic languages, such as database engines, execute many instructions processing the metadata (or otherwise determining the type of their data operands), so losing a few cycles to cache thrashing here and there is unlikely to matter much. For more information about the overhead associated with a dynamic typing system, see "Variant Types" on page 356.

11.2 Discriminant Unions

A discriminant union (or just union) is very similar to a record. A *discriminant* is something that distinguishes or separates items in a quantity. In the case of a discriminant union, it means that different field names are used to distinguish the various ways that a given memory location's data type can be interpreted.

Like records, unions in typical languages that support them have fields that you access using dot notation. In fact, in many languages, about the only syntactical difference between records and unions is the use of the keyword `union` rather than `record` or `struct`. Semantically, however, there's a big difference between a record and a union. In a record, each field has its own offset from the base address of the record, and the fields do not overlap. In a union, however, all fields have the same offset, 0, and all the fields of the union overlap. As a result, the size of a record is the sum of the sizes of all the fields (plus, possibly, some padding bytes), whereas a union's size is the size of its largest field (plus, possibly, some padding bytes at the end).

Because the fields of a union overlap, changing the value of one field changes the values of all the other fields as well. This typically means that the use of a union's field is mutually exclusive—that is, you can use only one field at any given time. As a result, unions aren't as generally applicable as records, but they still have many uses. As you'll see later in this chapter, you can use unions to save memory by reusing memory for different values, to coerce data types, and to create variant data types. For the most part, though, programs use unions to share memory between different variable objects whose use never overlaps (that is, the variables' use is mutually exclusive).

For example, imagine that you have a 32-bit double word variable, and you find yourself constantly extracting out the LO or the HO 16-bit word. In most HLLs, this would require a 32-bit read and then an AND operation to mask out the unwanted word. If that wasn't enough, if you want the HO word, you have to then shift the result to the right 16 bits. With a union, you can overlay the 32-bit double word and a two-element 16-bit word array and access the words directly. You'll see how to do this in "Using Unions in Other Ways" on page 355.

11.2.1 Declaring Unions in Various Languages

The C/C++, Pascal, and HLA languages provide discriminant union type declarations. The Java language doesn't provide the equivalent of a union. Swift has a special version of the `Enum` declaration that provides variant record capabilities, but it does not store members of such declarations at the same address in memory. So, for the purposes of this discussion, we'll assume Swift doesn't provide union declarations.

11.2.1.1 Union Declarations in C/C++

Here's an example of a union declaration in C/C++:

```
typedef union
{
    unsigned int  i;
    float         r;
    unsigned char c[4];

} unionType;
```

Assuming the C/C++ compiler in use allocates 4 bytes for unsigned integers, the size of a `unionType` object will be 4 bytes (because all three fields are 4-byte objects).

11.2.1.2 Union Declarations in Pascal/Delphi

Pascal and Delphi use *case-variant records* to create a discriminant union. The syntax for a case-variant record is as follows:

```
type
    typeName =
        record

            <<nonvariant/union record fields go here>>

            case tag of
                const1:( field_declaration );
                const2:( field_declaration );
                    .
                    .
                    .
                constn:( field_declaration )

        end;
```

The *tag* item can be either a type identifier (such as `boolean`, `char`, or some user-defined type), or it can be a field declaration of the form *identifier*:*type*. If it takes the latter form, then *identifier* becomes another field of the record (and not a member of the variant section) and has the specified type. In addition, the Pascal compiler could generate code that raises an exception whenever the application attempts to access any of the variant fields except the one allowed by the value of the *tag* field. In practice, almost no Pascal compilers do this check. Still, keep in mind that the Pascal language standard suggests that compilers *should* do it, so some compilers might.

Here's an example of two different case-variant record declarations in Pascal:

```
type
    noTagRecord=
        record
            someField: integer;
            case boolean of
                true:( i:integer );
                false:( b:array[0..3] of char)
        end; (* record *)

    hasTagRecord=
        record
            case which:0..2 of
                0:( i:integer );
                1:( r:real );
                2:( c:array[0..3] of char )
        end; (* record *)
```

As you can see in the hasTagRecord union, a Pascal case-variant record does not require any normal record fields. This is true even if you do not have a tag field.

11.2.1.3 Union Declarations in HLA

HLA supports unions as well. Here's a typical union declaration in HLA:

```
type
    unionType:
        union
            i: int32;
            r: real32;
            c: char[4];
        endunion;
```

11.2.2 Storing Unions in Memory

Remember that the big difference between a union and a record is the fact that records allocate storage for each field at different offsets, whereas unions overlay each of the fields at the same offset in memory. For example, consider the following HLA record and union declarations:

```
type
    numericRec:
        record
            i: int32;
            u: uns32;
            r: real64;
        endrecord;

    numericUnion:
        union
            i: int32;
            u: uns32;
            r: real64;
        endunion;
```

If you declare a variable, say n, of type numericRec, you access the fields as n.i, n.u, and n.r, exactly as though you had declared the n variable to be type numericUnion. However, the size of a numericRec object is 16 bytes, because the record contains two double word fields and a quad word (real64) field. The size of a numericUnion variable, however, is 8 bytes. Figure 11-2 shows the memory arrangement of the i, u, and r fields in both the record and union.

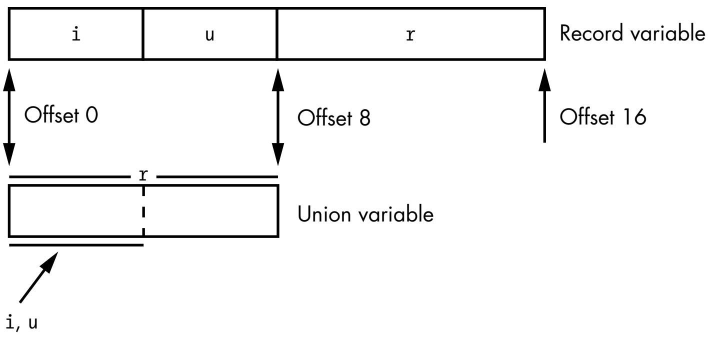


Figure 11-2: Layout of a union versus a record variable

11.2.3 Using Unions in Other Ways

In addition to conserving memory, programmers often use unions to create aliases in their code. An *alias* is a different name for the same memory object. Although aliases are often a source of confusion in a program and should be used sparingly, sometimes using them is convenient. For example, in some section of your program you might need to constantly use type coercion to refer to a particular object. To avoid this, you could use a union variable with each field representing one of the different types you want to use for the object. Consider the following HLA code fragment:

```
type
    CharOrUns:
        union
            c:char;
            u:uns32;
        endunion;

static
    v:CharOrUns;
```

With a declaration like this one, you can manipulate an `uns32` object by accessing `v.u`. If, at some point, you need to treat the LO byte of this `uns32` variable as a character, you can do so by simply accessing the `v.c` variable, as follows:

```
mov( eax, v.u );
stdout.put( "v, as a character, is '", v.c, "'" nl );
```

Another common practice is to use unions to disassemble a larger object into its constituent bytes. Consider the following C/C++ code fragment:

```
typedef union
{
    unsigned int u;
    unsigned char bytes[4];
} asBytes;

asBytes composite;
        .
        .
        .
    composite.u = 1234567890;
    printf
    (
        "HO byte of composite.u is %u, LO byte is %u\n",
        composite.bytes[3],
        composite.bytes[0]
    );
```

Although composing and decomposing data types this way is a useful trick to employ every now and then, keep in mind that this code isn't portable. The HO and LO bytes of a multibyte object appear at different addresses on big endian versus little endian machines. As a result, this code fragment works fine on little endian machines, but fails to display the correct bytes on big endian CPUs. Any time you use unions to decompose larger objects, you should be aware of this limitation. Still, this trick is usually much more efficient than using shift lefts, shift rights, and AND operations, so you'll see it used quite a bit.

11.3 Variant Types

A variant object has a *dynamic* type—that is, the object's type can vary at runtime. This spares the programmer from having to decide on a data type when designing the program and allows the end user to enter whatever data they like as the program operates. Programs written in a dynamically typed language are typically far more compact than languages written in a traditional statically typed language. This makes dynamically typed languages very popular for rapid prototyping, interpretive, and very high-level languages. A few mainstream languages (including Visual Basic and Delphi) also support variant types. In this section, we'll look at how compilers implement variant types and discuss the efficiency costs associated with them.

To implement a variant type, most languages use a union to reserve storage for all the different types the variant object supports. This means that a variant object will consume at least as much space as the largest primitive data type it supports. In addition to the storage required to keep its value, the variant object will also need storage to keep track of its current type. If the language allows variants to assume an array type, even more storage may be necessary to specify how many elements are in the array (or the bounds on each dimension, if the language allows multidimensional variant arrays). The bottom line is that a variant consumes a fair amount of memory, even if the actual data consumes only a single byte.

Perhaps the best way to illustrate how a variant data type works is to implement one manually. Consider the following Delphi case-variant record declaration:

```
type
    dataTypes =
           (
               vBoolean, paBoolean, vChar, paChar,
               vInteger, paInteger, vReal, paReal,
               vString, paString
           );

       varType =
           record
               elements : integer;
               case theType: dataTypes of
                   vBoolean:  ( b:boolean );
                   paBoolean: ( pb:array[0..0] of ^boolean );
                   vChar:     ( c:char );
                   paChar:    ( pc:array [0..0] of ^char );
                   vInteger:  ( i:integer );
                   paInteger: ( pi:array[0..0] of ^integer );
                   vReal:     ( r:real );
                   paReal:    ( pr:array[0..0] of ^real );
                   vString:   ( s:string[255] );
                   paString:  ( ps:array[0..0] of ^string[255] )
          end;
```

In this record, `elements` will contain the number of elements in the array if the object is a single-dimensional array (this particular data structure does not support multidimensional arrays). If, on the other hand, the object is a scalar variable, then the `elements` value will be irrelevant. The `theType` field specifies the current type of the object. If this field contains one of the enumerated constants `vBoolean`, `vChar`, `vInteger`, `vReal`, or `vString`, the object is a scalar variable; if it contains one of the constants `paBoolean`, `paChar`, `paInteger`, `paReal`, or `paString`, then the object is a single-dimensional array of the specified type.

The fields in the case-variant section of the Pascal record hold the variant's value if it is a scalar object, or they hold a pointer to an array of objects if the variant is an array object. Technically, Pascal requires that you specify the bounds of the array in its declaration. But fortunately, Delphi lets you turn off bounds checking (as well as allowing you to allocate memory for an array of arbitrary size), hence the dummy array bounds in this example.

Manipulating two variant objects that have the same type is easy. For example, suppose you want to add two variant values together. First, you'd determine the current type of both objects and whether the addition operation even makes sense for the data types.[4] Once you've decided that the

4. For example, you can't add two Boolean values together.

addition operation is reasonable, it's easy enough to use a `case` (or `switch`) statement based on the tag field of the two variant types:

```
// Handle the addition operation:

// Load variable theType with either left.theType
// or right.theType (which, presumably, contain
// the same value at this point).

case( theType ) of

    vBoolean: writeln( "Cannot add two Boolean values!" );
    vChar: writeln( "Cannot add two character values!" );
    vString: writeln( "Cannot add two string values!" );
    vInteger: intResult := left.vInteger + right.vInteger;
    vReal: realResult := left.vReal + right.vReal;
    paBoolean: writeln( "Cannot add two Boolean arrays!" );
    paChar: writeln( "Cannot add two character arrays!" );
    paInteger: writeln( "Cannot add two integer arrays!" );
    paReal: writeln( "Cannot add two real arrays!" );
    paString: writeln( "Cannot add two Boolean arrays!" );

end;
```

If the left and right operands are not the same type, then the operation is a bit more complex. Some mixed-type operations are legal. For example, adding an integer operand and a real operand together is reasonable (it produces a real type result in most languages). Other operations may be legal only if the values of the operands can be added. For example, it's reasonable to add a string and an integer together if the string happens to contain a string of digits that could be converted to an integer prior to the addition (likewise for string and real operands). What is needed here is a two-dimensional `case`/`switch` statement. Unfortunately, outside of assembly language, you won't find such a creature.[5] However, you can simulate one easily enough by nesting `case`/`switch` statements:

```
case( left.theType ) of

    vInteger:
        case( right.theType ) of
            vInteger:
                (* code to handle integer + integer operands *)
            vReal:
                (* code to handle integer + real operands *)
            vBoolean:
                (* code to handle integer + boolean operands *)
            vChar:
                (* code to handle integer + char operands *)
```

5. You won't really find it in assembly language, either, but you can easily write assembly code that does the same thing as a two-dimensional `case`/`switch` statement.

```
            vString:
                (* code to handle integer + string operands *)
            paInteger:
                (* code to handle integer + intArray operands *)
            paReal:
                (* code to handle integer + realArray operands *)
            paBoolean:
                (* code to handle integer + booleanArray operands *)
            paChar:
                (* code to handle integer + charArray operands *)
            paString:
                (* code to handle integer + stringArray operands *)
        end;

    vReal:
        case( right.theType ) of
            (* cases for each of the right operand types
                REAL + type *)
        end;

    Boolean:
        case( right.theType ) of
            (* cases for each of the right operand types:
                BOOLEAN + type *)
        end;

    vChar:
        case( right.theType ) of
            (* cases for each of the right operand types:
                CHAR + type *)
        end;

    vString:
        case( right.theType ) of
            (* cases for each of the right operand types:
                STRING + type *)
        end;

    paInteger:
        case( right.theType ) of
            (* cases for each of the right operand types:
                intArray + type *)
        end;

    paReal:
        case( right.theType ) of
            (* cases for each of the right operand types:
                realArray + type *)
        end;

    paBoolean:
        case( right.theType ) of
            (* cases for each of the right operand types:
                booleanArray + type *)
        end;
    paChar:
        case( right.theType ) of
            (* cases for each of the right operand types:
                charArray + type *)
        end;

    paString:
        case( right.theType ) of
            (* cases for each of the right operand types:
                stringArray + type *)
        end;

end;
```

Once you expand all the code alluded to in these comments, you'll have quite a few statements. And this is just for one operator! Obviously, it takes considerable work to implement all the basic arithmetic, string, character, and Boolean operations—and expanding this code inline whenever you need to add two variant values together is out of the question. Generally, you'd write a function like `vAdd()` that would accept two variant parameters and produce a variant result (or raise some sort of exception if the addition of the operands is illegal).

The takeaway here is not that the code to do variant addition is long—the real problem is performance. It's not at all unreasonable to expect a variant addition operation to require dozens, if not hundreds, of machine instructions to accomplish. By contrast, it takes only two or three machine instructions to add two integer or floating-point values together. Therefore, you can expect operations involving variant objects to run approximately one to two orders of magnitude slower than the standard operations. This, in fact, is one of the major reasons why "typeless" languages (usually very high-level languages) are so slow. When you truly need a variant type, the performance is often just as good (or even better) than the alternative code you'd have to write to get around using one. However, if you're using variant objects to hold values whose type you know when you first write the program, you'll pay a heavy performance penalty for not using typed objects.

In object-oriented languages such as C++, Java, Swift, and Delphi (Object Pascal), there's a better solution for variant calculations: inheritance and polymorphism. A big problem with the `union`/`switch` statement version is that it can be a major pain to extend the variant type by adding a new type to it. For example, suppose you want to add a new complex data type supporting complex numbers. You'd have to locate every function you've written (typically one for each operator) and add a new `case` to the `switch` statement. This can be a maintenance nightmare (especially if you don't have access to the original source code). However, by using objects, you can create a new class (such as `ComplexNumber`) that overrides the existing base class (perhaps `Numeric`) without having to modify any of the existing code (for other numeric types and operations). For more information on this method, see *Write Great Code, Volume 4: Designing Great Code.*

11.4 Namespaces

As your programs become larger, and particularly as these large programs use third-party software libraries to reduce development time, it becomes increasingly likely that name conflicts will arise in your source files. A name conflict occurs when you want to use a specific identifier at one point in your program, but that name is already in use elsewhere (for example, in a library you're using). At some point in a very large project, you may dream up a new name to resolve a naming conflict only to discover that the new name is also already in use. Software engineers call this *namespace pollution*. Like environmental pollution, the problem is easy to live with when it's small and localized. As your programs get larger, however, dealing with the fact that "all the good identifiers are already used up" is a real challenge.

At first blush, it might seem that this problem is exaggerated; after all, a programmer can always think of a different name. However, programmers who write great code often adhere to certain naming conventions so that their source code is consistent and easy to read (I'll come back to this subject in *Write Great Code, Volume 5: Great Coding*). Constantly devising new names, even if they aren't all that bad, tends to produce inconsistencies in the source code that make programs harder to read. It would be nice to choose whatever name you like for your identifiers and not have to worry about conflicts with other code or libraries. Enter namespaces.

A *namespace* is a mechanism by which you can associate a set of identifiers with a namespace identifier. In many respects, a namespace is like a record declaration. Indeed, you can use a `record` (or `struct`) declaration as a poor man's namespace in languages that don't support namespaces directly (with a few major restrictions). For example, consider the following Pascal variable declarations:

```
var
    myNameSpace:
        record
            i: integer;
            j: integer;
            name: string[64];
            date: string[10];
            grayCode: integer;
        end;

    yourNameSpace:
        record
            i: integer;
            j: integer;
            profits: real;
            weekday: integer;
        end;
```

As you can see, the `i` and `j` fields in these two records are distinct variables. There will never be a naming conflict because the program must qualify these two field names with the record variable name. That is, you refer to these variables using the following names:

```
myNameSpace.i, myNameSpace.j,
yourNameSpace.i, yourNameSpace.j
```

The record variable that prefixes the fields uniquely identifies each of these field names. This is clear to anyone who has ever written code that uses a record or structure. Therefore, in languages that don't support namespaces, you can use records (or classes) in their place.

There is one major problem with creating namespaces by using records or structures, though: many languages let you declare only variables within a record. Namespace declarations (like those available in C++ and HLA) specifically allow you to include other types of objects as well. In HLA, for example, a namespace declaration takes the following form:

```
namespace nsIdentifier;

    << constant, type, variable, procedure,
            and other declarations >>

end nsIdentifier;
```

A class declaration (if available in your chosen language) can overcome some of these problems. At the very least, most languages allow procedure or function declarations within a class, but many allow constant and type declarations as well.

Namespaces are a declaration section unto themselves. In particular, they do not have to go in a `var` or `static` (or any other) section. You can create constants, types, variables, static objects, procedures, and so on, all within a namespace.

To access namespace objects in HLA, you use the familiar dot notation that records, classes, and unions use. To access a name in a C++ namespace, you use the `::` operator.

As long as the namespace identifier is unique and all the fields within the namespace are unique to that namespace, you won't have any problems. By carefully partitioning a project into various namespaces, you can easily avoid most of the problems that occur because of namespace pollution.

Another interesting aspect to namespaces is that they are extensible. For example, consider the following declarations in C++:

```
namespace aNS
{
    int i;
    int j;
}

int i;  // Outside the namespace, so this is unique.
int j;  // ditto.
```

```
namespace aNS
{
    int k;
}
```

This example code is perfectly legal. The second declaration of aNS does not conflict with the first: it extends the aNS namespace to include identifier aNS::k as well as aNS::i and aNS::j. This feature is very handy when, for example, you want to extend a set of library routines and header files without modifying the original header files for that library (assuming the library names all appear within a namespace).

From an implementation point of view, there's really no difference between a namespace and a set of declarations appearing outside a namespace. The compiler typically deals with both types of declarations in a nearly identical fashion, with the only difference being that the program prefixes all objects located within the namespace with the namespace's identifier.

11.5 Classes and Objects

The *class* data type is the bedrock of modern object-oriented programming (OOP). In most OOP languages, the class is closely related to the record or structure. However, unlike records (which have a surprisingly uniform implementation across most languages), class implementations tend to vary. Nevertheless, many contemporary OOP languages achieve their results using similar approaches, so this section demonstrates a few concrete examples from C++, Java, Swift, HLA, and Delphi (Object Pascal). Users of other languages will find their languages work similarly.

11.5.1 Classes vs. Objects

Many programmers confuse the terms *class* and *object*. A class is a data type; it is a template for how the compiler organizes memory with respect to the class's fields. An object is an instantiation of a class—that is, an object is a variable of some class type that has memory allocated to hold the data associated with the class's fields. For a given class, there is only one class definition. You may, however, have several objects (variables) of that class type.

11.5.2 Simple Class Declarations in C++

Classes and structures are syntactically and semantically similar in C++. Indeed, there is only one syntactical difference between them: the use of the class keyword versus the struct keyword. Consider the following two valid type declarations in C++:

```
struct student
{
        // Room for a 64-character zero-terminated string:

        char Name[65];

        // Typically a 2-byte integer in C/C++:

        short Major;

        // Room for an 11-character zero-terminated string:

        char SSN[12];

        // Each of the following is typically a 2-byte integer

        short Mid1;
        short Mid2;
        short Final;
        short Homework;
        short Projects;
};


class myClass
{
public:

// Room for a 64-character zero-terminated string:

        char Name[65];

        // Typically a 2-byte integer in C/C++:

        short Major;

        // Room for an 11-character zero-terminated string:

        char SSN[12];

        // Each of the following is typically a 2-byte integer

        short Mid1;
        short Mid2;
        short Final;
        short Homework;
        short Projects;
};
```

Although these two data structures contain the same fields, and you would access those fields the same way, their memory implementation is slightly different. A typical memory layout for the structure appears in Figure 11-3, which can be compared with the memory layout for the class shown in Figure 11-4. (Figure 11-3 is the same as Figure 11-1, but appears here for easy comparison with Figure 11-4.)

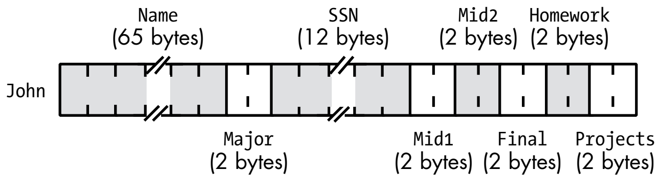


Figure 11-3: The student *structure storage in memory*

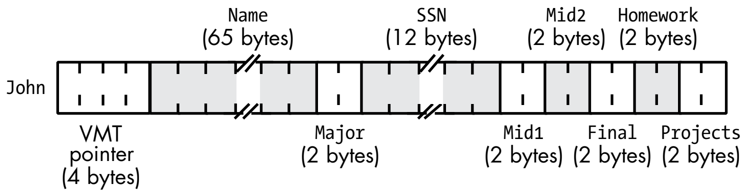


Figure 11-4: The student *class storage in memory*

The *VMT pointer* is a field that appears if the class contains any class member functions (aka *methods*). Some C++ compilers do not emit a VMT pointer field if there are no member functions, in which case the class and struct objects will have the same layout in memory.

NOTE VMT *stands for* virtual method table *and will be discussed further in the section "Virtual Method Tables" on page 367.*

Although a C++ class declaration could contain only data fields, classes generally contain member function definitions as well as data members. In the myClass example, you might have the following member functions:

```
class myClass
{
public:

// Room for a 64-character zero-terminated string:

        char Name[65];

        // Typically a 2-byte integer in C/C++:

        short Major;

        // Room for an 11-character zero-terminated string:

        char SSN[12];

        // Each of the following is typically a 2-byte integer

        short Mid1;
        short Mid2;
        short Final;
        short Homework;
        short Projects;

        // Member functions:

        double computeGrade( void );
        double testAverage( void );
};
```

The `computeGrade()` function might compute the total grade in the course (based on relative weights attached to the midterms, final, homework, and project scores). The `testAverage()` function might return the average of all the test scores.

11.5.3 Class Declarations in C# and Java

C# and Java classes look very similar to C/C++ class declarations. Here's a sample C# class declaration (which also works for Java):

```
class student
{
        // Room for a 64-character zero-terminated string:

        public char[] Name;

        // Typically a 2-byte integer in C/C++:

        public short Major;

        // Room for an 11-character zero terminated string:

        public char[] SSN;

        public short Mid1;
        public short Mid2;
        public short Final;
        public short Homework;
        public short Projects;

        public double computeGrade()
        {
            return Mid1 * 0.15 + Mid2 * 0.15 + Final *
                   0.2 + Homework * 0.25 + Projects * 0.25;
        }
        public double testAverage()
        {
            return (Mid1 + Mid2 + Final) / 3.0;
        }
    };
```

11.5.4 Class Declarations in Delphi (Object Pascal)

Delphi (Object Pascal) classes look very similar to Pascal records. Classes use the class keyword instead of record, and you can include function prototype declarations in the class.

```
type
  student =
    class
      Name:     string [64];
      Major:    smallint;    // 2-byte integer in Delphi
      SSN:      string[11];
      Mid1:     smallint;
      Mid2:     smallint;
      Final:    smallint;
      Homework: smallint;
      Projects: smallint;

      function computeGrade:real;
      function testAverage:real;
  end;
```

11.5.5 Class Declarations in HLA

HLA classes look very similar to HLA records. Classes use the class keyword instead of record, and you can include function (method) prototype declarations in the class.

```
type
    student:
        class
          var
            sName:    char[65];
            Major:    int16;
            SSN:      char[12];
            Mid1:     int16;
            Mid2:     int16;
            Final:    int16;
            Homework: int16;
            Projects: int16;

            method computeGrade;
            method testAverage;

        endclass;
```

11.5.6 Virtual Method Tables

As you saw in Figures 11-3 and 11-4, the difference between the class definition and the structure definition is that the former contains a VMT field. VMT, which stands for *virtual method table*, is an array of pointers to all the member functions, or *methods*, within an object's class. Virtual methods (*virtual member functions* in C++) are special class-related functions that you declare as fields in the class. In the current student example, the class doesn't actually have any virtual methods, so most C++ compilers would eliminate the VMT field, but some OOP languages will still allocate storage for the VMT pointer within the class.

Here's a little C++ class that actually has a virtual member function and, therefore, also has a VMT:

```
class myclass
{
    public:
        int a;
        int b;
        virtual int f( void );
};
```

When C++ calls a standard function, it directly calls that function. Virtual member functions are another story, as you can see in Figure 11-5.

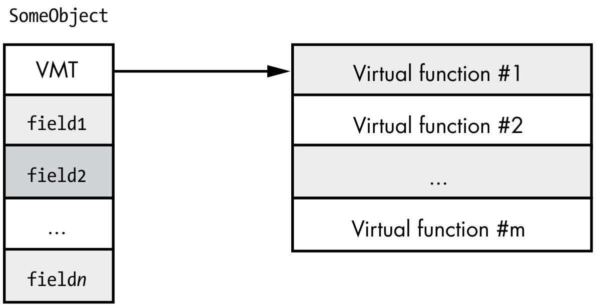


Figure 11-5: A virtual method table in C++

Calling a virtual member function requires *two* indirect accesses. First, the program has to fetch the VMT pointer from the class object and use that to indirectly fetch a particular virtual function address from the VMT. Then the program has to make an indirect call to the virtual member function via the pointer it retrieved from the VMT. As an example, consider the following C++ function:

```
#include <stdlib.h>

// A C++ class with two trivial
// member functions (so the VMT
// will have two entries).

class myclass
{
    public:
        int a;
        int b;
        virtual int f( void );
        virtual int g( void );
};

// Some trivial member functions.
// We're really only interested
// in looking at the calls, so
// these functions will suffice
// for now.

int myclass::f( void )
{
    return b;
}

int myclass::g( void )
{
    return a;
}


// A main function that creates
// a new instance of myclass and
// then calls the two member functions

int main( int argc, char **argv )
{
    myclass *c;

    // Create a new object:

    c = new myclass;

    // Call both member functions:

    c->a = c->f() + c->g();
    return 0;

}
```

Here's the corresponding x86-64 assembly code that Visual C++ generates:

```
; Here is the VMT for myclass. It contains
; three entries:
; a pointer to the constructor for myclass,
; a pointer to the myclass::f member function,
; and a pointer to the myclass::g member function.

CONST   SEGMENT
??_7myclass@@6B@ DQ FLAT:??_R4myclass@@6B@ ; myclass::`vftable'
        DQ      FLAT:?f@myclass@@UEAAHXZ
        DQ      FLAT:?g@myclass@@UEAAHXZ
CONST   ENDS
;
```

```
    .
    .
    .
;
; Allocate storage for a new instance of myclass:
; 16 = two 4-byte ints plus 8-byte VMT pointer
        mov     ecx, 16

        call    ??2@YAPEAX_K@Z          ; operator new
        mov     rdi, rax                ; Save pointer to allocated object
        test    rax, rax                ; Did NEW FAIL (returning NULL)?
        je      SHORT $LN3@main

; Initialize VMT field with the address of the VMT:

        lea     rax, OFFSET FLAT:??_7myclass@@6B@
        mov     QWORD PTR [rdi], rax
        jmp     SHORT $LN4@main
$LN3@main:
        xor     edi, edi                ; For failure, put NULL in EDI

; At this point, RDI contains the "THIS" pointer
; that refers to the object in question. In this
; particular code sequence, "THIS" is the address
; of the object whose storage we allocated above.

; Get the VMT into RAX (first indirect access
; needed to make a virtual member function call)

        mov     rax, QWORD PTR [rdi]

        mov     rcx, rdi                ; Pass THIS in RCX
        call    QWORD PTR [rax+8]       ; Call c->f()
        mov     ebx, eax                ; Save function result

        mov     rdx, QWORD PTR [rdi]    ; Load VMT into RDX
        mov     rcx, rdi                ; Pass THIS in RCX
        call    QWORD PTR [rdx]         ; Call c->g()

; Compute sum of function results:

        add     ebx, eax
        mov     DWORD PTR [rdi+8], ebx  ; Save sum in c->a
```

This example amply demonstrates why object-oriented programs generally run a little more slowly than standard procedural programs: extra indirection when calling virtual methods. C++ attempts to address this inefficiency by providing *static member functions*, but they lose many of the benefits of virtual member functions that make object-oriented programming possible.

11.5.7 Abstract Methods

Some languages (such as C++) allow you to declare *abstract methods* within a class. An abstract method declaration tells the compiler that you will not be supplying the actual code for that method. Instead, you're promising that some derived class will provide the method's implementation. Here's a version of myclass that has an abstract method:

```
class myclass
{
public:
    int a;
    int b;
    virtual int f(void);
    virtual int g(void);
    virtual int h(void) = 0;
};
```

Why the strange syntax? It doesn't really make sense to assign 0 to a virtual function. Why not just use an abstract keyword (rather than virtual) like most other languages do? These are good questions. The answer probably has a lot to do with the fact that a 0 (NULL pointer) was being placed in the VMT entry for the abstract function. In modern versions of C++, compiler implementers typically place the address of some function that generates an appropriate runtime message (like cannot call an abstract method) here, rather than a NULL pointer. The following code snippet shows the Visual C++ VMT for this version of myclass:

```
CONST   SEGMENT
??_7myclass@@6B@ DQ FLAT:??_R4myclass@@6B@           ; myclass::`vftable'
        DQ      FLAT:?f@myclass@@UEAAHXZ
        DQ      FLAT:?g@myclass@@UEAAHXZ
        DQ      FLAT:_purecall
CONST   ENDS
```

The _purecall entry corresponds to the abstract function h(). This is the name of the subroutine that handles illegal calls to abstract functions. When you override an abstract function, the C++ compiler replaces the pointer in the VMT to the _purecall function with the address of the overriding function (just as it would replace the address of any overridden function).

11.5.8 Sharing VMTs

For a given class there is only one copy of the VMT in memory. This is a static object, so all objects of a given class type share the same VMT. This is reasonable, because all objects of the same class type have exactly the same member functions (see Figure 11-6).

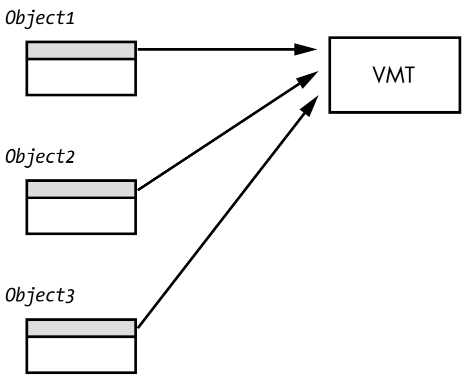


Figure 11-6: Objects sharing the same VMT (note that objects are all the same class type)

Because the addresses in a VMT never change during program execution, most languages place the VMT in a constant (write-protected) section in memory. In the previous example, the compiler places the `myclass` VMT in the `CONST` segment.

11.5.9 Inheritance in Classes

Inheritance is one of the fundamental concepts behind object-oriented programming. The basic idea is that a class inherits, or copies, all the fields from some existing class and then possibly expands the number of fields in the new class data type. For example, suppose you created a data type point that describes a point in the planar (two-dimensional) space. The class for this point might look like the following:

```
class point
{
    public:
        float x;
        float y;

        virtual float distance( void );
};
```

The `distance()` member function would probably compute the distance from the origin (0,0) to the coordinate specified by the (x,y) fields of the object.

Here's a typical implementation of this member function:

```
float point::distance( void )
{
    return sqrt( x*x + y*y );
}
```

Inheritance allows you to extend an existing class by adding new fields or replacing existing fields. For example, suppose you want to extend the two-dimensional point definition to a third spatial dimension. You can easily do this with the following C++ class definition:

```
class point3D :public point
{
    public:
        float z;

        virtual void rotate( float angle1, float angle2 );
};
```

The `point3D` class inherits the `x` and `y` fields, as well as the `distance()` member function. (Of course, `distance()` does not compute the proper result for a point in three-dimensional space, but I'll address that in a moment.) By "inherits," I mean that `point3D` objects locate their `x` and `y` fields at exactly the same offsets as `point` objects do (see Figure 11-7).

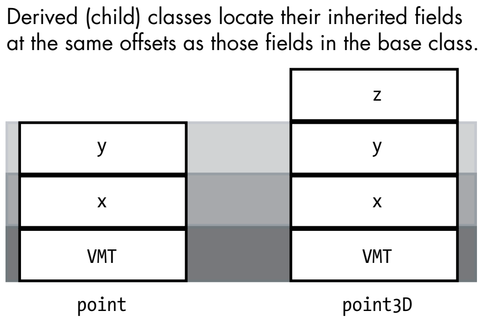

Derived (child) classes locate their inherited fields at the same offsets as those fields in the base class.
z
y
y
x
x
VMT
VMT
point
point3D

Figure 11-7: Inheritance in classes

As you might have noticed, there were actually two items added to the `point3D` class—a new data field, `z`, and a new member function, `rotate()`. In Figure 11-7, you can see that adding the `rotate()` virtual member function has had no impact at all on the layout of a `point3D` object. This is because virtual member functions' addresses appear in the VMT, not in the object itself. Although both `point` and `point3D` contain a field named `VMT`, these fields do not point at the same table in memory. Every class has its own unique VMT, which, as previously defined, consists of an array of pointers to all of the member functions (inherited or explicitly declared) for the class (see Figure 11-8).

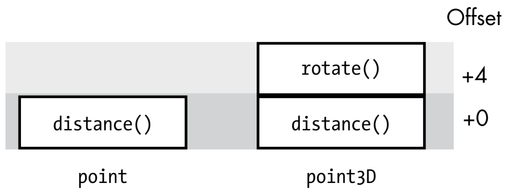


Figure 11-8: VMTs for inherited classes (assuming 32-bit pointers)

All the objects for a given class share the same VMT, but this is not true for objects of different classes. Because `point` and `point3D` are different classes, their objects' VMT fields will point at different VMTs in memory. (See Figure 11-9.)

```
point       p1;
point       p2;
point       p3;
point3D     t1;
point3D     t2;
```

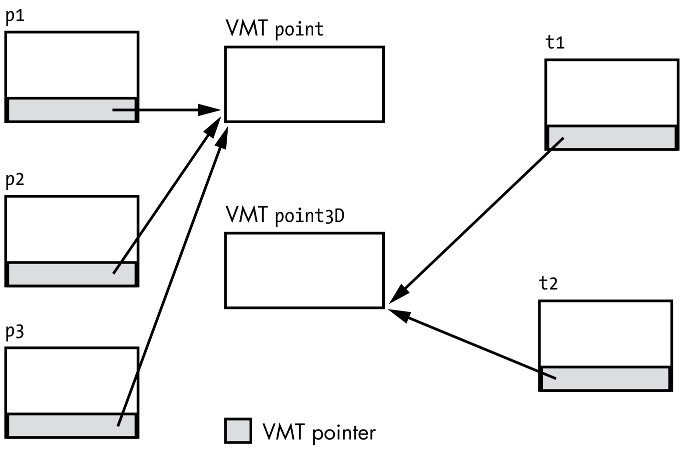


Figure 11-9: VMT access

One problem with the `point3D` definition given thus far is that it inherits the `distance()` function from the `point` class. By default, if a class inherits member functions from some other class, the entries in the VMT corresponding to those inherited functions will point at the functions associated with the base class. If you have an object pointer variable of type `point3D`, let's say `p3D`, and you invoke the member function `p3D->distance()`, you will not get a correct result. Because `point3D` inherits the `distance()` function from class `point`, `p3->distance()` will compute the distance to the projection of `(x,y,z)` onto the two-dimensional plane rather than the correct value on the three-dimensional plane. In C++ you can overcome this problem by *overloading* the inherited function and writing a new, `point3D`-specific member function like so:

```
class point3D :public point
{
    public:
        float z;

        virtual float distance( void );
        virtual void rotate( float angle1, float angle2 );
};
```

```
float point3D::distance( void )
{
    return sqrt( x*x + y*y + z*z );
}
```

Creating an overloaded member function does not change the layout of the class's data or the layout of the `point3D` VMT. The only change this function evokes is that the C++ compiler initializes the `distance()` entry in the `point3D` VMT with the address of the `point3D::distance()` function rather than the address of the `point::distance()` function.

11.5.10 Polymorphism in Classes

In addition to inheritance and overloading, *polymorphism* is the other anchor upon which object-oriented programming is based. Polymorphism, which literally means "many-faced" (or, translated a little better, "many forms" or "many shapes"), describes how a single instance of a function call in your program, such as `x->distance()`, could wind up calling different functions (in the examples from the previous section, this could be the `point::distance()` or `point3D::distance()` function). What makes this possible is the fact that C++ relaxes its type-checking facilities a bit when dealing with derived (inherited) classes.

Let's look at an example. Normally, a C++ compiler will generate an error if you try to do the following:

```
float f;
int *i;
    .
    .
    .
i = &f; // C++ isn't going to allow this.
```

C++ does not allow you to assign the address of some object to a pointer whose base type doesn't exactly match the object's type—with one major exception. C++ relaxes this restriction so you can assign the address of some object to a pointer as long as the pointer's base type either matches *or is an ancestor of* the object's type (an ancestor class is one from which some other class type is derived, directly or indirectly, via inheritance). That means the following code is legal:

```
point *p;
point3D *t;
point *generic;

    p = new point;
    t = new point3D;
        .
        .
        .
    generic = t;
```

If you're wondering how this could be legitimate, take another look at Figure 11-7. If generic's base type is point, then the C++ compiler will allow access to a VMT at offset 0 in the object, an x field at offset 4 (8 on 64-bit machines) in the object, and a y field at offset 8 (16) in the object. Similarly, any attempt to invoke the distance() member function will access the function pointer at offset 0 into the VMT pointed at by the object's VMT field. If generic points at an object of type point, all of these requirements are satisfied. This is also true if generic points at any derived class of point (that is, any class that inherits the fields from point). None of the extra fields in the derived class (point3D) will be accessible via the generic pointer, but that's to be expected because generic's base class is point.

A crucial thing to note, however, is that when you invoke the distance() member function, you're calling the one pointed at by the point3D VMT, not the one pointed at by the point VMT. This fact is the basis for polymorphism in an OOP language such as C++. The code a compiler emits is exactly the same code it would emit if generic contained the address of an object of type point. All of the "magic" occurs because the compiler allows the programmer to load the address of a point3D object into generic.

11.5.11 Multiple Inheritance (in C++)

C++ is one of the few modern programming languages that support *multiple inheritance*, whereby a class can inherit the data and member functions from multiple classes. Consider the following C++ code fragment:

```
class a
{
    public:
        int i;
        virtual void setI(int i) { this->i = i; }
};

class b
{
    public:
        int j;
        virtual void setJ(int j) { this->j = j; }
};

class c : public a, public b
{
    public:
        int k;
        virtual void setK(int k) { this->k = k; }
};
```

In this example, class c inherits all the information from classes a and b. In memory, a typical C++ compiler will create an object like that shown in Figure 11-10.

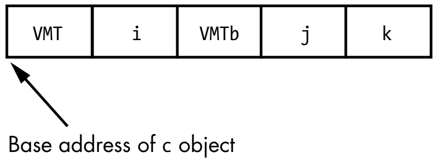


Figure 11-10: Multiple inheritance memory layout

The VMT pointer entry points at a typical VMT containing the addresses of the `setI()`, `setJ()`, and `setK()` methods, as shown in Figure 11-11. If you call the `setI()` method, the compiler will generate code that loads the `this` pointer with the address of the VMT pointer entry in the object (the base address of the `c` object in Figure 11-10). Upon entry into `setI()`, the system believes that `this` is pointing at an object of type `a`. In particular, the `this.VMT` field points at a VMT whose first (and, as far as type `a` is concerned, only) entry is the address of the `setI()` method. Likewise, at offset (`this+8`) in memory (as the VMT pointer is 8 bytes, assuming 64-bit pointers), the `setI()` method will find the `i` data value. As far as `setI()` is concerned, `this` is pointing at a class type `a` object (even though it's actually pointing at a type `c` object).

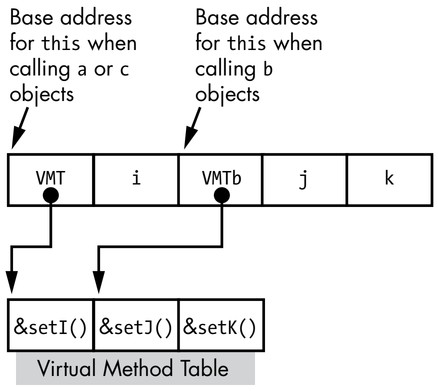

Base address for this when calling a or c objects
Base address for this when calling b objects
VMT
i
VMTb
j
k
&setI()
&setJ()
&setK()
Virtual Method Table

Figure 11-11: Multiple inheritance `this` *values*

When you call the `setK()` method, the system also passes the base address of the `c` object. Of course, `setK()` is expecting a type `c` object and `this` is pointing at a type `c` object, so all the offsets into the object are exactly as `setK()` expects. Note that objects of type `c` (and methods in the `c` class) will normally ignore the `VMT2` pointer field in the `c` object.

The problem occurs when the program attempts to call the `setJ()` method. Because `setJ()` belongs to class `b`, it expects `this` to hold the address of a VMT pointer pointing at a VMT for class `b`. It also expects to find data field `j` at offset (`this+8`). Were we to pass the `c` object's `this` pointer to `setJ()`, accessing (`this+8`) would reference the `i` data field, not `j`. Furthermore, were a class `b` method to make a call to another method in class `b` (such as `setJ()` making a recursive call to itself), the VMT pointer would be wrong—it points at a VMT with a pointer to `setI()` at offset 0, whereas class `b` expects it to point at a VMT with a pointer to `setJ()` at offset 0. To resolve this issue, a typical C++ compiler will insert an extra VMT pointer into the c object immediately prior to the j data field. It will initialize this second VMT field to point into the c VMT at the location where the class b method pointers begin (see Figure 11-11). When calling a method in class b, the compiler will emit code that initializes the this pointer with the address of this second VMT pointer (rather than pointing at the beginning of c-type object in memory). Now, upon entry to a class b method—such as setJ()—this will point at a legitimate VMT pointer for class b, and the j data field will appear at the offset (this+8) that class b methods expect.

11.6 Protocols and Interfaces

Java and Swift don't support multiple inheritance, because it has some logical problems. The classic example is the "diamond lattice" data structure. This occurs when two classes (say, b and c) both inherit information from the same class (say, a), and then a fourth class (say, d) inherits from both b and c. As a result, d inherits the data from a twice—once through b and once through c. This can create some consistency problems.

Although multiple inheritance can lead to some weird problems like this, there's no question that being able to inherit from multiple locations is often useful. Thus, the solution in languages like Java and Swift is to allow a class to inherit methods/functions from multiple parents but allow inheritance from only a single ancestor class. This avoids most of the problems with multiple inheritance (specifically, an ambiguous choice of inherited data fields) while allowing programmers to include methods from various sources. Java calls such extensions *interfaces*, and Swift calls them *protocols*.

Here's an example of a couple Swift protocol declarations and a class supporting that protocol:

```
protocol someProtocol
{
    func doSomething()->Void;
    func doSomethingElse() ->Void;
}
protocol anotherProtocol
{
    func doThis()->Void;
    func doThat() ->Void;
}

class supportsProtocols: someProtocol, anotherProtocol
{
    var i:Int = 0;
    func doSomething()->Void
    {
        // appropriate function body
    }
    func doSomethingElse()->Void
    {
        // appropriate function body
    }
    func doThis()->Void
    {
        // appropriate function body
    }
    func doThat()->Void
    {
        // appropriate function body
    }

}
```

Swift protocols don't supply any functions. Instead, a class that supports a protocol promises to provide an implementation of the functions the protocol(s) specify. In the preceding example, the `supportsProtocols` class is responsible for supplying all functions required by the protocols it supports. Effectively, protocols are like abstract classes containing only abstract methods—the inheriting class must provide actual implementations for all the abstract methods.

Here's the previous example coded in Java and demonstrating its comparable mechanism, the interface:

```
interface someInterface
{
    void doSomething();
    void doSomethingElse();
}
interface anotherInterface
{
    void doThis();
    void doThat();
}

class supportsInterfaces  implements someInterface, anotherInterface
{
    int i;
    public void doSomething()
    {
        // appropriate function body
    }
    public void doSomethingElse()
    {
        // appropriate function body
    }
    public void doThis()
    {
        // appropriate function body
    }
    public void doThat()
    {
        // appropriate function body
    }

}
```

Interfaces/protocols behave somewhat like base class types in Java and Swift. If you instantiate a class object and assign that instance to a variable that is an interface/protocol type, you can execute the supported member functions for that interface/protocol. Consider the following Java example:

```
someInterface some = new supportsInterfaces();

// We can call the member functions defined for someInterface:

some.doSomething();
some.doSomethingElse();

// Note that it is illegal to try and call doThis or doThat
// (or access the i data field)
// using the "some" variable.
```

Here's a comparable example in Swift:

```
import Foundation

protocol a
{
    func b()->Void;
    func c()->Void;
}

protocol d
{
    func e()->Void;
    func f()->Void;
}
class g : a, d
{
    var i:Int = 0;

    func b()->Void {print("b")}
    func c()->Void {print("c")}
    func e()->Void {print("e")}
    func f()->Void {print("f")}

    func local()->Void {print( "local to g" )}
}


var x:a = g()
x.b()
x.c()
```

The implementation of a protocol or interface is quite simple—it's just a pointer to a VMT that contains the addresses of the functions declared in that protocol/interface. So, the data structure for the Swift `g` class in the previous example would have three VMT pointers in it: one for protocol `a`, one for protocol `d`, and one for the class `g` (holding a pointer to the `local()` function). Figure 11-12 shows the class and VMT layout.

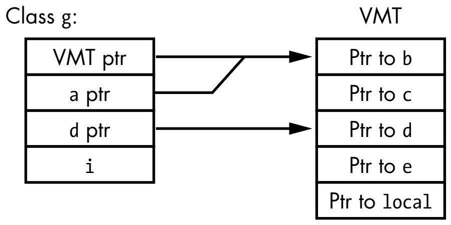


Figure 11-12: Multiple inheritance memory layout

In Figure 11-12 the VMT pointer for class `g` contains the address of the entire VMT. There are two entries in the class that contain pointers to the VMTs for protocol `a` and protocol `d`. As the VMT for class `g` also contains pointers to the functions belonging to these protocols, there's no need to create a separate VMT for these two protocols; instead, the `aPtr` and `dPtr` fields can point to the corresponding entries within class `g`'s VMT.

When the assignment `var x:a = g()` occurs in the previous example, the Swift code will load variable `x` with the `aPtr` pointer held in the `g` object. Therefore, the calls to `x.b()` and `x.c()` work just like a normal method call—the system uses the pointer held in `x` to reference the VMT and then it calls `b` or `c` by indexing the appropriate amount into the VMT. Had `x` been of type `d` rather than `a`, then the assignment `var x:d = g()` would have loaded `x` with the address of the `d` protocol VMT (pointed at by `dPtr`). Calls to `d` and `e` would happen at offsets 0 and 8 (64-bit pointers) into the `d` VMT.

11.7 Classes, Objects, and Performance

As you've seen in this chapter, the direct cost associated with object-oriented programming isn't terribly significant. Calls to member functions (methods) are a bit more expensive because of double indirection; however, that's a small price to pay for the flexibility OOP gives you. The extra instructions and memory accesses will probably cost only about 10 percent of your application's total performance. Some languages, such as C++ and HLA, support the notion of a *static member function* that allows direct calls to member functions when polymorphism is unnecessary.

The big problem that object-oriented programmers sometimes face is taking things to an extreme. Rather than directly accessing the fields of an object, they write accessor functions to read and write those field values. Unless the compiler does a very good job of inlining such accessor functions, the cost of accessing the object's fields increases by about an order of magnitude. In other words, application performance can actually suffer when OOP paradigms are overused. There may be good reasons for doing things the "object-oriented way" (such as using accessor functions to access all fields of an object), but keep in mind that these costs add up rather quickly. Unless you absolutely need the facilities provided by OOP techniques, your programs may wind up running considerably slower (and taking up a whole lot more space) than necessary.

Swift is a good example of object-oriented programming taken to an extreme. Anyone who has compared the performance of compiled Swift code against an equivalent C++ program knows that Swift is much slower. Largely, this is because Swift makes objects out of everything (and constantly checks their types and bounds at runtime). The result is that it can take hundreds of machine instructions in Swift to do the same task as a half-dozen machine instructions produced by an optimizing C++ compiler.

Another common problem with many object-oriented programs is overgeneralization. This can occur when a programmer uses a lot of class libraries, extending classes through inheritance in order to solve some problem with as little programming effort as possible. While saving programming effort is generally a good idea, extending class libraries can lead to situations where you need some minor task done and you call a library routine that does everything you want. The problem is that in object-oriented systems, library routines tend to be highly layered. That is, you need some work done, so you invoke some member function from a class you've inherited. That function probably does a little bit of work on the data you pass it and then it calls a member function in a class that it inherits. And then that function massages the data a bit and calls a member function it inherits, and so on down the line. Before too long, the CPU spends more time calling and returning from functions than it does doing any useful work. While this same situation could occur in standard (non-OOP) libraries, it's far more common in object-oriented applications.

Carefully designed object-oriented programs needn't run significantly slower than comparable procedural programs. Just be careful not to make a lot of expensive function calls to do trivial tasks.

11.8 For More Information

Dershem, Herbert, and Michael Jipping. *Programming Languages, Structures and Models.* Belmont, CA: Wadsworth, 1990.

Duntemann, Jeff. *Assembly Language Step-by-Step.* 3rd ed. Indianapolis: Wiley, 2009.

Ghezzi, Carlo, and Jehdi Jazayeri. *Programming Language Concepts.* 3rd ed. New York: Wiley, 2008.

Hyde, Randall. *The Art of Assembly Language.* 2nd ed. San Francisco: No Starch Press, 2010.

Knuth, Donald. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* 3rd ed. Boston: Addison-Wesley Professional, 1997.

Ledgard, Henry, and Michael Marcotty. *The Programming Language Landscape*. Chicago: SRA, 1986.

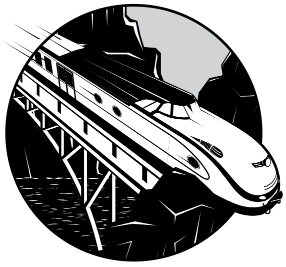
Louden, Kenneth C., and Kenneth A. Lambert. *Programming Languages, Principles and Practice*. 3rd ed. Boston: Course Technology, 2012.

Pratt, Terrence W., and Marvin V. Zelkowitz. *Programming Languages, Design and Implementation*. 4th ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Sebesta, Robert. *Concepts of Programming Languages*. 11th ed. Boston: Pearson, 2016.

12

ARITHMETIC AND LOGICAL EXPRESSIONS



One of the major advantages that high-level languages provide over low-level languages is the use of algebraic arithmetic and logical expressions (hereafter, "arithmetic expressions"). HLL arithmetic expressions are an order of magnitude more readable than the sequence of machine instructions the compiler produces. However, the conversion process from arithmetic expressions into machine code is also one of the more difficult transformations to do efficiently, and a fair percentage of a typical compiler's optimization phase is dedicated to handling it. Because of the difficulty with translation, this is one area where you can help the compiler. This chapter will describe:

- How computer architecture affects the computation of arithmetic expressions
- The optimization of arithmetic expressions

- Side effects of arithmetic expressions
- Sequence points in arithmetic expressions
- Order of evaluation in arithmetic expressions
- Short-circuit and complete evaluation of arithmetic expressions
- The computational cost of arithmetic expressions

Armed with this information, you'll be able to write more efficient and more robust applications.

12.1 Arithmetic Expressions and Computer Architecture

With respect to arithmetic expressions, we can classify traditional computer architectures into three basic types: stack-based machines, register-based machines, and accumulator-based machines. The major difference between these architectural types has to do with where the CPUs keep the operands for the arithmetic operations. Once the CPU fetches the data from these operands, the data is passed along to the arithmetic and logical unit, where the actual arithmetic or logical calculation occurs.[1] We'll explore each of these architectures in the following sections.

12.1.1 Stack-Based Machines

Stack-based machines use memory for most calculations, employing a data structure called the *stack* in memory to hold all operands and results. Computer systems with a stack architecture offer some important advantages over other architectures:

- The instructions are often smaller in stack architectures because the instructions generally don't have to specify any operands.
- It is usually easier to write compilers for stack architectures than for other machines because converting arithmetic expressions to a sequence of stack operations is very easy.
- Temporary variables are rarely needed in a stack architecture, because the stack itself serves that purpose.

Unfortunately, stack machines also suffer from some serious disadvantages:

- Almost every instruction references memory (which is slow on modern machines). Though caches can help mitigate this problem, memory performance is still a major problem on stack machines.

1. All calculations are logical in nature. Even arithmetic operations such as addition and subtraction are "logical" in the sense that the CPU computes their result based on a series of Boolean expressions. For our purposes, therefore, "logical expression" and "arithmetic expression" are synonymous. See *WGC1* for more details concerning Boolean expressions and low-level arithmetic.

- Even though conversion from HLLs to a stack machine is very easy, there's less opportunity for optimization than there is with other architectures.
- Because stack machines are constantly accessing the same data elements (that is, data on the *top of the stack*), pipelining and instruction parallelism is difficult to achieve.

NOTE *See* WGC1 *for details on pipelining and instruction parallelism.*

With a stack you generally do one of three things: push new data onto it, pop data from it, or operate on the data that is currently sitting on the *top of stack* (and possibly the data immediately below that, or *next on stack*).

12.1.1.1 Basic Stack Machine Organization

A typical stack machine maintains a couple of registers inside the CPU (see Figure 12-1). In particular, you can expect to find a *program counter register* (like the 80x86's RIP register) and a *stack pointer register* (like the 80x86 RSP register).

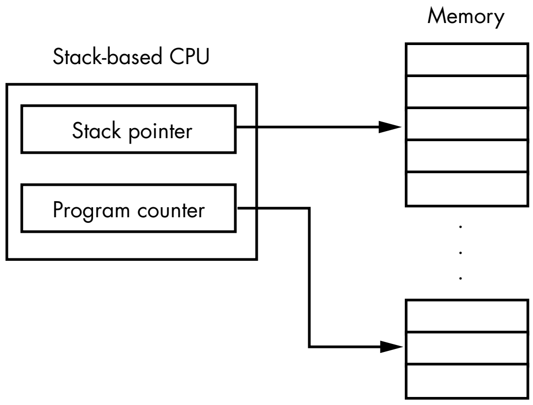


Figure 12-1: Typical stack machine architecture

The stack pointer register contains the memory address of the current top of stack (TOS) element in memory. The CPU increments or decrements the stack pointer register whenever a program places data onto the stack or removes data from the stack. On some architectures the stack expands from higher memory locations to lower memory locations; on other architectures, the stack grows from lower memory locations toward higher memory locations. Fundamentally, the direction of stack growth is irrelevant; all it really determines is whether the machine decrements the stack pointer register when placing data on the stack (if the stack grows toward lower memory addresses) or increments the stack pointer register (when the stack grows toward higher memory addresses).

12.1.1.2 The push Instruction

To place data on the stack, you typically use the machine instruction `push`. This instruction generally takes a single operand that specifies the value to push onto the stack, like so:

```
push memory or constant operand
```

Here are a couple of concrete examples:

```
push 10  ; Pushes the constant 10 onto the stack
push mem ; Pushes the contents of memory location mem
```

A `push` operation typically increases the value of the stack pointer register by the size of its operand in bytes and then copies that operand to the memory location the stack pointer now specifies. For example, Figures 12-2 and 12-3 illustrate what the stack looks like before and after a `push 10` operation.

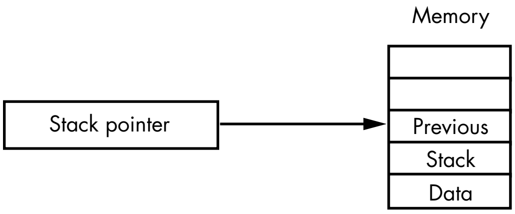


Figure 12-2: Before a `push 10` operation

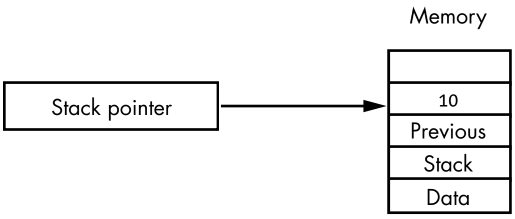


Figure 12-3: After a `push 10` operation

12.1.1.3 The pop Instruction

To remove a data item from the top of a stack, you use a `pop` or `pull` instruction. (This book will use `pop`; just be aware that some architectures use `pull` instead.) A typical `pop` instruction might look as follows:

```
pop memory location
```

NOTE *You cannot pop data into a constant. The `pop` operand must be a memory location.*

The `pop` instruction makes a copy of the data pointed at by the stack pointer and stores it into the destination memory location. Then it decrements (or increments) the stack pointer register to point at the next lower item on the stack, or next on stack (NOS); see Figures 12-4 and 12-5.

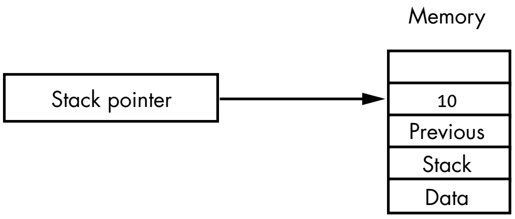


Figure 12-4: Before a `pop mem` *operation*

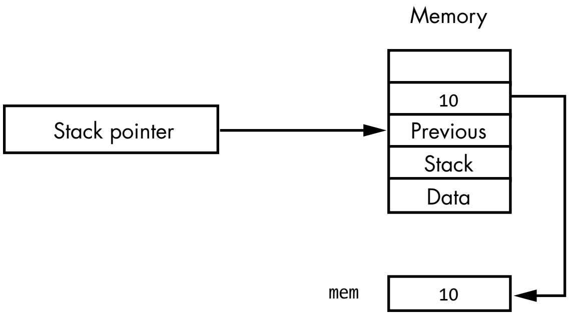


Figure 12-5: After a `pop mem` *operation*

Note that the value in stack memory that the `pop` instruction removes from the stack is still physically present in memory above the new TOS. However, the next time the program pushes data onto the stack, it will overwrite this value with the new value.

12.1.1.4 Arithmetic Operations on a Stack Machine

The arithmetic and logical instructions found on a stack machine generally do not allow any operands. This is why stack machines are often called *zero-address machines*; the arithmetic instructions themselves do not encode any operand addresses. For example, consider an `add` instruction on a typical stack machine. This instruction will pop two values from the stack (TOS and NOS), compute their sum, and push the result back onto the stack (see Figures 12-6 and 12-7).

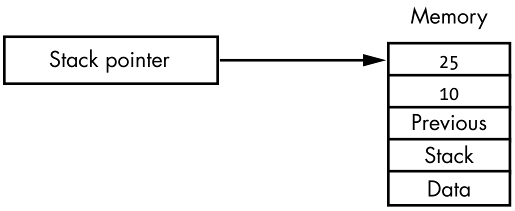


Figure 12-6: Before an `add` *operation*

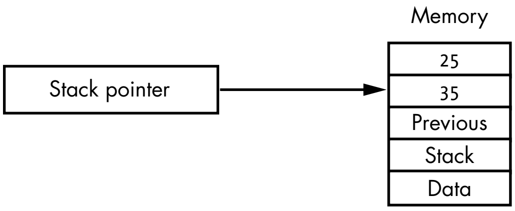


Figure 12-7: After an `add` *operation*

Because arithmetic expressions are recursive in nature, and recursion requires a stack for proper implementation, it's no surprise that converting arithmetic expressions to a sequence of stack machine instructions is relatively simple. Arithmetic expressions found in common programming languages use an *infix notation*, where the operator appears between two operands. For example, `a + b` and `c - d` are examples of infix notation because the operators (`+` and `-`) appear between the operands ([`a`, `b`] and [`c`, `d`]). Before you can do the conversion to stack machine instructions, you must convert these infix expressions into *postfix notation* (also known as *reverse polish notation*), where the operator immediately follows the operands to which it applies. For example, the infix expressions `a + b` and `c - d` would have the corresponding postfix forms `a b +` and `c d -`, respectively.

Once you have an expression in postfix form, converting it to a sequence of stack machine instructions is very easy. You simply emit a `push` instruction for each operand and the corresponding arithmetic instruction for the operators. For example, `a b +` becomes:

```
push a
push b
add
```

and `c d -` becomes:

```
push c
push d
sub
```

assuming, of course, that `add` adds the top two items on the stack and `sub` subtracts the TOS from the value immediately below it on the stack.

12.1.1.5 Real-World Stack Machines

A big advantage of the stack architecture is that it's easy to write a compiler for such a machine. It's also very easy to write an emulator for a stack-based machine. For these reasons, stack architectures are popular in *virtual machines (VMs)* such as the Java Virtual Machine, the UCSD Pascal p-machine, and the Microsoft Visual Basic, C#, and F# CIL. Although a few real-world stack-based CPUs do exist, such as a hardware implementation of the Java VM, they're not very popular because of the performance limitations of memory access. Nonetheless, understanding the basics of a stack architecture is important, because many compilers translate HLL source code into a stack-based form prior to emitting actual machine code. Indeed, in the worst (though rare) case, compilers are forced to emit code that emulates a stack-based machine when compiling complex arithmetic expressions.

12.1.2 Accumulator-Based Machines

The simplicity of a stack machine instruction sequence hides an enormous amount of complexity. Consider the following stack-based instruction from the previous section:

```
add
```

This instruction looks simple, but it actually specifies a large number of operations:

- Fetch an operand from the memory location pointed to by the stack pointer.
- Send the stack pointer's value to the *ALU (arithmetic/logical unit)*.
- Instruct the ALU to decrement the stack pointer's value just sent to it.
- Route the ALU's value back to the stack pointer.
- Fetch the operand from the memory location pointed to by the stack pointer.
- Send the values from the previous step and the first step to the ALU.
- Instruct the ALU to add those values.
- Store the sum away in the memory location pointed to by the stack pointer.

The organization of a typical stack machine prevents many parallel operations that are possible with pipelining (see *WGC1* for more details on pipelining). So stack architectures are hit twice: typical instructions require many steps to complete, and those steps are difficult to execute in parallel with other operations.

One big problem with the stack architecture is that it goes to memory for just about everything. For example, if you simply want to compute the sum of two variables and store this result in a third variable, you have to fetch the two variables and write them to the stack (four memory operations); then you have to fetch the two values from the stack, add them, and write their sum back to the stack (three memory operations); and finally, you have to pop the item from the stack and store the result into the destination memory location (two memory operations). That's a total of nine memory operations. When memory access is slow, this is an expensive way to compute the sum of two numbers.

One way to avoid this large number of memory accesses is to provide a general-purpose arithmetic register within the CPU. This is the idea behind an accumulator-based machine: you provide a single *accumulator* register, where the CPU computes temporary results rather than computing temporary values in memory (on the stack). Accumulator-based machines are also known as *one-address* or *single-address machines*, because most instructions that operate on two operands use the accumulator as the default destination operand for the computation and require a single memory or constant operand to use as the source operand. A typical example of an accumulator machine is the 6502, which includes the following instructions:

```
LDA constant or memory ; Load accumulator register
STA memory             ; Store accumulator register
ADD constant or memory ; Add operand to accumulator
SUB constant or memory ; Subtract operand from accumulator
```

Because one-address instructions require an operand that isn't present in many of the zero-address instructions, individual instructions found on an accumulator-based machine tend to be larger than those found on a typical stack-based machine (because you have to encode the operand address as part of the instruction; see *WGC1* for details). However, programs are often smaller because fewer instructions are needed to do the same thing. Suppose, for example, you want to compute `x = y + z`. On a stack machine, you might use an instruction sequence like the following:

```
push y
push z
add
pop x
```

On an accumulator machine, you'd probably use a sequence like this:

```
lda y
add z
sta x
```

Assuming that the `push` and `pop` instructions are roughly the same size as the accumulator machine's `lda`, `add`, and `sta` instructions (a safe assumption), it's clear that the stack machine's instruction sequence is actually longer, because it requires more instructions. Even ignoring the extra instruction on the stack machine, the accumulator machine will probably execute the code faster, because it requires only three memory accesses (to fetch `y` and `z` and to store `x`), compared with the nine memory accesses the stack machine will require. Furthermore, the accumulator machine doesn't waste any time manipulating the stack pointer register during computation.

Even though accumulator-based machines generally have higher performance than stack-based machines (for reasons you've just seen), they're not without their own problems. Having only one general-purpose register available for arithmetic operations creates a bottleneck in the system, resulting in *data hazards*. Many calculations produce temporary results that the application must write to memory in order to compute other components of the expression. This leads to extra memory accesses that could be avoided if the CPU provided additional accumulator registers. Thus, most modern general-purpose CPUs do not use an accumulator-based architecture, but instead provide a large number of general-purpose registers.

NOTE *See* WGC1 *for a discussion of data hazards.*

Accumulator-based architectures were popular in early computer systems when the manufacturing process limited the number of features within the CPU, but today you rarely see them outside of low-cost embedded microcontrollers.

12.1.3 Register-Based Machines

Of the three architectures discussed in this chapter, register-based machines are the most prevalent today because they offer the highest performance. By providing a fair number of on-CPU registers, this architecture spares the CPU from expensive memory accesses during the computation of complex expressions.

In theory, a register-based machine could have as few as two general-purpose (arithmetic-capable) registers. In practice, about the only machines that fall into this category are the Motorola 680x processors, which most people consider to be a special case of the accumulator architecture with two separate accumulators. Register machines generally contain at least eight "general-purpose" registers (this number isn't arbitrary; it's the number of general-purpose registers found on the 80x86 CPU, the 8080 CPU, and the Z80 CPU, which are probably the minimalist examples of what a computer architect would call a "register-based" machine).

Although some register-based machines (such as the 32-bit 80x86) have a small number of registers available, a general principle is "the more, the better." Typical RISC machines, such as the PowerPC and ARM, have at least 16 general-purpose registers and often at least 32 registers. Intel's Itanium processor, for example, provides 128 general-purpose integer registers. IBM's CELL processor provides 128 registers in each of the processing units found on the device (each processing unit is a mini-CPU capable of certain operations); a typical CELL processor contains eight such processing units along with a PowerPC CPU core.

The main reason for having as many general-purpose registers as possible is to avoid memory access. In an accumulator-based machine, the accumulator is a transient register used for calculations, but you can't keep a variable's value there for long periods of time, because you'll need the accumulator for other purposes. In a register machine with a large number of registers, it's possible to keep certain (often-used) variables in registers so you don't have to access memory at all when using those variables. Consider the assignment statement `x := y+z;`. On a register-based machine (such as the 80x86), we could compute this result using the following HLA code:

```
// Note: Assume x is held in EBX, y is held in ECX,
// and z is held in EDX:

mov( ecx, ebx );
add( edx, ebx );
```

Only two instructions and no memory accesses (for the variables) are required here. This is quite a bit more efficient than the accumulator- or stack-based architectures. From this example, you can see why the register-based architecture has become prevalent in modern computer systems.

As you'll see in the following sections, register machines are often described as either two-address machines or three-address machines, depending on the particular CPU's architecture.

12.1.4 Typical Forms of Arithmetic Expressions

Computer architects have studied typical source files extensively, and one thing they've discovered is that a large percentage of assignment statements take one of the following forms:

```
var = var2;
var = constant;
var = op var2;
var = var op var2;
var = var2 op var3;
```

Although other assignments do exist, the set of statements in a program that takes one of these forms is generally larger than any other group of assignment statements. Therefore, computer architects usually optimize their CPUs to efficiently handle these forms.

12.1.5 Three-Address Architectures

Many machines use a *three-address architecture*. This means that an arithmetic statement supports three operands: two source operands and a destination operand. For example, most RISC CPUs offer an `add` instruction that will add together the values of two operands and store the result into a third operand:

```
add source1, source2, dest
```

On such architectures, the operands are usually machine registers (or small constants), so typically you'd write this instruction as follows (assuming you use the names *R*0, *R*1, . . . , *Rn* to denote registers):

```
add r0, r1, r2   ; computes r2 := r0 + r1
```

Because RISC compilers attempt to keep variables in registers, this single instruction handles the last assignment statement given in the previous section:

```
var = var2 op var3;
```

Handling an assignment of the form:

```
var = var op var2;
```

is also relatively easy—just use the destination register as one of the source operands, like so:

```
add r0, r1, r0  ; computes r0 := r0 + r1
```

The drawback to a three-address architecture is that you must encode all three operands into each instruction that supports three operands. This is why three-operand instructions generally operate only upon register operands. Encoding three separate memory addresses can be quite expensive—just ask any VAX programmer. The DEC VAX computer system is a good example of a three-address CISC machine.

12.1.6 Two-Address Architectures

The 80x86 architecture is known as a *two-address machine*. In this architecture, one of the source operands is also the destination operand. Consider the following 80x86/HLA `add` instruction:

```
add( ebx, eax );  ; computes eax := eax + ebx;
```

Two-address machines, such as the 80x86, can handle the first four forms of the assignment statement given earlier with a single instruction. The last form, however, requires two or more instructions and a temporary register. For example, to compute:

```
var1 = var2 + var3;
```

you'd need to use the following code (assuming *var2* and *var3* are memory variables and the compiler is keeping *var1* in the EAX register):

```
mov( var2, eax );
add( var3, eax );  //Result (var1) is in EAX.
```

12.1.7 Architectural Differences and Your Code

One-address, two-address, and three-address architectures have the following hierarchy:

$$\textbf{1Address} \subset \textbf{2Address} \subset \textbf{3Address}$$

That is, two-address machines are capable of doing anything a one-address machine can do, and three-address machines are capable of anything one-address or two-address machines can do. The proof is very simple:[2]

- To show that a two-address machine is capable of doing anything a one-address machine can do, simply choose one register on the two-address machine and use it as the "accumulator" when simulating a one-address architecture.
- To show that a three-address machine is capable of anything a two-address machine can do, simply use the same register for one of the source operands and the destination operand, thereby limiting yourself to two registers (operands/addresses) for all operations.

Given this hierarchy, you might think that if you limit the code you write so that it runs well on a one-address machine, you'll get good results on all machines. In reality, most general-purpose CPUs available today are two- or three-address machines, so writing your code to favor a one-address machine may limit the optimizations that are possible on a two- or three-address machine. Furthermore, optimization quality varies so widely among compilers that backing up an assertion like this would be very difficult. You should probably try to create expressions that take one of the five forms given earlier (in "Typical Forms of Arithmetic Expressions" on page 394) if you want your compiler to produce the best possible code. Because most modern programs run on two- or three-address machines, the remainder of this chapter assumes that environment.

12.1.8 Complex Expressions

Once your expressions get more complex than the five forms given earlier, the compiler will have to generate a sequence of two or more instructions to evaluate them. When compiling the code, most compilers internally translate a complex expression into a sequence of "three-address statements" that are semantically equivalent to it, as in the following example:

```
// complex = ( a + b ) * ( c - d ) - e/f;

temp1 = a + b;
temp2 = c - d;
temp1 = temp1 * temp2;
temp2 = e / f;
complex = temp1 - temp2;
```

2. Technically, completing this proof would require showing that you can do things with a two-address machine that cannot be done with a one-address machine, and that you can do things with a three-address machine that cannot be done with a two-address machine. I'll leave that as an exercise for readers. It's still a fairly simple proof.

As you can see, these five statements are semantically equivalent to the complex expression appearing in the comment. The major difference in the computation is the introduction of two temporary values (`temp1` and `temp2`). Most compilers will attempt to use machine registers to maintain these temporary values.

Because the compiler internally translates a complex instruction into a sequence of three-address statements, you may wonder if you can help it by converting complex expressions into three-address statements yourself. Well, it depends on your compiler. For many (good) compilers, breaking a complex calculation into smaller pieces may, in fact, thwart the compiler's ability to optimize certain sequences. So, when it comes to arithmetic expressions, most of the time you should do your job (write the code as clearly as possible) and let the compiler do its job (optimize the result). However, if you can specify a calculation using a form that naturally converts to a two-address or three-address form, by all means do so. At the very least, it will have no effect on the code the compiler generates. At best, under some special circumstances, it could help the compiler produce better code. Either way, the resulting code will probably be easier to read and maintain if it is less complex.

12.2 Optimization of Arithmetic Statements

Because HLL compilers were originally designed to let programmers use algebraic-like expressions in their source code, this is one area in computer science that has been well researched. Most modern compilers that provide a reasonable optimizer do a decent job of translating arithmetic expressions into machine code. You can usually assume that the compiler you're using doesn't need a whole lot of help with optimizing arithmetic expressions (and if it does, you might consider switching to a better compiler instead of trying to manually optimize the code).

To help you appreciate the job the compiler is doing for you, this section discusses some of the typical optimizations you can expect from modern optimizing compilers. By understanding what a (decent) compiler does, you can avoid hand-optimizing those things that it is capable of handling.

12.2.1 Constant Folding

Constant folding is an optimization that computes the value of constant expressions or subexpressions at compile time rather than emitting code to compute their result at runtime. For example, a Pascal compiler that supports this optimization would translate a statement of the form `i := 5 + 6;` to `i := 11;` prior to generating machine code for the statement. This saves it from emitting an `add` instruction that would have to execute at runtime. As another example, suppose you want to allocate an array containing 16MB of storage. One way to do this is as follows:

```
char bigArray[ 16777216 ]; // 16 MB of storage
```

The only problem with this approach is that 16,777,216 is a magic number. It represents the value 2^{24} and not some other arbitrary value. Now consider the following C/C++ declaration:

```
char bigArray[ 16*1024*1024 ]; // 16 MB of storage
```

Most programmers realize that 1,024 times 1,024 is a binary million, and 16 times this value corresponds to 16 mega-somethings. Yes, you need to recognize that the subexpression `16*1024*1024` is equivalent to 16,777,216. But this pattern is easier to recognize as 16MB (at least, when used within a character array) than `16777216` (or was it `16777214`?). In both cases the amount of storage the compiler allocates is exactly the same, but the second case is, arguably, more readable. Hence, it is better code.[3]

Variable declarations aren't the only place a compiler can use this optimization. Any arithmetic expression (or subexpression) containing constant operands is a candidate for constant folding. Therefore, if you can write an arithmetic expression more clearly by using constant expressions rather than computing the results by hand, you should definitely go for the more readable version and leave it up to the compiler to handle the constant calculation at compile time. If your compiler doesn't support constant folding, you can certainly simulate it by performing all constant calculations manually. However, you should do this only as a last resort. Finding a better compiler is almost always a better choice.

Some good optimizing compilers may take extreme steps when folding constants. For example, some compilers with a sufficiently high optimization level enabled will replace certain function calls, with constant parameters, to the corresponding constant value. For example, a compiler might translate a C/C++ statement of the form `sineR = sin(0);` to `sineR = 0;` during compilation (as the sine of zero radians is `0`). This type of constant folding, however, is not all that common, and you usually have to enable a special compiler mode to get it.

If you ever have any questions about whether your particular compiler supports constant folding, have the compiler generate an assembly listing and look at its output (or view the disassembled output with a debugger). Here's a trivial case written in C/C++ (compiled with Visual C++):

```
#include <stdio.h>
int main(int argc, char **argv)
{
      int i = 16 * 1024 * 1024;
      printf( "%d\n", i);
       return 0;
}
```

3. Of course, using a manifest constant identifier in place of a numeric expression is probably a better solution here. However, at some point or another you need to actually define the constant value. Using `16*1024*1024` in the definition is better than `16777216`.

```
// Assembly output for sequence above (optimizations turned off!)

        mov     DWORD PTR i$[rsp], 16777216             ; 01000000H

        mov     edx, DWORD PTR i$[rsp]
        lea     rcx, OFFSET FLAT:$SG7883
        call    printf
```

Here's a comparable program written in Java:

```
public class Welcome
{
      public static void main( String[] args )
      {
            int i = 16 * 1024 * 1024;
            System.out.println( i );
      }
}

// JBC generated by the compiler:

javap -c Welcome
Compiled from "Welcome.java"
public class Welcome extends java.lang.Object{
public Welcome();
  Code:
   0:   aload_0

        ; //Method java/lang/Object."<init>":()V
   1:   invokespecial   #1

   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   ldc #2; //int 16777216
   2:   istore_1

        ; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   getstatic   #3

   6:   iload_1

        ; //Method java/io/PrintStream.println:(I)V
   7:   invokevirtual   #4   10:  return

}
```

Note that the `ldc #2` instruction pushes a constant from a constant pool onto the stack. The comment attached to this bytecode instruction explains that the Java compiler converted `16*1024*1024` into a single constant `16777216`. Java performs the constant folding at compile time rather than computing this product at runtime.

Here's the comparable program in Swift, along with the assembly code emitted for the relevant portion[4] of the main program:

```
import Foundation

var i:Int = 16*1024*1024
print( "i=\(i)" )

// code produced by
// "xcrun -sdk macosx
//      swiftc -O -emit-assembly main.swift -o result.asm"

       movq    $16777216, _$S6result1iSivp(%rip)
```

As you can see, Swift also supports the constant folding optimization.

12.2.2 Constant Propagation

Constant propagation is an optimization a compiler uses to replace a variable access by a constant value if the compiler determines that it's possible. For example, a compiler that supports constant propagation will make the following optimization:

```
// original code:

    variable = 1234;
    result = f( variable );

// code after constant propagation optimization

    variable = 1234;
    result = f( 1234 );
```

In object code, manipulating immediate constants is often more efficient than manipulating variables; therefore, constant propagation often produces much better code. In some cases, constant propagation also allows the compiler to eliminate certain variables and statements altogether (in this example, the compiler could remove `variable = 1234;` if there are no later references to the variable object in the source code).

In some cases, well-written compilers can do some outrageous optimizations involving constant folding. Consider the following C code:

```
#include <stdio.h>
static int rtn3( void )
{
    return 3;
}
```

4. Swift generates considerable extra code that has no bearing on the question of whether the language performs constant folding, so that extra code does not appear here.

```
int main( void )
{
    printf( "%d", rtn3() + 2 );
    return( 0 );
}
```

Here's the 80x86 output that GCC produces with the -O3 (maximum) optimization option:

```
.LC0:
        .string "%d"
        .text
        .p2align 2,,3
.globl main
        .type   main,@function
main:
        ; Build main's activation record:

        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $8, %esp

        ; Print the result of "rtn3()+5":

        pushl   $5      ; Via constant propagation/folding!
        pushl   $.LC0
        call    printf
        xorl    %eax, %eax
        leave
        ret
```

A quick glance shows that the rtn3() function is nowhere to be found. With the -O3 command-line option enabled, GCC figured out that rtn3() simply returns a constant, so it propagates that constant return result everywhere you call rtn3(). In the case of the printf() function call, the combination of constant propagation and constant folding yielded a single constant, 5, that the code passes on to the printf() function.

As with constant folding, if your compiler doesn't support constant propagation you can simulate it manually, but only as a last resort. Again, finding a better compiler is almost always a better choice.

You can turn on the compiler's assembly language output to determine if your compiler support constant propagation. For example, here is Visual C++'s output (with the /O2 optimization level turned on):

```
#include <stdio.h>


int f(int a)
{
      return a + 1;
}

int main(int argc, char **argv)
{
      int i = 16 * 1024 * 1024;
      int j = f(i);
      printf( "%d\n", j);
}

// Assembly language output for the above code:

main    PROC                                          ; COMDAT

$LN6:
        sub     rsp, 40                               ; 00000028H

        mov     edx, 16777217                         ; 01000001H
        lea     rcx, OFFSET FLAT:??_C@_02DPKJAMEF@?$CFd?$AA@
        call    printf

        xor     eax, eax
        add     rsp, 40                               ; 00000028H
        ret     0
main    ENDP
```

As you can see, Visual C++ also eliminated the `f()` function as well as the `i` and `j` variables. It computed the function result (`i+1`) at compile time and substituted the constant `16777217` (`16*1024*1024 + 1`) for all the computations.

Here's an example using Java:

```
public class Welcome
{
      public static int f( int a ) { return a+1;}
      public static void main( String[] args )
      {
            int i = 16 * 1024 * 1024;
            int j = f(i);
            int k = i+1;
            System.out.println( j );
            System.out.println( k );
       }
}

// JBC emitted for this Java source code:



javap -c Welcome
Compiled from "Welcome.java"
```

```
public class Welcome extends java.lang.Object{
public Welcome();
  Code:
   0:   aload_0

        ; //Method java/lang/Object."<init>":()V
   1:   invokespecial   #1
   4:   return

public static int f(int);
  Code:
   0:   iload_0
   1:   iconst_1
   2:   iadd
   3:   ireturn

public static void main(java.lang.String[]);
  Code:
   0:   ldc #2; //int 16777216
   2:   istore_1
   3:   iload_1
   4:   invokestatic    #3; //Method f:(I)I
   7:   istore_2
   8:   iload_1
   9:   iconst_1
   10:  iadd
   11:  istore_3

        ; //Field java/lang/System.out:Ljava/io/PrintStream;
   12:  getstatic   #4
   15:  iload_2

        ; //Method java/io/PrintStream.println:(I)V
   16:  invokevirtual   #5

        ; //Field java/lang/System.out:Ljava/io/PrintStream;
   19:  getstatic   #4
   22:  iload_3

        ; //Method java/io/PrintStream.println:(I)V
   23:  invokevirtual   #5
   26:  return

}
```

A quick review of this Java bytecode shows that the Java compiler (`java version "1.6.0_65"`) does not support the constant propagation optimization. Not only did it not eliminate the `f()` function, but it also doesn't eliminate variables `i` and `j`, and it passes the value of `i` to function `f()` rather than passing the appropriate constant. One could argue that Java's bytecode interpretation dramatically affects performance, so a simple optimization such as constant propagation won't impact performance that much.

Here's the comparable program written in Swift, with the compiler's assembly output:

```
import Foundation

func f( _ a:Int ) -> Int
{
    return a + 1
}
let i:Int = 16*1024*1024
let j = f(i)
print( "i=\(i), j=\(j)" )

// Assembly output via the command:
// xcrun -sdk macosx swiftc -O -emit-assembly main.swift -o result.asm

    movq    $16777216, _$S6result1iSivp(%rip)
    movq    $16777217, _$S6result1jSivp(%rip)
    .
    .   // Lots of code that has nothing to do with the Swift source
    .
    movl    $16777216, %edi
    callq   _$Ss26_toStringReadOnlyPrintableySSxs06CustomB11ConvertibleRzlFSi_Tg5
    .
    .
    .
    movl    $16777217, %edi
    callq   _$Ss26_toStringReadOnlyPrintableySSxs06CustomB11ConvertibleRzlFSi_Tg5
```

The Swift compiler generates a tremendous amount of code in support of its runtime system, so you can hardly call Swift an *optimizing* compiler. That being said, the assembly code that it does generate demonstrates that Swift supports the constant propagation optimization. It eliminates the function `f()` and propagates the constants resulting from the calculations into the calls that print the values of `i` and `j`. It doesn't eliminate `i` and `j` (probably because of some consistency issues regarding the runtime system), but it does propagate the constants through the compiled code.

Given the excessive amount of code that the Swift compiler generates, it's questionable whether this optimization is worthwhile. However, even with all the extra code (too much to print here, so feel free to look at it yourself), the output still runs faster than interpreted Java code.

12.2.3 Dead Code Elimination

Dead code elimination is the removal of the object code associated with a particular source code statement if the program never again uses the result of that statement. Often, this is a result of a programming error. (After all, why would someone compute a value and not use it?) If a compiler encounters dead code in the source file, it may warn you to check the logic of your code. In some cases, however, earlier optimizations can produce dead code. For example, the constant propagation for the value variable in the earlier example could result in the statement `variable = 1234;` being dead. Compilers that support dead code elimination will quietly remove the object code for this statement from the object file.

As an example of dead code elimination, consider the following C program and its corresponding assembly code:

```
static int rtn3( void )
{
    return 3;
}

int main( void )
{
    int i = rtn3() + 2;

    // Note that this program
    // never again uses the value of i.

    return( 0 );
}
```

Here's the 32-bit 80x86 code GCC emits when supplied the `-O3` command-line option:

```
.file   "t.c"
        .text
        .p2align 2,,3
.globl main
        .type   main,@function
main:
        ; Build main's activation record:

        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp

        ; Notice that there is no
        ; assignment to i here.

        ; Return 0 as main's function result.

        xorl    %eax, %eax
        leave
        ret
```

Now consider the 80x86 output from GCC when optimization is not enabled:

```
.file   "t.c"
        .text
        .type   rtn3,@function
rtn3:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $3, %eax
        leave
        ret
.Lfe1:
        .size   rtn3,.Lfe1-rtn3
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp

        ; Note the call and computation:

        call    rtn3
        addl    $2, %eax
        movl    %eax, -4(%ebp)

        ; Return 0 as the function result.

        movl    $0, %eax
        leave
        ret
```

In fact, one of the main reasons that program examples throughout this book call a function like `printf()` to display various values is to explicitly use those values to prevent dead code elimination from erasing the code we're examining from the assembly output file. If you remove the final `printf()` from the C program in many of these examples, most of the assembly code will disappear because of dead code elimination.

Here's the output from the previous C++ code from Visual C++:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 19.00.24234.1

include listing.inc

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

PUBLIC  main
; Function compile flags: /Ogtpy
; File c:\users\rhyde\test\t\t\t.cpp
_TEXT   SEGMENT
main    PROC

        xor     eax, eax

        ret     0
```

```
main    ENDP
_TEXT   ENDS
; Function compile flags: /Ogtpy
; File c:\users\rhyde\test\t\t\t.cpp
_TEXT   SEGMENT
rtn3    PROC

        mov     eax, 3

        ret     0
rtn3    ENDP
_TEXT   ENDS
END
```

Unlike GCC, Visual C++ did not eliminate the `rtn3()` function. However, it did remove the assignment to `i`—and the call to `rtn3()`—in the main program.

Here's the equivalent Java program and the JBC output:

```
public class Welcome
{
    public static int rtn3() { return 3;}
    public static void main( String[] args )
    {
        int i = rtn3();
    }
}

// JBC output:

public class Welcome extends java.lang.Object{
public Welcome();
  Code:
   0:   aload_0

        ; //Method java/lang/Object."<init>":()V
   1:   invokespecial   #1
   4:   return

public static int rtn3();
  Code:
   0:   iconst_3
   1:   ireturn

public static void main(java.lang.String[]);
  Code:
   0:   invokestatic    #2; //Method rtn3:()I
   3:   istore_1
   4:   return

}
```

At first blush, it looks like Java does not support dead code elimination. However, the problem might be that our example code doesn't trigger this optimization in the compiler. Let's try something more obvious to the compiler:

```
public class Welcome
{
    public static int rtn3() { return 3;}
    public static void main( String[] args )
    {
        if( false )
        {   int i = rtn3();
        }
    }
}

// Here's the output bytecode:

Compiled from "Welcome.java"
public class Welcome extends java.lang.Object{
public Welcome();
  Code:
   0:   aload_0

        ; //Method java/lang/Object."<init>":()V
   1:   invokespecial   #1
   4:   return

public static int rtn3();
  Code:
   0:   iconst_3
   1:   ireturn

public static void main(java.lang.String[]);
  Code:
   0:   return

}
```

Now we've given the Java compiler something it can chew on. The main program eliminates the call to `rtn3()` and the assignment to `i`. The optimization isn't quite as smart as GCC's or Visual C++'s optimization, but (at least) for some cases, it works. Unfortunately, without constant propagation, Java misses many opportunities for dead code elimination.

Here's the equivalent Swift code for the earlier example:

```
import Foundation

func rtn3() -> Int
{
    return 3
}
let i:Int = rtn3()
```

```
// Assembly language output:

_main:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    $3, _$S6result1iSivp(%rip)
    xorl    %eax, %eax
    popq    %rbp
    retq

    .private_extern _$S6result4rtn3SiyF
    .globl  _$S6result4rtn3SiyF
    .p2align    4, 0x90
_$S6result4rtn3SiyF:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $3, %eax
    popq    %rbp
    retq
```

Note that Swift (at least for this example) does not support dead code elimination. However, let's try the same thing we did with Java. Consider the following code:

```
import Foundation

func rtn3() -> Int
{
    return 3
}
if false
{
    let i:Int = rtn3()
}

// Assembly output

_main:
    pushq   %rbp
    movq    %rsp, %rbp
    xorl    %eax, %eax
    popq    %rbp
    retq

    .private_extern _$S6result4rtn3SiyF
    .globl  _$S6result4rtn3SiyF
    .p2align    4, 0x90
_$S6result4rtn3SiyF:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $3, %eax
    popq    %rbp
    retq
```

Compiling this code produces a list of warnings about the dead code, but the output demonstrates that Swift does support dead code elimination. Furthermore, because Swift supports constant propagation as well, it won't miss as many opportunities for dead code elimination as Java (though Swift will need to mature a bit more before it catches up to GCC or Visual C++).

12.2.4 Common Subexpression Elimination

Often, a portion of some expressions—a *subexpression*—may appear elsewhere in the current function. If there are no changes to the values of the variables appearing in the subexpression, the program doesn't need to compute its value twice. Instead, it can save the subexpression's value on the first evaluation and then use that value everywhere the subexpression appears again. For example, consider the following Pascal code:

```
complex := ( a + b ) * ( c - d ) - ( e div f );
lessSo  := ( a + b ) - ( e div f );
quotient := e div f;
```

A decent compiler might translate these to the following sequence of three-address statements:

```
temp1 := a + b;
temp2 := c - d;
temp3 := e div f;
complex := temp1 * temp2;
complex := complex - temp3;
lessSo := temp1 - temp3;
quotient := temp3;
```

Although the former statements use the subexpression `(a + b)` twice and the subexpression `(e div f)` three times, the three-address code sequence computes these subexpressions only once and uses their values when the common subexpressions appear later.

As another example, consider the following C/C++ code:

```
#include <stdio.h>

static int i, j, k, m, n;
static int expr1, expr2, expr3;

extern int someFunc( void );

int main( void )
{
    // The following is a trick to
    // confuse the optimizer. When we call
    // an external function, the optimizer
    // knows nothing about the value this
    // function returns, so it cannot optimize
    // the values away. This is done to demonstrate
    // the optimizations that this example is
    // trying to show (that is, the compiler
    // would normally optimize away everything
    // and we wouldn't see the code the optimizer
    // would produce in a real-world example without
    // the following trick).

    i = someFunc();
    j = someFunc();
    k = someFunc();
    m = someFunc();
    n = someFunc();

    expr1 = (i+j) * (k*m+n);
    expr2 = (i+j);
    expr3 = (k*m+n);

    printf( "%d %d %d", expr1, expr2, expr3 );
    return( 0 );
}
```

Here's the 32-bit 80x86 assembly file that GCC generates (with the -O3 option) for the preceding C code:

```
.file   "t.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "%d %d %d"
        .text
        .p2align 2,,3
.globl main
        .type   main,@function
main:
        ; Build the activation record:

        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp

        ; Initialize i, j, k, m, and n:

        call    someFunc
        movl    %eax, i
        call    someFunc
        movl    %eax, j
        call    someFunc
        movl    %eax, k
        call    someFunc
        movl    %eax, m
        call    someFunc ;n's value is in EAX.

        ; Compute EDX = k*m+n
        ; and ECX = i+j
```

```
movl    m, %edx
movl    j, %ecx
imull   k, %edx
addl    %eax, %edx
addl    i, %ecx

; EDX is expr3, so push it
; on the stack for printf

pushl   %edx

; Save away n's value:

movl    %eax, n
movl    %ecx, %eax

; ECX is expr2, so push it onto
; the stack for printf:

pushl   %ecx

; expr1 is the product of the
; two subexpressions (currently
; held in EDX and EAX), so compute
; their product and push the result
; for printf.

imull   %edx, %eax
pushl   %eax

; Push the address of the format string
; for printf:

pushl   $.LC0

; Save the variable's values and then
; call printf to print the values

movl    %eax, expr1
movl    %ecx, expr2
movl    %edx, expr3
call    printf

; Return 0 as the main function's result:

xorl    %eax, %eax
leave
ret
```

Note how the compiler maintains the results of the common subexpressions in various registers (see the comments in the assembly output for details).

Here's the (64-bit) output from Visual C++:

```
_TEXT   SEGMENT
main    PROC

$LN4:
        sub     rsp, 40                                 ; 00000028H

        call    someFunc
        mov     DWORD PTR i, eax

        call    someFunc
        mov     DWORD PTR j, eax

        call    someFunc
        mov     DWORD PTR k, eax

        call    someFunc
        mov     DWORD PTR m, eax

        call    someFunc

        mov     r9d, DWORD PTR m

        lea     rcx, OFFSET FLAT:$SG7892
        imul    r9d, DWORD PTR k
        mov     r8d, DWORD PTR j
        add     r8d, DWORD PTR i
        mov     edx, r8d
        mov     DWORD PTR n, eax
        mov     DWORD PTR expr2, r8d
        add     r9d, eax
        imul    edx, r9d
        mov     DWORD PTR expr3, r9d
        mov     DWORD PTR expr1, edx
        call    printf

        xor     eax, eax

        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
_TEXT   ENDS
```

Because of the extra registers available on the x86-64, Visual C++ was able to keep all the temporaries in registers and did an even better job of reusing precomputed values for common subexpressions.

If the compiler you're using doesn't support common subexpression optimizations (you can determine this by examining the assembly output), chances are pretty good that its optimizer is subpar, and you should consider using a different compiler. However, in the meantime, you can always explicitly code this optimization yourself. Consider this version of the former C code, which manually computes common subexpressions:

```
#include <stdio.h>

static int i, j, k, m, n;
static int expr1, expr2, expr3;
static int ijExpr, kmnExpr;

extern int someFunc( void );

int main( void )
{
    // The following is a trick to
    // confuse the optimizer. By calling
    // an external function, the optimizer
    // knows nothing about the value this
    // function returns, so it cannot optimize
    // the values away because of constant propagation.

    i = someFunc();
    j = someFunc();
    k = someFunc();
    m = someFunc();
    n = someFunc();

    ijExpr = i+j;
    kmnExpr = (k*m+n);
    expr1 = ijExpr * kmnExpr;
    expr2 = ijExpr;
    expr3 = kmnExpr;

    printf( "%d %d %d", expr1, expr2, expr3 );
    return( 0 );
}
```

Of course, there was no reason to create the `ijExpr` and `kmnExpr` variables, as we could have simply used the *`expr2`* and *`expr3`* variables for this purpose. However, this code was written to make the changes to the original program as obvious as possible.

Here's the similar Java code:

```
public class Welcome
{
    public static int someFunc() { return 1;}
    public static void main( String[] args )
    {
        int i = someFunc();
        int j = someFunc();
        int k = someFunc();
        int m = someFunc();
        int n = someFunc();
        int expr1 = (i + j) * (k*m + n);
        int expr2 = (i + j);
        int expr3 = (k*m + n);
    }
}

// JBC output

public class Welcome extends java.lang.Object{
public Welcome();
  Code:
   0:   aload_0

        ; //Method java/lang/Object."<init>":()V
   1:   invokespecial   #1
   4:   return

public static int someFunc();
  Code:
   0:   iconst_1
   1:   ireturn

public static void main(java.lang.String[]);
  Code:
   0:   invokestatic    #2; //Method someFunc:()I
   3:   istore_1
   4:   invokestatic    #2; //Method someFunc:()I
   7:   istore_2
   8:   invokestatic    #2; //Method someFunc:()I
   11:  istore_3
   12:  invokestatic    #2; //Method someFunc:()I
   15:  istore  4
   17:  invokestatic    #2; //Method someFunc:()I
   20:  istore  5
; iexpr1 = (i + j) * (k*m + n);
   22:  iload_1
   23:  iload_2
   24:  iadd
   25:  iload_3
   26:  iload   4
   28:  imul
   29:  iload   5
   31:  iadd
   32:  imul
   33:  istore  6
; iexpr2 = (i+j)
   35:  iload_1
   36:  iload_2
   37:  iadd
   38:  istore  7
; iexpr3 = (k*m + n)
   40:  iload_3
   41:  iload   4
   43:  imul
   44:  iload   5
   46:  iadd
   47:  istore  8
   49:  return

}
```

Notice that Java does not optimize common subexpressions; instead, it recomputes the subexpressions each time it encounters them. Therefore, you should manually compute the values of common subexpressions when writing Java code.

Here's a variant of the current example in Swift (along with the assembly output):

```
import Foundation

func someFunc() -> UInt32
{
    return arc4random_uniform(100)
}
let i = someFunc()
let j = someFunc()
let k = someFunc()
let m = someFunc()
let n = someFunc()

let expr1 = (i+j) * (k*m+n)
let expr2 = (i+j)
let expr3 = (k*m+n)
print( "\(expr1), \(expr2), \(expr3)" )

// Assembly output for the above expressions:

; Code for the function calls:

    movl    $0x64, %edi
    callq   arc4random_uniform
    movl    %eax, %ebx  ; EBX = i
    movl    %ebx, _$S6result1is6UInt32Vvp(%rip)
    callq   _arc4random
    movl    %eax, %r12d ; R12d = j
    movl    %r12d, _$S6result1js6UInt32Vvp(%rip)
    callq   _arc4random
    movl    %eax, %r14d ; R14d = k
    movl    %r14d, _$S6result1ks6UInt32Vvp(%rip)
    callq   _arc4random
    movl    %eax, %r15d ; R15d = m
    movl    %r15d, _$S6result1ms6UInt32Vvp(%rip)
    callq   _arc4random
    movl    %eax, %esi  ; ESI = n
    movl    %esi, _$S6result1ns6UInt32Vvp(%rip)

; Code for the expressions:

    addl    %r12d, %ebx ; R12d = i + j (which is expr2)
    jb  LBB0_11         ; Branch if overflow occurs

    movl    %r14d, %eax ;
    mull    %r15d
    movl    %eax, %ecx  ; ECX = k*m
    jo  LBB0_12         ; Bail if overflow
    addl    %esi, %ecx  ; ECX = k*m + n (which is expr3)
    jb  LBB0_13         ; Bail if overflow

    movl    %ebx, %eax
    mull    %ecx        ; expr1 = (i+j) * (k*m+n)
    jo  LBB0_14         ; Bail if overflow
    movl    %eax, _$S6result5expr1s6UInt32Vvp(%rip)
    movl    %ebx, _$S6result5expr2s6UInt32Vvp(%rip)
    movl    %ecx, _$S6result5expr3s6UInt32Vvp(%rip)
```

If you carefully read through this code, you can see the Swift compiler properly optimizes away the common subexpressions and computes each subexpression only once.

12.2.5 Strength Reduction

Often, the CPU can directly compute some value using a different operator than the source code specifies, thereby replacing a more complex (or stronger) instruction with a simpler instruction. For example, a `shift` operation can implement multiplication or division by a constant that is a power of 2, and certain modulo (remainder) operations are possible using a bitwise `and` instruction (the `shift` and `and` instructions generally execute much faster than `multiply` and `divide` instructions). Most compiler optimizers are good at recognizing such operations and replacing the more expensive computation with a less expensive sequence of machine instructions. To see strength reduction in action, consider this C code and the 80x86 GCC output that follows it:

```
#include <stdio.h>

unsigned i, j, k, m, n;

extern unsigned someFunc( void );
extern void preventOptimization( unsigned arg1, ... );

int main( void )
{
    // The following is a trick to
    // confuse the optimizer. By calling
    // an external function, the optimizer
    // knows nothing about the value this
    // function returns, so it cannot optimize
    // the values away.

    i = someFunc();
    j = i * 2;
    k = i % 32;
    m = i / 4;
    n = i * 8;

    // The following call to "preventOptimization" is done
    // to trick the compiler into believing the above results
    // are used somewhere (GCC will eliminate all the code
    // above if you don't actually use the computed result,
    // and that would defeat the purpose of this example).

    preventOptimization( i,j,k,m,n);
    return( 0 );
}
```

Here's the resulting 80x86 code generated by GCC:

```
.file   "t.c"
        .text
        .p2align 2,,3
.globl main
        .type   main,@function
main:
        ; Build main's activation record:

        pushl   %ebp
        movl    %esp, %ebp
        pushl   %esi
        pushl   %ebx
        andl    $-16, %esp

        ; Get i's value into EAX:

        call    someFunc

        ; compute i*8 using the scaled-
        ; indexed addressing mode and
        ; the LEA instruction (leave
        ; n's value in EDX):

        leal    0(,%eax,8), %edx

        ; Adjust stack for call to
        ; preventOptimization:

        subl    $12, %esp

        movl    %eax, %ecx      ; ECX = i
        pushl   %edx            ; Push n for call
        movl    %eax, %ebx      ; Save i in k
        shrl    $2, %ecx        ; ECX = i/4 (m)
        pushl   %ecx            ; Push m for call

        andl    $31, %ebx       ; EBX = i % 32
        leal    (%eax,%eax), %esi ;j=i*2
        pushl   %ebx            ; Push k for call
        pushl   %esi            ; Push j for call
        pushl   %eax            ; Push i for call
        movl    %eax, i         ; Save values in memory
        movl    %esi, j         ; variables.
        movl    %ebx, k
        movl    %ecx, m
        movl    %edx, n
        call    preventOptimization

        ; Clean up the stack and return
        ; 0 as main's result:

        leal    -8(%ebp), %esp
        popl    %ebx
        xorl    %eax, %eax
        popl    %esi
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
        .comm   i,4,4
        .comm   j,4,4
        .comm   k,4,4
        .comm   m,4,4
        .comm   n,4,4
```

In this 80x86 code, note that GCC never emitted a multiplication or division instruction, even though the C code used these two operators extensively. GCC replaced each of these (expensive) operations with less expensive address calculations, shifts, and logical AND operations.

This C example declared its variables as `unsigned` rather than as `int`. There's a very good reason for this modification: strength reduction produces more efficient code for certain unsigned operands than it does for signed operands. This is a very important point: if you have a choice between using either signed or unsigned integer operands, always try to use unsigned values, because compilers can often generate better code when processing unsigned operands. To see the difference, here's the previous C code rewritten using signed integers, followed by GCC's 80x86 output:

```
#include <stdio.h>

int i, j, k, m, n;

extern int someFunc( void );
extern void preventOptimization( int arg1, ... );

int main( void )
{
    // The following is a trick to
    // confuse the optimizer. By calling
    // an external function, the optimizer
    // knows nothing about the value this
    // function returns, so it cannot optimize
    // the values away. That is, this prevents
    // constant propagation from computing all
    // the following values at compile time.

    i = someFunc();
    j = i * 2;
    k = i % 32;
    m = i / 4;
    n = i * 8;

    // The following call to "preventOptimization"
    // prevents dead code elimination of all the
    // preceding statements.

    preventOptimization( i,j,k,m,n);
    return( 0 );
}
```

Here is GCC's (32-bit) 80x86 assembly output for this C code:

```
.file   "t.c"
        .text
        .p2align 2,,3
        .globl main
        .type   main,@function
main:
        ; Build main's activation record:

        pushl   %ebp
        movl    %esp, %ebp
        pushl   %esi
        pushl   %ebx
        andl    $-16, %esp

        ; Call someFunc to get i's value:

        call    someFunc
        leal    (%eax,%eax), %esi ; j = i * 2
        testl   %eax, %eax        ; Test i's sign
        movl    %eax, %ecx
        movl    %eax, i
        movl    %esi, j
        js      .L4

; Here's the code we execute if i is non-negative:

.L2:
        andl    $-32, %eax       ; MOD operation
        movl    %ecx, %ebx
        subl    %eax, %ebx
        testl   %ecx, %ecx       ; Test i's sign
        movl    %ebx, k
        movl    %ecx, %eax
        js      .L5
.L3:
        subl    $12, %esp
        movl    %eax, %edx
        leal    0(,%ecx,8), %eax ; i*8
        pushl   %eax
        sarl    $2, %edx         ; Signed div by 4
        pushl   %edx
        pushl   %ebx
        pushl   %esi
        pushl   %ecx
        movl    %eax, n
        movl    %edx, m
        call    preventOptimization
        leal    -8(%ebp), %esp
        popl    %ebx
        xorl    %eax, %eax
        popl    %esi
        leave
        ret
        .p2align 2,,3

; For signed division by 4,
; using a sarl operation, we need
; to add 3 to i's value if i was
; negative.

.L5:
        leal    3(%ecx), %eax
        jmp     .L3
        .p2align 2,,3

; For signed % operation, we need to
; first add 31 to i's value if it was
; negative to begin with:

.L4:
        leal    31(%eax), %eax
        jmp     .L2
```

The difference in these two coding examples demonstrates why you should opt for unsigned integers (over signed integers) whenever you don't absolutely need to deal with negative numbers.

Attempting strength reduction manually is risky. While certain operations (like division) are almost always slower than others (like shifting to the right) on most CPUs, many strength reduction optimizations are not portable across CPUs. That is, substituting a left shift operation for multiplication may not always produce faster code when you compile for different CPUs. Some older C programs contain manual strength reductions that were originally added to improve performance. Today, those strength reductions can actually cause the programs to run slower than they should. Be very careful about incorporating strength reductions directly into your HLL code—this is one area where you should let the compiler do its job.

12.2.6 Induction

In many expressions, particularly those appearing within a loop, the value of one variable in the expression is completely dependent on some other variable. As an example, consider the following for loop in Pascal:

```
for i := 0 to 15 do begin

    j := i * 2;
    vector[ j ] := j;
    vector[ j+1 ] := j + 1;

end;
```

A compiler's optimizer may recognize that j is completely dependent on the value of i and rewrite this code as follows:

```
ij := 0;  {ij is the combination of i and j from
           the previous code}
while( ij < 32 ) do begin

    vector[ ij ] := ij;
    vector[ ij+1 ] := ij + 1;
    ij := ij + 2;

end;
```

This optimization saves some work in the loop (specifically, the computation of j := i * 2).

As another example, consider the following C code and the MASM output that Microsoft's Visual C++ compiler produces:

```
extern unsigned vector[32];

extern void someFunc( unsigned v[] );
extern void preventOptimization( int arg1, ... );

int main( void )
{

    unsigned i, j;

    //  "Initialize" vector (or, at least,
    //  make the compiler believe this is
    //  what's going on):

    someFunc( vector );

    // For loop to demonstrate induction:

    for( i=0; i<16; ++i )
    {
        j = i * 2;
        vector[ j ] = j;
        vector[ j+1 ] = j+1;
    }

    // The following prevents dead code elimination
    // of the former calculations:

    preventOptimization( vector[0], vector[15] );
    return( 0 );
}
```

Here's the MASM (32-bit 80x86) output from Visual C++:

```
_main   PROC

        push    OFFSET _vector
        call    _someFunc
        add     esp, 4
        xor     edx, edx

        xor     eax, eax
$LL4@main:

        lea     ecx, DWORD PTR [edx+1]      ; ECX = j+1
        mov     DWORD PTR _vector[eax], edx ; EDX = j
        mov     DWORD PTR _vector[eax+4], ecx

; Each time through the loop, bump j up by 2 (i*2)

        add     edx, 2

; Add 8 to index into vector, as we are filling two elements
; on each loop.

        add     eax, 8

; Repeat until we reach the end of the array.

        cmp     eax, 128                                ; 00000080H
        jb      SHORT $LL4@main

        push    DWORD PTR _vector+60
        push    DWORD PTR _vector
        call    _preventOptimization
        add     esp, 8

        xor     eax, eax

        ret     0
_main   ENDP
_TEXT   ENDS
```

As you can see in this MASM output, the Visual C++ compiler recognizes that `i` is not used in this loop. There are no calculations involving `i`, and it's completely optimized away. Furthermore, there's no `j = i * 2` computation. Instead, the compiler uses induction to determine that `j` increases by 2 on each iteration, and emits the code to do this rather than computing the value of `j` value from `i`. Finally, note that the compiler doesn't index into the vector array. Instead, it marches a pointer through the array on each iteration of the loop—once again using induction to produce a faster and shorter code sequence than you'd get without this optimization.

As for common subexpressions, you can manually incorporate induction optimization into your programs. The result is almost always harder to read and understand, but if your compiler's optimizer fails to produce good machine code in a section of your program, manual optimization is always an option.

Here's the Java variation of this example and the JBC output:

```
public class Welcome
{
    public static void main( String[] args )
    {
          int[] vector = new int[32];
          int j;
          for (int i = 0; i<16; ++i)
          {
            j = i * 2;
            vector[j] = j;
            vector[j + 1] = j + 1;
          }
    }
}

// JBC:

Compiled from "Welcome.java"
public class Welcome extends java.lang.Object{
public Welcome();
  Code:
   0:   aload_0

        ; //Method java/lang/Object."<init>":()V
   1:   invokespecial   #1
   4:   return

public static void main(java.lang.String[]);
  Code:
; Create vector array:

   0:   bipush  16
   2:   newarray int
   4:   astore_1

; i = 0   -- for( int i=0;...;...)
```

```
   5:   iconst_0
   6:   istore_3

; If i >= 16, exit loop  -- for(...;i<16;...)

   7:   iload_3
   8:   bipush  16
   10:  if_icmpge   35

; j = i * 2

   13:  iload_3
   14:  iconst_2
   15:  imul
   16:  istore_2

; vector[j] = j

   17:  aload_1
   18:  iload_2
   19:  iload_2
   20:  iastore

; vector[j+1] = j + 1

   21:  aload_1
   22:  iload_2
   23:  iconst_1
   24:  iadd
   25:  iload_2
   26:  iconst_1
   27:  iadd
   28:  iastore

; Next iteration of loop -- for(...;...; ++i )

   29:  iinc    3, 1
   32:  goto    7

; exit program here.

   35:  return

}
```

It's probably obvious that Java doesn't optimize this code at all. If you want better code, you'll have to manually optimize it:

```
for ( j = 0; j < 32; j = j + 2 )
{
    vector[j] = j;
    vector[j + 1] = j + 1;
}
```

```
  Code:
; Create array:

   0:   bipush  16
   2:   newarray int
   4:   astore_1

; for( int j = 0;...;...)

   5:   iconst_0
   6:   istore_2

; if j >= 32, bail -- for(...;j<32;...)

   7:   iload_2
   8:   bipush  32
   10:  if_icmpge   32

; vector[j] = j

   13:  aload_1
   14:  iload_2
   15:  iload_2
   16:  iastore

; vector[j + 1] = j + 1

   17:  aload_1
   18:  iload_2
   19:  iconst_1
   20:  iadd
   21:  iload_2
   22:  iconst_1
   23:  iadd
   24:  iastore

; j += 2  -- for(...;...; j += 2 )

   25:  iload_2
   26:  iconst_2
   27:  iadd
   28:  istore_2
   29:  goto    7

   32:  return
```

As you can see, Java isn't the best language choice if you're interested in producing optimized runtime code. Perhaps Java's authors felt that as a result of the interpreted bytecode execution, there was no real reason to try to optimize the compiler's output, or perhaps they felt that optimization was the JIT compiler's responsibility.

12.2.7 Loop Invariants

The optimizations shown so far have all been techniques a compiler can use to improve code that is already well written. Handling loop invariants, by contrast, is a compiler optimization for fixing bad code. A *loop invariant* is an expression that does not change on each iteration of some loop. The following Visual Basic code demonstrates a trivial loop-invariant calculation:

```
i = 5
for j = 1 to 10
    k = i*2
next j
```

The value of k does not change during the loop's execution. Once the loop completes execution, k's value is exactly the same as if the calculation of k had been moved before or after the loop. For example:

```
i = 5
k = i * 2
for j = 1 to 10
next j
rem At this point, k will contain the same
rem value as in the previous example
```

The difference between these two code fragments, of course, is that the second example computes the value `k = i * 2` only once rather than on each iteration of the loop.

Many compilers' optimizers will spot a loop-invariant calculation and use *code motion* to move it outside the loop. As an example of this operation, consider the following C program and its corresponding output:

```
extern unsigned someFunc( void );
extern void preventOptimization( unsigned arg1, ... );

int main( void )
{
    unsigned i, j, k, m;

    k = someFunc();
    m = k;
    for( i = 0; i < k; ++i )
    {
        j = k + 2;    // Loop-invariant calculation
        m += j + i;
    }
    preventOptimization( m, j, k, i );
    return( 0 );
}
```

Here's the 80x86 MASM code emitted by Visual C++:

```
_main   PROC NEAR ; COMDAT
; File t.c
; Line 5
        push    ecx
        push    esi
; Line 8
        call    _someFunc
; Line 10
        xor     ecx, ecx  ; i = 0
        test    eax, eax  ; see if k == 0
        mov     edx, eax  ; m = k
        jbe     SHORT $L108
        push    edi

; Line 12
; Compute j = k + 2, but only execute this
; once (code was moved out of the loop):

        lea     esi, DWORD PTR [eax+2] ; j = k + 2

; Here's the loop the above code was moved
; out of:

$L99:
; Line 13
        ; m(edi) = j(esi) + i(ecx)

        lea     edi, DWORD PTR [esi+ecx]
        add     edx, edi

        ; ++i
        inc     ecx

        ; While i < k, repeat:

        cmp     ecx, eax
        jb      SHORT $L99

        pop     edi
; Line 15
;
; This is the code after the loop body:

        push    ecx
        push    eax
        push    esi
        push    edx
        call    _preventOptimization
        add     esp, 16                             ; 00000010H
; Line 16
        xor     eax, eax
        pop     esi
```

```
; Line 17
        pop     ecx
        ret     0
$L108:
; Line 10
        mov     esi, DWORD PTR _j$[esp+8]
; Line 15
        push    ecx
        push    eax
        push    esi
        push    edx
        call    _preventOptimization
        add     esp, 16                                 ; 00000010H
; Line 16
        xor     eax, eax
        pop     esi
; Line 17
        pop     ecx
        ret     0
_main   ENDP
```

As you can see by reading the comments in the assembly code, the loop-invariant expression `j = k + 2` was moved out of the loop and executed prior to the start of the loop's code, saving some execution time on each iteration of the loop.

Unlike most optimizations, which you should leave up to the compiler if possible, you should move all loop-invariant calculations out of a loop unless there's a justifiable reason for leaving them there. Loop-invariant calculations raise questions for someone reading your code ("Isn't this supposed to change in the loop?"), because their presence actually makes the code harder to read and understand. If you want to leave the invariant code in the loop for some reason, be sure to comment your justification for anyone looking at your code later.

12.2.8 Optimizers and Programmers

HLL programmers fall into three groups based on their understanding of these compiler optimizations:

- The first group is unaware of how compiler optimizations work, and they write their code without considering the effect that their code organization will have on the optimizer.
- The second group understands how compiler optimizations work, so they write their code to be more readable. They assume that the optimizer will handle issues such as converting multiplication and division to shifts (where appropriate) and preprocessing constant expressions. This second group places a fair amount of faith in the compiler's ability to correctly optimize their code.

- The third group is also aware of the general types of optimizations that compilers can do, but they don't trust the compilers to do the optimization for them. Instead, they manually incorporate those optimizations into their code.

Interestingly enough, compiler optimizers are actually designed for the first group of programmers, those who are ignorant of how the compiler operates. Therefore, a good compiler will usually produce roughly the same quality of code for all three types of programmers (at least with respect to arithmetic expressions). This is particularly true when you compile the same program across different compilers. However, keep in mind that this assertion is valid only for compilers that have decent optimization capabilities. If you have to compile your code on a large number of compilers and you can't be confident that all of them have good optimizers, manual optimization may be one way to achieve consistently good performance across all compilers.

Of course, the real question is, "Which compilers are good, and which are not?" It would be nice to provide a table or chart in this book that describes the optimization capabilities of all the different compilers you might encounter, but unfortunately, the rankings change as compiler vendors improve their products, so anything printed here would rapidly become obsolete.[5] Fortunately, there are several websites that try to keep up-to-date comparisons of compilers.

12.3 Side Effects in Arithmetic Expressions

You'll definitely want to give a compiler some guidance with respect to side effects that may occur in an expression. If you don't understand how compilers deal with side effects in arithmetic expressions, you might write code that doesn't always produce correct results, particularly when moving source code between different compilers. Wanting to write the fastest or the smallest possible code is all well and good, but if it doesn't produce the correct answer any optimizations you make on the code are all for naught.

A *side effect* is any modification to the global state of a program outside the immediate result a piece of code is producing. The primary purpose of an arithmetic expression is to produce the expression's result. Any other change to the system's state in an expression is a side effect. The C, C++, C#, Java, Swift, and other C-based languages are especially guilty of allowing side effects in an arithmetic expression. For example, consider the following C code fragment:

```
i = i + *pi++ + (j = 2) * --k
```

5. Indeed, for the second edition of this book I had to replace many of the assembly output listings from the compilers.

This expression exhibits four separate side effects:

- The decrement of `k` at the end of the expression
- The assignment to `j` prior to using `j`'s value
- The increment of the pointer `pi` after dereferencing `pi`
- The assignment to `i`[6]

Although few non–C-based languages provide as many ways to create side effects in arithmetic expressions as C does, most languages do allow you to create side effects within an expression via a function call. Side effects in functions are useful, for example, when you need to return more than a single value as a function result in languages that don't directly support this capability. Consider the following Pascal code fragment:

```
var
   k:integer;
   m:integer;
   n:integer;

function hasSideEffect( i:integer; var j:integer ):integer;
begin

    k := k + 1;
    hasSideEffect := i + j;
    j := i;

end;
        .
        .
        .
    m := hasSideEffect( 5, n );
```

In this example, the call to the `hasSideEffect()` function produces two different side effects:

- The modification of the global variable `k`.
- The modification of the pass-by-reference parameter `j` (the actual parameter is `n` in this code fragment).

The real purpose of the function is to compute its return result. Any modification of global values or reference parameters constitutes a side effect of that function; hence, invoking that function within an expression produces side effects. Any language that allows you to modify global values (either directly or through parameters) from a function is capable of producing side effects within an expression; this concept is not limited to Pascal programs.

6. Generally, if we converted this expression to a stand-alone statement by placing a semicolon after it, we'd consider the assignment to `i` to be the purpose of the statement, not a side effect.

The problem with side effects in an expression is that most languages do not guarantee the order of evaluation of the components that make up an expression. Many novice programmers incorrectly assume that when they write an expression such as the following:

```
i := f(x) + g(x);
```

the compiler will emit code that first calls function `f()` and then calls function `g()`. Very few programming languages, however, require this order of execution. That is, some compilers will indeed call `f()`, then `g()`, and add their return results together. Other compilers, however, will call `g()` first, then `f()`, and compute the sum of the function return results. That is, the compiler could translate this expression into either of the following simplified code sequences before actually generating native machine code:

```
{ Conversion #1 for "i := f(x) + g(x);" }

    temp1 := f(x);
    temp2 := g(x);
    i := temp1 + temp2;

{ Conversion #2 for "i := f(x) + g(x);" }

    temp1 := g(x);
    temp2 := f(x);
    i := temp2 + temp1;
```

These two different function call sequences could produce completely different results if `f()` or `g()` produces a side effect. For example, if function `f()` modifies the value of the `x` parameter you pass to it, the preceding sequence could produce different results.

Note that issues such as precedence, associativity, and commutativity have no bearing on whether the compiler evaluates one subcomponent of an expression before another.

For example, consider the following arithmetic expression and several possible intermediate forms for the expression:

```
    j := f(x) - g(x) * h(x);

{ Conversion #1 for this expression: }

    temp1 := f(x);
    temp2 := g(x);
    temp3 := h(x);
    temp4 := temp2 * temp3
    j := temp1 - temp4;

{ Conversion #2 for this expression: }

    temp2 := g(x);
    temp3 := h(x);
    temp1 := f(x);
    temp4 := temp2 * temp3
    j := temp1 - temp4;

{ Conversion #3 for this expression: }

    temp3 := h(x);
    temp1 := f(x);
    temp2 := g(x);
    temp4 := temp2 * temp3
    j := temp1 - temp4;
```

Other combinations are also possible.

The specifications for most programming languages explicitly leave the order of evaluation undefined. This may seem somewhat bizarre, but there's a good reason for it: sometimes the compiler can produce better machine code by rearranging the order in which it evaluates certain subexpressions within an expression. Any attempt by the language designer to force a particular order of evaluation on a compiler's implementer, therefore, could limit the range of optimizations possible.

There are, of course, certain rules that most languages do enforce. Probably the most common rule is that all side effects within an expression will occur prior to the completion of that statement's execution. For example, if the function `f()` modifies the global variable `x`, then the following statements will always print the value of `x` after `f()` modifies it:

```
i := f(x);
writeln( "x=", x );
```

Another rule you can count on is that the assignment to a variable on the left-hand side of an assignment statement does not occur prior to the use of that same variable on the right-hand side of the expression. That is, the following code won't store the result of the expression into variable `n` until it uses the previous value of `n` within the expression:

```
n := f(x) + g(x) - n;
```

Because the order of the production of side effects within an expression is undefined in most languages, the result of the following code is generally undefined (in Pascal):

```
function incN:integer;
begin
    incN := n;
    n := n + 1;
end;
        .
        .
        .
    n := 2;
    writeln( incN + n*2 );
```

The compiler is free to call the `incN()` function first (so `n` will contain `3` prior to executing the subexpression `n * 2`), or it can compute `n * 2` first and then call the `incN()` function. As a result, one compilation of this statement could produce the output `8`, while a different compilation might produce `6`. In both cases, `n` would contain `3` after the `writeln` statement is executed, but the order of computation of the expression in the `writeln` statement could vary.

Don't make the mistake of thinking you can run some experiments to determine the order of evaluation. At the very best, such experiments will tell you only the order a particular compiler uses. A different compiler may well compute subexpressions in a different order. In fact, the same compiler might also compute the components of a subexpression differently based on the context of that subexpression. This means that a compiler might compute the result using one ordering at one point in the program and using a different ordering somewhere else in the same program. This is why it's dangerous to "determine" the ordering your particular compiler uses and rely on that ordering. Even if the compiler is consistent in the order it uses to compute side effects, the compiler vendor could change the ordering in a later version. If you must depend upon the order of evaluation, first break the expression down into a sequence of simpler statements whose computational order you can control. For example, if you really need to have your program call `f()` before `g()` in this statement:

```
i := f(x) + g(x);
```

then you should write the code this way:

```
temp1 := f(x);
temp2 := g(x);
i := temp1 + temp2;
```

If you must control the order of evaluation within an expression, take special care to ensure that all side effects are computed at the appropriate time. To do this, you need to learn about sequence points.

12.4 Containing Side Effects: Sequence Points

As noted earlier, most languages guarantee that the computation of side effects completes before certain points, known as *sequence points*, in your program's execution. For example, almost every language guarantees that all side effects will be computed by the time the statement containing the expression completes execution. The end of a statement is an example of a sequence point.

The C programming language provides several important sequence points within expressions, in addition to the semicolon at the end of a statement. C defines sequence points between each of the following operators:

```
expression1, expression2                   (comma operator in an expression)
expression1 && expression2                 (logical AND operator)
expression1 || expression2                 (logical OR operator)
expression1 ? expression2 : expression3    (conditional expression operator)
```

In these examples, C[7] guarantees that all side effects in *`expression1`* are completed before the computation of *`expression2`* or *`expression3`*. Note that for the conditional expression, C evaluates only one of *`expression2`* or *`expression3`* so the side effects of only one of these subexpressions ever occurs on a given execution of the conditional expression. Similarly, short-circuit evaluation may cause only *`expression1`* to evaluate in the `&&` and `||` operations. So, take care when using the last three forms.

To understand how side effects and sequence points can affect the operation of your program, consider the following example in C:

```
int array[6] = {0, 0, 0, 0, 0, 0};
int i;
    .
    .
    .
i = 0;
array[i] = i++;
```

Note that C does not define a sequence point across the assignment operator. Therefore, the language makes no guarantees about the value of the expression `i` it uses as an index. The compiler can choose to use the value of `i` before or after indexing into array. That the `++` operator is a post-increment operation implies only that `i++` returns the value of `i` prior to the increment; it doesn't guarantee that the compiler will use the pre-increment value of `i` anywhere else in the expression. The bottom line is that the last statement in this example could be semantically equivalent to either of the following statements:

```
      array[0] = i++;
-or-
      array[1] = i++;
```

The C language definition allows either form; it doesn't require the first form simply because the array index appears in the expression before the post-increment operator.

7. Modern C++ compilers generally provide the same sequence points as C, although the original C++ standard did not define any sequence points.

To control the assignment to array in this example, you have to ensure that no part of the expression depends upon the side effects of some other part of the expression. That is, you cannot both use the value of i at one point in the expression and apply the post-increment operator to i in another part of the expression, unless there is a sequence point between the two uses. Because there's no such sequence point in this statement, the result is undefined by the C language standard.

To guarantee that a side effect occurs at an appropriate point, you must have a sequence point between two subexpressions. For example, if you'd like to use the value of i prior to the increment as the index into the array, you could write the following code:

```
array [i] = i; //<-semicolon marks a sequence point.
++i;
```

To use the value of i after the increment operation as the array index, you could use code such as the following:

```
++i;               //<-semicolon marks a sequence point.
array[ i ] = i-1;
```

Note, by the way, that a decent compiler won't increment i and then compute i - 1. It will recognize the symmetry here, grab the value of i prior to the increment, and use that value as the index into array. This is an example of where someone who is familiar with typical compiler optimizations could take advantage of this knowledge to write code that is more readable. A programmer who inherently mistrusts compilers and their ability to optimize well might write code like this:

```
j=i++;            //<-semicolon marks a sequence point.
array[ i ] = j;
```

An important distinction is that a sequence point does not specify exactly when a computation will take place, only that it will happen before crossing the sequence point. The side effect could have been computed much earlier in the code, at any point between the previous sequence point and the current one. Another takeaway is that sequence points do not force the compiler to complete some computations between a pair of sequence points if that computation does not produce any side effects. Eliminating common subexpressions, for example, would be a far less useful optimization if the compiler could only use the result of common subexpression computations between sequence points. The compiler is free to compute the result of a subexpression as far ahead as necessary as long as that subexpression produces no side effects. Similarly, a compiler can compute the result of a subexpression as late as it cares to, as long as that result doesn't become part of a side effect.

Because statement endings (that is, semicolons) are a sequence point in most languages, one way to control the computation of side effects is to manually break a complex expression down into a sequence of three-address-like statements. For example, rather than relying on the Pascal compiler to translate an earlier example into three-address code using its own rules, you can explicitly write the code using whichever set of semantics you prefer:

```
{ Statement with an undefined result in Pascal }

    i := f(x) + g(x);

{ Corresponding statement with well-defined semantics }

    temp1 := f(x);
    temp2 := g(x);
    i := temp1 + temp2;

{ Another version, also with well-defined but different semantics }

    temp1 := g(x);
    temp2 := f(x);
    i := temp2 + temp1;
```

Again, operator precedence and associativity do not control when a computation takes place in an expression. Even though addition is left associative, the compiler may compute the value of the addition operator's right operand before it computes the value of the addition operator's left operand. Precedence and associativity control how the compiler arranges the computation to produce the result. They do not control when the program computes the subcomponents of the expression. As long as the final computation produces the results expected based on precedence and associativity, the compiler is free to compute the subcomponents in any order and at any time it pleases.

Thus far, this section has implied that a compiler always computes the value of an assignment statement and completes that assignment (and any other side effects) upon encountering the semicolon at the end of the statement. Strictly speaking, this isn't true. What many compilers do is ensure that all side effects occur between a sequence point and the next reference to the object changed by the side effect. For example, consider the following two statements:

```
j = i++;
k = m*n + 2;
```

Although the first statement in this code fragment has a side effect, some compilers might compute the value (or portions thereof) of the second statement before completing the execution of the first statement. Many compilers will rearrange various machine instructions to avoid data hazards and other execution dependencies in the code that might hamper performance (for details on data hazards, see *WGC1*). The semicolon sitting between these two statements does not guarantee that all computations for the first statement are complete before the CPU begins any new computation; it guarantees only that the program computes any side effects that precede the semicolon before executing any code that depends on them. Because the second statement does not depend upon the values of `j` or `i`, the compiler is free to start computing the second assignment prior to completing the first statement.

Sequence points act as barriers. A code sequence must complete its execution before any subsequent code affected by the side effect can execute. A compiler cannot compute the value of a side effect before executing all the code up to the previous sequence point in the program. Consider the following two code fragments:

```
// Code fragment #1:

    i = j + k;
    m = ++k;

// Code fragment #2:

    i = j + k;
    m = ++n;
```

In code fragment 1, the compiler must not rearrange the code so that it produces the side effect `++k` prior to using `k` in the previous statement. The end-of-statement sequence point guarantees that the first statement in this example uses the value of `k` prior to any side effects produced in subsequent statements. In code fragment 2, however, the result of the side effect that `++n` produces does not affect anything in the `i = j + k;` statement, so the compiler is free to move the `++n` operation into the code that computes `i`'s value if doing so is more convenient or efficient.

12.5 Avoiding Problems Caused by Side Effects

Because it's often difficult to see the impact of side effects in your code, it's a good idea to try to limit your program's exposure to problems with side effects. Of course, the best way to do this is to eliminate side effects altogether in your programs. Unfortunately, that isn't a realistic option. Many algorithms depend upon side effects for proper operation (functions returning multiple results via reference parameters or even global variables are good examples). You can, however, reduce unintended consequences of side effects by observing a few simple rules:

- Avoid placing side effects in Boolean expressions within program flow control statements such as `if`, `while`, and `do..until`.
- If a side effect exists on the right side of an assignment operator, try moving the side effect into its own statement before or after the assignment (depending on whether the assignment statement uses the value of the object before or after it applies the side effect).
- Avoid multiple assignments in the same statement; break them into separate statements.

- Avoid calling more than one function (that might produce a side effect) in the same expression.
- Avoid modifications to global objects (such as side effects) when writing functions.
- Always document side effects thoroughly. For functions, you should note the side effect in the function's documentation, as well as on every call to that function.

12.6 Forcing a Particular Order of Evaluation

As noted earlier, operator precedence and associativity do not control when a compiler may compute subexpressions. For example, if `X`, `Y`, and `Z` are each subexpressions (which could be anything from a single constant or variable reference to a complex expression in and of themselves), then an expression of the form `X / Y * Z` does not imply that the compiler computes the value for `X` before it computes the value for `Y` and `Z`. In fact, the compiler is free to compute the value for `Z` first, then `Y`, and finally `X`. Operator precedence and associativity require only that the compiler must compute the value of `X` and `Y` (in any order) before computing `X/Y`, and must compute the value of the subexpression `X/Y` before computing `(X / Y) * Z`. Of course, compilers can transform expressions via applicable algebraic transformations, but they're generally careful about doing so, because not all standard algebraic transformations apply in limited-precision arithmetic.

Although compilers can compute subexpressions in any order they choose (which is why side effects can create obscure problems), they usually avoid rearranging the order of actual computations. For example, mathematically, the following two expressions are equivalent following the standard rules of algebra (versus limited-precision computer arithmetic):

```
X / Y * Z
Z * X / Y
```

In standard mathematics, this identity exists because the multiplication operator is *commutative*; that is, $A \times B$ is equal to $B \times A$. Indeed, these two expressions will generally produce the same result as long as they are computed as follows:

```
(X / Y) * Z
Z * (X / Y)
```

The parentheses are used here not to show precedence, but to group calculations that the CPU must perform as a unit. That is, the statements are equivalent to:

```
A = X / Y;
B = Z
C = A * B
D = B * A
```

In most algebraic systems, `C` and `D` should have the same value. To understand why `C` and `D` may not be equivalent, consider what happens when `X`, `Y`, and `Z` are all integer objects with the values `5`, `2`, and `3`, respectively:

```
    X / Y * Z
=   5 / 2 * 3
=   2 * 3
=   6

    Z * X / Y
=   3 * 5 / 2
=   15 / 2
=   7
```

Again, this is why compilers are careful about algebraically rearranging expressions. Most programmers realize that `X * (Y / Z)` is not the same thing as `(X * Y) / Z`. Most compilers realize this too. In theory, a compiler should translate an expression of the form `X * Y / Z` as though it were `(X * Y) / Z`, because the multiplication and division operators have the same precedence and are left associative. However, good programmers never rely on the rules of associativity to guarantee this. Although most compilers will correctly translate this expression as intended, the next engineer who comes along might not realize what's going on. Therefore, explicitly including the parentheses to clarify the intended evaluation is a good idea. Better still, treat integer truncation as a side effect and break the expression down into its constituent computations (using three-address-like expressions) to ensure the proper order of evaluation.

Integer arithmetic obviously obeys its own rules, and those of real algebra don't always apply. However, don't assume that floating-point arithmetic doesn't suffer from the same set of problems. Any time you're doing limited-precision arithmetic involving the possibility of rounding, truncation, overflow, or underflow—as is the case with floating-point arithmetic—standard real-arithmetic algebraic transformations may not be legal. In other words, applying arbitrary real-arithmetic transformations to a floating-point expression can introduce inaccuracies in the computation. Therefore, a good compiler won't perform these types of transformations on real expressions. Unfortunately, some compilers do apply the rules of real arithmetic to floating-point operations. Most of the time, the results they produce are reasonably correct (within the limitations of the floating-point representation); in some special cases, however, they're particularly bad.

In general, if you must control the order of evaluation and when the program computes subcomponents of an expression, your only choice is to use assembly language. Subject to minor issues, such as out-of-order instruction execution, you can specify exactly when your software will compute various components of an expression when implementing the expression in assembly code. For very accurate computations, when the order of evaluation can affect the results you obtain, assembly language may be the safest approach. Although fewer programmers are capable of reading and understanding it, there's no question that it allows you to exactly specify the semantics of an arithmetic expression—what you read is what you get without any modification by the assembler. This simply isn't true for most HLL systems.

12.7 Short-Circuit Evaluation

For certain arithmetic and logical operators, if one component of the expression has a certain value, the value for the whole expression is automatically known regardless of the values of the expression's remaining components. A classic example is the multiplication operator. If you have an expression `A * B` and you know that either `A` or `B` is `0`, there's no need to compute the other component, because the result is already `0`. If the cost of computing the subexpressions is expensive relative to the cost of a comparison, then a program can save some time by testing the first component to determine if it needs to bother computing the second component. This optimization is known as *short-circuit evaluation* because the program skips over ("short-circuits" in electronics terminology) computing the remainder of the expression.

Although a couple of arithmetic operations could employ short-circuit evaluation, the cost of checking for the optimization is usually more expensive than just completing the computation. Multiplication, for example, could use short-circuit evaluation to avoid multiplication by zero, as just described. However, in real programs, multiplication by zero occurs so infrequently that the cost of the comparison against zero in all the other cases generally overwhelms any savings achieved by avoiding multiplication by zero. For this reason, you'll rarely see a language system that supports short-circuit evaluation for arithmetic operations.

12.7.1 Using Short-Circuit Evaluation with Boolean Expressions

One type of expression that *can* benefit from short-circuit evaluation is a Boolean/logical expression. Boolean expressions are good candidates for short-circuit evaluation for three reasons:

- Boolean expressions produce only two results, `true` and `false`; therefore, it's highly likely (50/50 chance, assuming random distribution) that one of the short-circuit "trigger" values will appear.
- Boolean expressions tend to be complex.
- Boolean expressions occur frequently in programs.

Because of these characteristics, you'll find that many compilers use short-circuit evaluation when processing Boolean expressions.

Consider the following two C statements:

```
A = B && C;
D = E || F;
```

Note that if B is false, then A will be false regardless of C's value. Similarly, if E is true, then D will be true regardless of F's value. We can, therefore, compute the values for A and D as follows:

```
A = B;
if( A )
{
    A = C;
}

D = E;
if( !D )
{
    D = F;
}
```

Now this might seem like a whole lot of extra work (it's certainly more typing!), but if C and F represent complex Boolean expressions, then this code sequence could possibly run much faster if B is usually false and E is usually true. Of course, if your compiler fully supports short-circuit evaluation, you'd never type this code; the compiler would generate the equivalent code for you.

By the way, the converse of short-circuit evaluation is *complete Boolean evaluation*. In complete Boolean evaluation, the compiler emits code that always computes each subcomponent of a Boolean expression. Some languages (such as C, C++, C#, Swift, and Java) specify the use of short-circuit evaluation. A few languages (such as Ada) let the programmer specify whether to use short-circuit or complete Boolean evaluation. Most languages (such as Pascal) don't define whether expressions will use short-circuit or complete Boolean evaluation—the language leaves the choice up to the implementer. Indeed, the same compiler could use complete Boolean evaluation for one instance of an expression and use short-circuit evaluation for another occurrence of that same expression in the same program. Unless you're using a language that strictly defines the type of Boolean evaluation, you'll have to check with your specific compiler's documentation to determine how it processes Boolean expressions. (Remember to avoid compiler-specific mechanisms if there's a chance you'll have to compile your code with a different compiler in the future.)

Look again at the expansions of the earlier Boolean expressions. It should be clear that the program won't evaluate C and F if A is false and D is true. Therefore, the left-hand side of a conjunction (&&) or disjunction (||) operator can act as a gate, preventing the execution of the right-hand side of the expression. This is an important point and, indeed, many algorithms depend on this property for correct operation. Consider the following (very common) C statement:

```
if( ptr != NULL && *ptr != '\0' )
{
    << process current character pointed at by ptr >>
}
```

This example could fail if it used complete Boolean evaluation. Consider the case where the ptr variable contains NULL. With short-circuit evaluation the program will not compute the subexpression *ptr != '\0'; because it realizes the result is always false. As a result, control immediately transfers to the first statement beyond the ending brace in this if statement. Consider, however, what would happen if this compiler utilized complete Boolean evaluation instead. After determining that ptr contains NULL, the program would still attempt to dereference ptr. Unfortunately, this attempt would probably produce a runtime error. Therefore, complete Boolean evaluation would cause this program to fail, even though it dutifully checks to make sure that access via pointer is legal.

Another semantic difference between complete and short-circuit Boolean evaluation has to do with side effects. In particular, if a subexpression does not execute because of short-circuit evaluation, then that subexpression doesn't produce any side effects. This behavior is incredibly useful but inherently dangerous. It is useful insofar as some algorithms absolutely depend upon short-circuit evaluation. It is dangerous because some algorithms also expect all the side effects to occur, even if the expression evaluates to false at some point. As an example, consider the following bizarre (but absolutely legal) C statement, which advances a "cursor" pointer to the next 8-byte boundary in a string or the end of the string (whichever comes first):

```
*++ptr && *++ptr && *++ptr && *++ptr && *++ptr && *++ptr && *++ptr && *++ptr;
```

This statement begins by incrementing a pointer and then fetching a byte from memory (pointed to by ptr). If the byte fetched was 0, execution of this expression/statement immediately stops, as the entire expression evaluates to false at that point. If the character fetched is not 0, the process repeats up to seven more times. At the end of this sequence, either ptr points at a 0 byte or it points 8 bytes beyond the original position. The trick here is that the expression immediately terminates upon reaching the end of the string rather than mindlessly skipping beyond that point.

Of course, there are complementary examples that demonstrate desirable behavior when side effects occur in Boolean expressions involving complete Boolean evaluation. The important thing to note is that no one scheme is correct or incorrect; it all depends on context. In different situations, a given algorithm may require the use of short-circuit Boolean evaluation or complete Boolean evaluation to produce correct results. If the definition of the language you're using doesn't explicitly specify which scheme to use, or you want to use the other one (such as complete Boolean evaluation in C), then you have to write your code such that it forces the evaluation scheme you prefer.

12.7.2 Forcing Short-Circuit or Complete Boolean Evaluation

Forcing complete Boolean evaluation in a language where short-circuit evaluation is used (or may be used) is relatively easy. All you have to do is break the expression into individual statements, place the result of each subexpression into a variable, and then apply the conjunction and disjunction operators to these temporary variables. For example, consider the following conversion:

```
// Complex expression:

if( (a < f(x)) && (b != g(y)) || predicate( a + b ))
{
    <<stmts to execute if this expression is true>>
}

// Translation to a form that uses complete Boolean evaluation:

temp1 = a < f(x);
temp2 = b != g(y);
temp3 = predicate( a + b );
if( temp1 && temp2 || temp3 )
{
    <<stmts to execute if this expression is true>>
}
```

The Boolean expression within the `if` statement still uses short-circuit evaluation. However, because this code evaluates the subexpressions prior to the `if` statement, this code ensures that all of the side effects produced by the `f()`, `g()`, and `predicate()` functions will occur.

Suppose you want to go the other way. That is, what if your language supports only complete Boolean evaluation (or doesn't specify the evaluation type), and you want to force short-circuit evaluation? This direction is a little more work than the converse, but it's still not difficult.

Consider the following Pascal code:[8]

```
if( ((a < f(x)) and (b <> g(y))) or predicate( a + b )) then begin

    <<stmts to execute if the expression is true>>

end; (*if*)
```

To force short-circuit Boolean evaluation, you need to test the value of the first subexpression, and, only if it evaluates to `true`, evaluate the second subexpression (and the conjunction of the two expressions). You can do this with the following code:

```
boolResult := a < f(x);
if( boolResult ) then
    boolResult := b <> g(y);

if( not boolResult ) then
    boolResult := predicate( a+b );
```

8. The standard definition for Pascal doesn't specify whether the compiler uses complete or short-circuit Boolean evaluation. Most Pascal compilers, however, implement complete Boolean evaluation.

```
if( boolResult ) then begin

    <<stmts to execute if the if's expression is true>>

end; (*if*)
```

This code simulates short-circuit evaluation by using `if` statements to block (or force) execution of the `g()` and `predicate()` functions based on the current state of the Boolean expression (kept in the `boolResult` variable).

Converting an expression to force short-circuit evaluation or complete Boolean evaluation looks as though it requires far more code than the original forms. If you're concerned about the efficiency of this translation, relax. Internally, the compiler translates those Boolean expressions to three-address code that is similar to the translation that you did manually.

12.7.3 Comparing Short-Circuit and Complete Boolean Evaluation Efficiency

While you might have inferred from the preceding discussion that complete Boolean evaluation and short-circuit evaluation have equivalent efficiencies, that's not the case. If you're processing complex Boolean expressions or the cost of some of your subexpressions is rather high, then short-circuit evaluation is generally faster than complete Boolean evaluation. As to which form produces less object code, they're roughly equivalent, and the exact difference will depend entirely upon the expression you're evaluating.

To understand the efficiency issues surrounding complete versus short-circuit Boolean evaluation, look at the following HLA code, which implements this Boolean expression using both forms:[9]

```
// Complex expression:

 //  if( (a < f(x)) && (b != g(y)) || predicate( a+b ))
 //  {
 //      <<stmts to execute if the if's expression is true>>
 //  }
 //
 // Translation to a form that uses complete
 //  Boolean evaluation:
 //
 //  temp1 = a < f(x);
 //  temp2 = b != g(y);
 //  temp3 = predicate( a + b );
 //  if( temp1 && temp2 || temp3 )
 //  {
 //      <<stmts to execute if the expression evaluates true>>
 //  }
 //
 //
```

9. HLA supports an `if` statement with short-circuit Boolean evaluation, but we won't use it here because the purpose of this exercise is to avoid the high-level abstractions of an `if` statement.

```
// Translation into 80x86 assembly language code,
// assuming all variables and return results are
// unsigned 32-bit integers:

    f(x);              // Assume f returns its result in EAX
    cmp( a, eax );     // Compare a with f(x)'s return result.
    setb( bl );        // bl = a < f(x)
    g(y);              // Assume g returns its result in EAX
    cmp( b, eax );     // Compare b with g(y)'s return result
    setne( bh );       // bh = b != g(y)
    mov( a, eax );     // Compute a + b to pass along to the
    add( b, eax );     // predicate function.
    predicate( eax );// al holds predicate's result (0/1)
    and( bh, bl );     // bl = temp1 && temp2
    or( bl, al );      // al = (temp1 && temp2) || temp3
    jz skipStmts;      // 0 if false, not 0 if true.

      <<stmts to execute if the condition is true>>

skipStmts:
```

Here's the same expression using short-circuit Boolean evaluation:

```
    //  if( (a < f(x)) && (b != g(y)) || predicate( a+b ))
    //  {
    //      <<stmts to execute if the if's expression evaluates true>>
    //  }

        f(x);
        cmp( a, eax );
        jnb TryOR;          // If a is not less than f(x), try the OR clause
        g(y);
        cmp( b, eax );
        jne DoStmts         // If b is not equal g(y) (and a < f(x)), then do the body.

TryOR:
        mov( a, eax );
        add( b, eax );
        predicate( eax );
        test( eax, eax );   // EAX = 0?
        jz SkipStmts;

DoStmts:
        <<stmts to execute if the condition is true>>
SkipStmts:
```

As you can see by simply counting statements, the version using short-circuit evaluation is slightly shorter (11 instructions versus 12). However, the short-circuit version will probably run much faster because half the time the code will evaluate only two of the three expressions. This code evaluates all three subexpressions only when the first subexpression, `a < f(x)`, evaluates to `true` and the second expression, `b != g(y)`, evaluates to `false`. If the outcomes of these Boolean expressions are equally probable, this code will test all three subexpressions 25 percent of the time. The remainder of the time it has to test only two subexpressions (50 percent of the time it will test `a < f(x)` and `predicate(a + b)`, 25 percent of the time it will test `a < f(x)` and `b != g(y)`, and the remaining 25 percent of the time it will need to test all three conditions).

The interesting thing to note about these two assembly language sequences is that complete Boolean evaluation tends to maintain the state of the expression (`true` or `false`) in an actual variable, whereas short-circuit evaluation maintains the current state of the expression by the program's position in the code. Take another look at the short-circuit example. Note that it does not maintain the Boolean results from each of the subexpressions anywhere other than the position in the code. For example, if you get to the `TryOR` label in this code, you know that the subexpression involving conjunction (logical AND) is `false`. Likewise, if the program executes the call to `g(y)`, you know that the first subexpression in the example, `a < f(x)`, has evaluated to `true`. When you make it to the `DoStmts` label, you know that the entire expression has evaluated to `true`.

If the execution time for the `f()`, `g()`, and `predicate()` functions is roughly the same in the current example, you can greatly improve the code's performance with a nearly trivial modification:

```
    //  if( predicate( a + b ) || (a < f(x)) && (b != g(y)))
    //  {
    //      <<stmts to execute if the expression evaluates true>>
    //  }

        mov( a, eax );
        add( b, eax );
        predicate( eax );
        test( eax, eax );   // EAX = true (not zero)?
        jnz DoStmts;

        f(x);
        cmp( a, eax );
        jnb SkipStmts;      // If a >= f(x), try the OR clause
        g(y);
        cmp( b, eax );
        je SkipStmts;       // If b != g(y) then do the body.

DoStmts:
        <<stmts to execute if the condition is true>>
SkipStmts:
```

Again, if you assume that the outcome of each subexpression is random and evenly distributed (that is, there is a 50/50 chance that each subexpression produces `true`), then this code will, on average, run about 50 percent faster than the previous version. Why? Moving the test for `predicate()` to the beginning of the code fragment means the code can now determine with one test whether it needs to execute the body. Because 50 percent of the time `predicate()` returns `true`, you can determine if you're going to execute the loop body with a single test about half the time. In the earlier example, it always took at least two tests to determine if we were going to execute the loop body.

The two assumptions here (that the Boolean expressions are equally likely to produce true or false and that the costs of computing each subexpression are equal) rarely hold in practice. However, this means that you have an even greater opportunity to optimize your code, not less. For example, if the cost of calling the predicate() function is high (relative to the computation of the remainder of the expression), then you'll want to arrange the expression so that it calls predicate() only when it absolutely must. Conversely, if the cost of calling predicate() is low compared to the cost of computing the other subexpressions, then you'll want to call it first. The situation for the f() and g() functions is similar. Because the logical AND operation is commutative, the following two expressions are semantically equivalent (in the absence of side effects):

```
a < f(x) && b != g(y)
b != g(y) && a < f(x)
```

When the compiler uses short-circuit evaluation, the first expression executes faster than the second if the cost of calling function f() is less than the cost of calling function g(). Conversely, if calling f() is more expensive than calling g(), then the second expression usually executes faster.

Another factor that affects the performance of short-circuit Boolean expression evaluation is the likelihood that a given Boolean expression will return the same value on each call. Consider the following two templates:

```
expr1 && expr2
expr3 || expr4
```

When working with conjunctions, try to place the expression that is more likely to return true on the right-hand side of the conjunction operator (&&). Remember, for the logical AND operation, if the first operand is false, a Boolean system employing short-circuit evaluation will not bother to evaluate the second operand. For performance reasons, you want to place the operand that is most likely to return false on the left-hand side of the expression. This will avoid the computation of the second operand more often than had you reversed the operands.

The situation is reversed for disjunction (||). In this case, you'd arrange your operands so that *expr3* is more likely to return true than *expr4*. By organizing your disjunction operations this way, you'll skip the execution of the right-hand expression more often than if you had swapped the operands.

You cannot arbitrarily reorder Boolean expression operands if those expressions produce side effects, because the proper computation of those side effects may depend upon the exact order of the subexpressions. Rearranging the subexpressions may cause a side effect to happen that wouldn't otherwise occur. Keep this in mind when you're trying to improve performance by rearranging operands in a Boolean expression.

12.8 The Relative Cost of Arithmetic Operations

Most algorithm analysis methodologies use a simplifying assumption that all operations take the same amount of time.[10] This assumption is rarely correct, because some arithmetic operations are two orders of magnitude slower than other computations. For example, a simple integer addition operation is often much faster than an integer multiplication. Similarly, integer operations are usually much faster than the corresponding floating-point operations. For algorithm analysis purposes, it may be okay to ignore the fact that one operation may be *n* times faster than some other operation. For someone interested in writing great code, however, knowing which operators are the most efficient is important, especially when you have the option of choosing among them.

Unfortunately, we can't create a table of operators that lists their relative speeds. The performance of a given arithmetic operator will vary by CPU. Even within the same CPU family, you see a wide variance in performance for the same arithmetic operation. For example, shift and rotate operations are relatively fast on a Pentium III (relative, say, to an addition operation). On a Pentium 4, however, they're considerably slower. These operations were faster on later Intel CPUs. So an operator such as the C/C++ << or >> can be fast or slow, relative to an addition operation, depending upon which CPU it executes.

That said, I can provide some general guidelines. For example, on most CPUs the addition operation is one of the most efficient arithmetic and logical operations around; few CPUs support faster arithmetic or logical operations than addition. Therefore, it's useful to group various operations into classes based on their performance relative to an operation like addition (see Table 12-1 for an example).

Table 12-1: Relative Performances of Arithmetic Operations (Guidelines)

Relative performance	Operations
Fastest	Integer addition, subtraction, negation, logical AND, logical OR, logical XOR, logical NOT, and comparisons
	Logical shifts
	Logical rotates
	Multiplication
	Division
	Floating-point comparisons and negation
	Floating-point addition and subtraction
	Floating-point multiplication
Slowest	Floating-point division

10. Actually, to be technically correct, these methodologies assume that different arithmetic operations vary by a constant amount and that they ignore constant multiplicative differences.

The estimates in Table 12-1 are not accurate for all CPUs, but they provide a "first approximation" from which you can work until you gain more experience with a particular processor. On many processors you'll find anywhere between two and three orders of magnitude difference in the performances between the fastest and slowest operations. In particular, division tends to be quite slow on most processors (floating-point division is even slower). Multiplication is usually slower than addition, but again, the exact variance differs greatly between processors.

If you absolutely need to do floating-point division, there's little you can do to improve your application's performance by using a different operation (although, in some cases, it is faster to multiply by the reciprocal). However, note that you can compute many integer arithmetic calculations using different algorithms. For example, a left shift is often less expensive than multiplication by 2. While most compilers automatically handle such "operator conversions" for you, compilers aren't omniscient and can't always figure out the best way to calculate some result. However, if you manually do the "operator conversion" yourself, you don't have to rely on the compiler to get this right for you.

12.9 For More Information

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* 2nd ed. Essex, UK: Pearson Education Limited, 1986.

Barrett, William, and John Couch. *Compiler Construction: Theory and Practice.* Chicago: SRA, 1986.

Fraser, Christopher, and David Hansen. *A Retargetable C Compiler: Design and Implementation.* Boston: Addison-Wesley Professional, 1995.

Duntemann, Jeff. *Assembly Language Step-by-Step.* 3rd ed. Indianapolis: Wiley, 2009.

Hyde, Randall. *The Art of Assembly Language.* 2nd ed. San Francisco: No Starch Press, 2010.

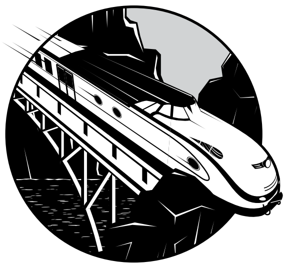
Louden, Kenneth C. *Compiler Construction: Principles and Practice.* Boston: Cengage, 1997.

Parsons, Thomas W. *Introduction to Compiler Construction.* New York: W. H. Freeman, 1992.

Willus.com. "Willus.com's 2011 Win32/64 C Compiler Benchmarks." Last updated April 8, 2012. *https://www.willus.com/ccomp_benchmark2.shtml.*

13

CONTROL STRUCTURES AND PROGRAMMATIC DECISIONS



Control structures are the bread and butter of high-level language (HLL) programming. The ability to make decisions based on the evaluation of stated conditions is fundamental to practically every kind of automation that computers provide. The translation of HLL control structures into machine code has, perhaps, the largest impact on program performance and size. As you'll see in this chapter, knowing which control structures to use in a given situation is the key to writing great code. In particular, this chapter describes the machine implementation of control structures related to decision making and unconditional flow, including:

- `if` statements
- `switch` or `case` statements
- `goto` and related statements

The following two chapters will expand this discussion to loop control structures and procedure/function calls and returns.

13.1 How Control Structures Affect a Program's Efficiency

A fair percentage of the machine instructions in a program control the execution path through that program. Because control transfer instructions often flush the instruction pipeline (see *WGC1*), they tend to be slower than instructions that perform simple calculations. To produce efficient programs, you should reduce the number of control transfer instructions or, if that's not possible, choose the fastest ones.

The exact set of instructions that CPUs use to control program flow varies across processors. Nevertheless, many CPUs (including the five families covered in this book) control program flow using the "compare-and-jump" paradigm. That is, after a compare or another instruction that modifies the CPU flags, a conditional jump instruction transfers control to another location based on the CPU flag settings. Some CPUs can do all this with a single instruction, while others require two, three, or more. Some CPUs allow you to compare two values for a large range of different conditions, whereas others allow only a few tests. Regardless of the mechanism, HLL statements that map to a given sequence on one CPU will map to a comparable sequence on a second CPU. Therefore, if you understand the basic conversion for one CPU, you'll have a good idea how the compiler works across all CPUs.

13.2 Introduction to Low-Level Control Structures

Most CPUs use a two-step process to make a programmatic decision. First, the program compares two values and saves the result of the comparison in a machine register or flag. Then the program tests that result and, based on what it learns, transfers control to one of two locations. With little more than this *compare and conditional branch* sequence, it is possible to synthesize most of the major HLL control structures.

Even within the compare and conditional branch paradigm, CPUs commonly implement conditional code sequences using two different approaches. One technique, especially common on stack-based architectures (such as the UCSD p-machine, Java Virtual Machine, and Microsoft CLR), is to have different forms of the compare instruction that test for specific conditions. For example, you might have *compare if equal*, *compare if not equal*, *compare if less than*, *compare if greater than*, and so on. The result of each is a Boolean value. Then a pair of conditional branch instructions, *branch if true* and *branch if false*, can test the result of the comparison and transfer control to the appropriate location. Some of these VMs might actually merge the compare and branch instructions into "compare and branch" instructions (one for each condition to test). Despite using fewer instructions, the end result is exactly the same.

The second, and historically more popular, approach is for the CPU's instruction set to contain a single comparison instruction that sets (or clears) several bits in the CPU's *program status* or *flags* register. Then the program uses one of several more specific conditional branch instructions to transfer control to some other location. These conditional branch instructions might have names such as *jump if equal*, *jump if not equal*, *jump if* *less than*, or *jump if greater than*. Because this "compare and jump" technique is the one the 80x86, ARM, and PowerPC use, that's also the approach this chapter's examples use; however, it's easy to convert them to the multiple comparisons/jump true/jump false paradigm.

The 32-bit variants of the ARM processor introduce a third technique: conditional execution. Most instructions (not just the branches) on the 32-bit ARM provide this option. For example, the `addeq` instruction adds two values if and only if the result of the previous comparison (or other operation) has set the zero flag. See "Conditional Suffixes for Instructions" in Appendix C online for more details.

Conditional branches are typically two-way branches. That is, they transfer control to one location in the program if the condition they're testing is `true` and to a different location if the condition is `false`. To reduce the size of the instruction, the conditional branches on most CPUs encode the address of only one of the two possible branch locations, and they use an implied address for the opposite condition. Specifically, most conditional branches transfer control to some target location if the condition is `true` and fall through to the next instruction if the condition is `false`. For example, consider the following 80x86 `je` (jump if equal) instruction sequence:

```
// Compare the value in EAX to the value in EBX

        cmp( eax, ebx );

// Branch to label EAXequalsEBX if EAX==EBX

        je EAXequalsEBX;

        mov( 4, ebx );      // Drop down here if EAX != EBX
            .
            .
            .
EAXequalsEBX:
```

This instruction sequence begins by comparing the value in the EAX register against the value in EBX (the `cmp` instruction); this sets the *condition-code bits* in the 80x86 EFLAGS register. In particular, this instruction sets the 80x86 zero flag to `1` if the value in EAX is equal to the value in EBX. The `je` instruction tests the zero flag to see if it is set, and if so, transfers control to the machine instruction immediately following the `EAXequalsEBX` label. If the value in EAX is not equal to EBX, then the `cmp` instruction clears the zero flag and the `je` instruction falls through to the `mov` instruction rather than transferring control to the destination label.

Certain machine instructions that access data can be smaller (and faster) if the memory location the machine instruction accesses is near the base address of the activation record containing that variable. This rule also applies to conditional jump instructions. The 80x86 provides two forms of the conditional jump instructions. One form is only 2 bytes long (1 byte for an opcode and 1 byte for a signed displacement in the range –128 through +127). The other form is 6 bytes long (2 bytes for the opcode and 4 bytes for a signed displacement in the range –2 billion through +2 billion). The displacement value specifies how far (in bytes) the program must jump to reach the target location. To transfer control to a nearby location, the program can use the short form of the branch. Because 80x86 instructions are between 1 and 15 bytes long (typically around 3 or 4 bytes long), the short forms of the conditional jump instructions can usually skip over about 32 to 40 machine instructions. Once the target location is out of the ±127-byte range, the 6-byte version of these conditional jump instructions extends the range to ±2 billion bytes around the current instruction. If you're interested in writing the most efficient code, then, you'll want to use the 2-byte form as often as possible.

Branching is an expensive operation in a modern (pipelined) CPU because a branch may require the CPU to flush the pipeline and reload it (see *WGC1* for more details). Conditional branches incur this cost only if the branch is taken; if the conditional branch instruction falls through to the next instruction, the CPU will continue to use the instructions found in the pipeline without flushing them. Therefore, on many systems the *branch that falls through to the next instruction is faster than the branch that is taken*. Note, however, that some CPUs (like the 80x86, PowerPC, and ARM) support a *branch prediction* feature that tells the CPU to begin fetching instructions for the pipeline from the branch's target location rather than from the instructions that immediately follow the conditional jump. Unfortunately, branch prediction algorithms vary from processor to processor (even within the 80x86 CPU family), so it's difficult to predict, in general, how branch prediction will affect your HLL code. It's probably safest to assume, unless you're writing code for a specific processor, that falling through to the next instruction is more efficient than taking the jump.

Although the compare and conditional branch paradigm is the most common control structure found in machine code programs, there are other ways to transfer control to another location in memory based on some computed result. Without question, the indirect jump (especially via a table of addresses) is the most common alternative form. Consider the following 32-bit 80x86 `jmp` instruction:

```
readonly
    jmpTable: dword[4] := [&label1, &label2, &label3, &label4];
            .
            .
            .
        jmp( jmpTable[ ebx*4 ] );
```

This `jmp` instruction fetches the double-word value at the index specified by the value in EBX in the `jmpTable` array. That is, the instruction transfers control to one of four different locations based upon the value (`0..3`) in EBX. For example, if EBX contains `0`, then the `jmp` instruction fetches the double word at index `0` in `jmpTable` (the address of the instruction prefixed by `label1`). Likewise, if EBX contains `2`, then this `jmp` instruction fetches the third double word from this table (the address of `label3` in the program). This is roughly equivalent to, but usually shorter than, the following sequence of instructions:

```
cmp( ebx, 0 );
je label1;
cmp( ebx, 1 );
je label2;
cmp( ebx, 2 );
je label3;
cmp( ebx, 3 );
je label4;

// Results are undefined if EBX <> 0, 1, 2, or 3
```

A few other conditional control transfer mechanisms are available on various CPUs, but these two mechanisms (compare and conditional branch and indirect jump) are the ones most HLL compilers use to implement standard control structures in the HLL.

13.3 The goto Statement

The `goto` statement is, perhaps, the most fundamental low-level control structure. Since the wave of "structured programming" in the late 1960s and 1970s, its use in HLL code has diminished. Indeed, some modern high-level programming languages (for example, Java and Swift) don't even provide an unstructured `goto` statement. Even in those languages where one is available, programming style guidelines usually restrict its use to special circumstances. Combined with the fact that student programmers have been religiously taught to avoid them in their programs since the mid 1970s, it's now rare to find many `goto` statements in a modern program. From a readability point of view, this is a good thing (check out some 1960s-era FORTRAN programs to get an idea of how hard to read code can be when it's peppered with `goto` statements). Nevertheless, some programmers believe that they can achieve higher efficiency by using `goto` statements in their code. While this is sometimes true, the gains are rarely worth the loss of readability that ultimately occurs.

One of the big efficiency arguments for `goto` is that it helps avoid duplicate code. Consider the following simple C/C++ example:

```
if( a == b || c < d )
{
    << execute some number of statements >>

    if( x == y )
    {
        << execute some statements if x == y >>
    }
    else
    {
        << execute some statements if x != y >>
    }
}
else
{
    << execute the same sequence of statements
       that the code executes if x!= y in the
       previous else section >>
}
```

Programmers looking for ways to make their programs more efficient will immediately notice all the duplicated code and might be tempted to rewrite the example as follows:

```
if( a == b || c < d )
{
    << execute some number of statements >>

    if( x != y ) goto DuplicatedCode;

    << execute some statements if x == y >>
}
else
{
DuplicatedCode:
    << execute the same sequence of statements
       if x != y or the original
       Boolean expression is false >>
}
```

There are, of course, several software engineering problems with this code, including the fact that it is a bit harder to read, modify, and maintain than the original example. (You *could* argue that it's actually a little easier to maintain, because you no longer have duplicated code and you only have to fix defects in the common code at one spot.) However, there's no denying that there's less code in this example. Or is there?

The optimizers in many modern compilers actually look for code sequences like the first example and generate code that's identical to what you'd expect for the second example. Therefore, a *good* compiler avoids generating duplicate machine code even when the source file contains duplication, as in the first example.

Consider the following C/C++ example:

```
#include <stdio.h>

static int a;
static int b;

extern int x;
extern int y;
extern int f( int );
extern int g( int );
```

```
int main( void )
{
    if( a==f(x))
    {
        if( b==g(y))
        {
            a=0;
        }
        else
        {
            printf( "%d %d\n", a, b );
            a=1;
            b=0;
        }
    }
    else
    {
        printf( "%d %d\n", a, b );
        a=1;
        b=0;
    }

    return( 0 );
}
```

Here's the compilation of the if sequence to PowerPC code by GCC:

```
        ; f(x):

        lwz r3,0(r9)
        bl L_f$stub

        ; Compute a==f(x), jump to L2 if false

        lwz r4,0(r30)
        cmpw cr0,r4,r3
        bne+ cr0,L2

        ; g(y):

        addis r9,r31,ha16(L_y$non_lazy_ptr-L1$pb)
        addis r29,r31,ha16(_b-L1$pb)
        lwz r9,lo16(L_y$non_lazy_ptr-L1$pb)(r9)
        la r29,lo16(_b-L1$pb)(r29)
        lwz r3,0(r9)
        bl L_g$stub

        ; Compute b==g(y), jump to L3 if false:

        lwz r5,0(r29)
        cmpw cr0,r5,r3
        bne- cr0,L3

        ; a=0
        li r0,0
        stw r0,0(r30)
        b L5

        ; Set up a and b parameters if
        ; a==f(x) but b!=g(y):

L3:
        lwz r4,0(r30)
        addis r3,r31,ha16(LC0-L1$pb)
        b L6

        ; Set up parameters if a!=f(x):
L2:
        addis r29,r31,ha16(_b-L1$pb)
        addis r3,r31,ha16(LC0-L1$pb)
        la r29,lo16(_b-L1$pb)(r29)
        lwz r5,0(r29)

        ; Common code shared by both
        ; ELSE sections:
L6:
        la r3,lo16(LC0-L1$pb)(r3) ; Call printf
        bl L_printf$stub
        li r9,1               ; a=1
        li r0,0               ; b=0
        stw r9,0(r30)         ; Store a
        stw r0,0(r29)         ; Store b
L5:
```

Of course, not every compiler has an optimizer that will recognize the duplicated code. So, if you want to write a program that compiles to efficient machine code regardless of the compiler, you might be tempted to use the version of the code that employs the `goto` statement. Indeed, you could make a strong software engineering argument that having duplicate code in a source file makes the program harder to read and harder to maintain. (If you fix a defect in one copy of the code, chances are that you'll forget to correct the defect in the other copies of the code.) While this is definitely true, if you make changes to the code at the target label, it's not immediately obvious that the change is appropriate for each and every section of code that jumps to the target label. And it's not immediately obvious how many different `goto` statements transfer control to the same target label when you're reading through the source code.

The traditional software engineering approach is to put the common code into a procedure or function and simply call that function. However, the overhead of a function call and return can be rather large (especially if there isn't much duplicated code), so from a performance point of view, that approach may not be satisfactory. For short sequences of common code, creating a macro or an inline function is probably the best solution. To complicate the issue, you might need a change that affects only one instance of the duplicated code (that is, it would no longer be a duplicate).

The bottom line is that using a goto statement to gain efficiency in this manner should be your last resort.

Another common use for goto statements is for exceptional conditions. When you find yourself nested deeply in several statements and you encounter a situation where you need to exit all those statements, the common consensus is that a goto is acceptable if restructuring the code wouldn't make it more readable. However, jumps out of nested blocks may thwart the optimizer's ability to generate decent code for the entire procedure or function. The use of the goto statement may save a few bytes or processor cycles in the code it immediately affects, but it could have detrimental effects on the rest of the function, resulting in less efficient code overall. So, take care when inserting goto statements into your code—they can make your source code harder to read, and might wind up making it less efficient as well.

For what it's worth, there's a programming trick you can use to solve the original problem. Consider the following modification to the code:

```
switch( a == b || c < d )
{
    case 1:
        << execute some number of statements >>

        if( x == y )
        {
            << execute some statements if x == y >>
            break;
        }
        // Fall through if x != y

    case 0:

        << execute some statements if x!= y or
            if !( a == b || c < d )  >>

}
```

Of course, this is tricky code, and tricky code isn't usually great code. However, it does have the benefit of avoiding duplication of source code in your program.

13.3.1 Restricted Forms of the goto Statement

In an effort to support structured goto-less programming, many programming languages have added restricted forms of the goto statement that allow a programmer to immediately exit a control structure such as a loop or a procedure/function. Typical statements include break and exit, which jump out of an enclosing loop; continue, cycle, and next, which restart an enclosing loop; and return and exit, which immediately return from an enclosing procedure or function. These statements are more structured than a standard goto because the programmer doesn't choose the destination; instead, control transfers to a fixed location based upon whatever control statement (or function or procedure) encloses the statement.

Almost every one of these statements compiles into a single `jmp` instruction. Those that jump out of a loop (such as `break`) compile into a single `jmp` instruction that transfers control to the first statement beyond the bottom of the loop. Those that restart a loop (for example, `continue`, `next`, or `cycle`) compile into a single `jmp` instruction that transfers control to the loop termination test (in the case of while or `repeat..until/do..while`) or to the top of the loop (in the case of most other loops).

However, just because these statements typically compile to a single `jmp` instruction doesn't mean they're efficient to use. Even ignoring the fact that a `jmp` can be somewhat expensive (because it forces the CPU to flush the instruction pipeline), statements that branch out of a loop can have a serious impact on the compiler's optimizer, dramatically reducing the opportunity to generate high-quality code. Therefore, you should attempt to use these statements as sparingly as possible.

13.4 The if Statement

Perhaps the most basic high-level control structure is the `if` statement. Indeed, with nothing more than an `if` and a `goto` statement, you can (semantically) implement all other control structures.[1] We'll revisit this point when discussing other control structures, but for now we'll look at how a typical compiler converts an `if` statement into machine code.

To implement a simple `if` statement that compares two values and executes the body if the condition is `true`, you can use a single compare and conditional branch instruction. Consider the following Pascal `if` statement:

```
if( EAX = EBX ) then begin

    writeln( 'EAX is equal to EBX' );
    i := i + 1;

end;
```

Here's the conversion to 80x86/HLA code:

```
    cmp( EAX, EBX );
    jne skipIfBody;
    stdout.put( "EAX is equal to EBX", nl );
    inc( i );
skipIfBody:
```

In the Pascal source code, the body of the `if` statement executes if the value of EAX is equal to EBX. In the resulting assembly code, the program compares EAX with EBX and then, if EAX does not equal EBX, branches over the statements that correspond to the `if` statement's body. This is the

1. Doing so isn't a good idea for reasons of maintainability, but it's certainly possible.

"boilerplate" conversion of an HLL if statement into machine code: test some condition and, if it's false, branch over the if statement's body.

The implementation of an if..then..else statement is only slightly more complicated than the basic if statement. An if..then..else statement typically employs syntax and semantics such as the following:

```
if( some_boolean_expression ) then

    << Statements to execute if the expression is true >>

else

    << Statements to execute if the expression is false >>

endif
```

Implementing this code sequence in machine code requires only a single machine instruction beyond what a simple if statement requires. Consider this example C/C++ code:

```
if( EAX == EBX )
{
    printf( "EAX is equal to EBX\n" );
    ++i;
}
else
{
    printf( "EAX is not equal to EBX\n" );
}
```

Here is the conversion to 80x86 assembly language code:

```
    cmp( EAX, EBX );        // See if EAX == EBX
    jne doElse;             // Branch around "then" code
    stdout.put( "EAX is equal to EBX", nl );
    inc( i );
    jmp skipElseBody;        // Skip over "else" section.

// if they are not equal.

doElse:
    stdout.put( "EAX is not equal to EBX", nl );

skipElseBody:
```

There are two things to note about this code. First, if the condition evaluates to false, the code transfers to the first statement of the else block rather than the first statement following the (entire) if statement. The second thing to note is the jmp instruction at the end of the true clause skips the else block.

Some languages, including HLA, support an elseif clause in their if statement to evaluate a second condition if the first one fails. This is a straightforward extension of the code generation of the `if` statement I've shown. Consider the following HLA `if..elseif..else..endif` statements:

```
if( EAX = EBX ) then

    stdout.put( "EAX is equal to EBX" nl );
    inc( i );

elseif( EAX = ECX ) then

    stdout.put( "EAX is equal to ECX" nl );

else

    stdout.put( "EAX is not equal to EBX or ECX" nl);

endif;
```

And here's the conversion to pure 80x86/HLA assembly language code:

```
// Test to see if EAX = EBX

    cmp( eax, ebx );
    jne tryElseif;    // Skip "then" section if equal

    // Start of the "then" section

    stdout.put( "EAX is equal to EBX", nl );
    inc( i );
    jmp skipElseBody  // End of "then" section, skip
                      // over the elseif clause.
tryElseif:
    cmp( eax, ecx );  // ELSEIF test for EAX = ECX
    jne doElse;       // Skip "then" clause if not equal

    // elseif "then" clause

    stdout.put( "EAX is equal to ECX", nl );
    jmp skipElseBody; // Skip over the "else" section

doElse: // else clause begins here
    stdout.put( "EAX is not equal to EBX or ECX", nl );

skipElseBody:
```

The translation of the `elseif` clause is very straightforward; the machine code for it is identical to an `if` statement. What's noteworthy here is how the compiler emits a `jmp` instruction at the end of the `if..then` clause to skip around the Boolean test emitted for the `elseif` clause.

13.4.1 Improving the Efficiency of Certain if/else Statements

From an efficiency point of view, it's important to note that there's no path through the `if..else` statement that doesn't involve a transfer of control (unlike the simple `if` statement, which simply falls through if the conditional expression is `true`). As this chapter has pointed out, branches are bad because they often flush the CPU's instruction pipeline, which takes several CPU cycles to refill. If both outcomes of the Boolean expression (`true` and `false`) are equally likely, there's little you can do to improve the code's performance by rearranging the `if..else` statement. For most `if` statements, however, one outcome is often more likely—perhaps much more likely—than the other. Assembly coders who understand the likelihood of one comparison over another will often encode their `if..else` statements as follows:

```
// if( eax == ebx ) then
//    //<likely case>
//    stdout.put( "EAX is equal to EBX", nl );
// else
//    // unlikely case
//    stdout.put( "EAX is not equal to EBX" nl );
// endif;

    cmp( EAX, EBX );
    jne goDoElse;
    stdout.put( "EAX is equal to EBX", nl );
backFromElse:
        .
        .
        .
// Somewhere else in the code (not in the direct path of the above):

goDoElse:
    stdout.put( "EAX is not equal to EBX", nl );
    jmp backFromElse
```

Note that in the most common case (where the expression evaluates to `true`), the code falls through to the `then` section, which then falls straight through to the code that follows the entire `if` statement. Therefore, if the Boolean expression (`eax == ebx`) is `true` most of the time, this code executes straight through without any branches. In the rare case, when EAX does not equal EBX, the program actually has to execute two branches: one to transfer control to the section of code that handles the `else` clause, and one to return control to the first statement following the `if`. As long as this occurs less than half of the time, the software sees an overall performance boost. You can achieve this same result in an HLL such as C using `goto` statements. For example:

```
if( eax != ebx ) goto doElseStuff;

    // << body of the if statement goes here>>
    // (statements between then and else)
```

```
endOfIF:
// << statements following the if..endif statement >>
    .
    .
    .
// Somewhere outside the direct execution path of the above

doElseStuff:
    << Code to do if the expression is false >>
    goto endOfIF;
```

Of course, the drawback to this scheme is that it produces *spaghetti code* that becomes unreadable once you add more than a few of these kludges. Assembly language programmers get away with this because most assembly language code is, by definition, spaghetti code.[2] For HLL code, however, this programming style is generally unacceptable, and you should use it only when necessary. (See "The goto Statement" on page 455.)

The following generic if statement is common in programs written in HLLs such as C:

```
if( eax == ebx )
{
    // Set i to some value along this execution path.

    i = j+5;
}
else
{
    // Set i to a different value along this path

    i = 0;
}
```

Here's the conversion of this C code into 80x86/HLA assembly code:

```
        cmp( eax, ebx );
        jne doElse;
        mov( j, edx );
        add( 5, edx );
        mov( edx, i );
        jmp ifDone;

doElse:
        mov( 0, i );
ifDone:
```

2. Although it is quite easy to write structured code with an assembler such as HLA.

As you've seen in previous examples, the if..then..else statement conversion to assembly language requires two control transfer instructions:

- The jne instruction that tests the comparison between EAX and EBX
- The unconditional jmp instruction that skips over the else section of the if statement

Regardless of which path the program takes (through the then or the else section), the CPU executes a slow branch instruction that winds up flushing the instruction pipeline. Consider the following code, which does not have this problem:

```
i = 0;
if( eax == ebx )
{
    i = j + 5;
}
```

Here is its conversion to pure 80x86/HLA assembly code:

```
        mov( 0, i );
        cmp( eax, ebx );
        jne skipIf;
        mov( j, edx );
        add( 5, edx );
        mov( edx, i );
skipIf:
```

As you can see, if the expression evaluates to true, the CPU executes no control transfer statements at all. Yes, the CPU executes an extra mov instruction whose result is immediately overwritten (so the execution of the first mov instruction is wasted); however, the execution of this extra mov instruction happens much more rapidly than the execution of the jmp instruction. This trick is a prime example of why it's a good idea to know some assembly language code (and know how compilers generate machine code from high-level code). It's not at all obvious that the second sequence is better than the first. Beginning programmers, in fact, would probably believe it to be inferior because the program "wastes" an assignment to i when the expression evaluates to true (and no such assignment is made in the first version). This is one reason why this chapter exists—to make sure you understand the costs associated with using high-level control structures.

13.4.2 Forcing Complete Boolean Evaluation in an if Statement

Because complete Boolean evaluation and short-circuit Boolean evaluation can produce different results (see "Short-Circuit Evaluation" on page 441), there are times when you'll need to force your code to use one form or the other when computing the result of a Boolean expression.

The general way to force complete Boolean evaluation is to evaluate each subcomponent of the expression and store the subresult into temporary variables. Then you can combine the temporary results after their computation to produce the complete result. For example, consider the following Pascal code fragment:

```
if( (i < g(y)) and (k > f(x)) ) then begin

    i := 0;

end;
```

Because Pascal doesn't guarantee complete Boolean evaluation, function `f()` might not be called in this expression—if `i` is less than `g(y)`—and thus any side effects produced by the call to `f()` might not occur. (See "Side Effects in Arithmetic Expressions" on page 430.) If the logic of the application depends on any side effects produced by the calls to `f()` and `g()`, then you must ensure that the application calls both functions. Note that simply swapping the two subexpressions around the AND operator is insufficient to solve this problem; with that change, the application might not call `g()`.

One way to solve this problem is to compute the Boolean results of the two subexpressions using separate assignment statements and then compute the logical AND of the two results within the `if` expression:

```
lexpr := i < g(y);
rexpr := k > f(x);
if( lexpr AND rexpr ) then begin

    i := 0;

end;
```

Don't be too concerned about the efficiency loss that could result from using these temporary variables. Any compiler that provides optimization facilities will put these values into registers and not bother using actual memory locations. Consider the following variant of the previous Pascal program written in C and compiled with the Visual C++ compiler:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    int lExpr;
    int rExpr;

    lExpr = i < g(y);
    rExpr = k > f(x);
    if( lExpr && rExpr )
    {
        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the conversion to 32-bit MASM code by the Visual C++ compiler (a few instructions have been rearranged to make their intent clearer):

```
main    PROC

$LN7:
        mov     QWORD PTR [rsp+8], rbx
        push    rdi
        sub     rsp, 32                                   ; 00000020H

; eax = g(y)
        mov     ecx, DWORD PTR y
        call    g
; ebx (lExpr) = i < g(y)
        xor     edi, edi
        cmp     DWORD PTR i, eax
        mov     ebx, edi ; ebx = 0
        setl    bl ;if i < g(y), set EBX to 1.

; eax = f(x)
        mov     ecx, DWORD PTR x
        call    f

; EDI = k > f(x)

        cmp     DWORD PTR k, eax
        setg    dil ; Sets EDI to 1 if k > f(x)

; See if lExpr is false:

        test    ebx, ebx
        je      SHORT $LN4@main

; See if rExpr is false:

        test    edi, edi
        je      SHORT $LN4@main

; "then" section of the if statement:

        lea     rcx, OFFSET FLAT:$SG7893
        call    printf
```

```
$LN4@main:

; return(0);
        xor     eax, eax

        mov     rbx, QWORD PTR [rsp+48]
        add     rsp, 32                                 ; 00000020H
        pop     rdi
        ret     0
main    ENDP
```

If you scan the assembly code, you'll see that this code fragment always executes the calls to both `f()` and `g()`. Contrast this with the following C code and assembly output:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    if( i < g(y) && k > f(x) )
    {
        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the MASM assembly output:

```
main    PROC

$LN7:
        sub     rsp, 40                                 ; 00000028H

; if (!(i < g(y))) then bail on the rest of the code:

        mov     ecx, DWORD PTR y
        call    g
        cmp     DWORD PTR i, eax
        jge     SHORT $LN4@main

; if (!(k > f(x))) then skip printf:

        mov     ecx, DWORD PTR x
        call    f
        cmp     DWORD PTR k, eax
        jle     SHORT $LN4@main

; Here's the body of the if statement.

        lea     rcx, OFFSET FLAT:$SG7891
        call    printf
$LN4@main:

; return 0
        xor     eax, eax

        add     rsp, 40                                  ; 00000028H
        ret     0
main    ENDP
```

In C, you can use another trick to force complete Boolean evaluation in any Boolean expression. The C bitwise operators do not support short-circuit Boolean evaluation. If your subexpressions in a Boolean expression always produce `0` or `1`, the bitwise Boolean conjunction and disjunction operators (that is, `&` and `|`) produce identical results to the logical Boolean operators (`&&` and `||`). Consider the following C code and the MASM code that the Visual C++ compiler produces:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    if( i < g(y) & k > f(x) )
    {
        printf( "Hello" );
    }
    return( 0 );
}
```

Here's the MASM code emitted by Visual C++:

```
main    PROC

$LN6:
        mov     QWORD PTR [rsp+8], rbx
        push    rdi
        sub     rsp, 32                                  ; 00000020H

        mov     ecx, DWORD PTR x
        call    f
        mov     ecx, DWORD PTR y
        xor     edi, edi
        cmp     DWORD PTR k, eax
        mov     ebx, edi
        setg    bl
        call    g
        cmp     DWORD PTR i, eax
        setl    dil
        test    edi, ebx
        je      SHORT $LN4@main

        lea     rcx, OFFSET FLAT:$SG7891
        call    printf
$LN4@main:

        xor     eax, eax

        mov     rbx, QWORD PTR [rsp+48]
        add     rsp, 32                                 ; 00000020H
        pop     rdi
        ret     0
main    ENDP
```

Note how the use of the bitwise operators produces comparable code to the earlier sequence that used temporary variables. This creates less clutter in your original C source file.

Do keep in mind, however, that C's bitwise operators produce the same results as the logical operators *only* if the operands are `0` and `1`. Fortunately, you can use a little C trick here: just write `!!(`*`expr`*`)`, and if the expression's value is zero or nonzero, C will convert the result to `0` or `1`. To see this in action, consider the following C/C++ code fragment:

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

int main( int argc, char **argv )
{
    int boolResult;

    boolResult = !!argc;
    printf( "!!(argc) = %d\n", boolResult );
    return 0;
}
```

Here's the 80x86 assembly code that Microsoft's Visual C++ compiler produces for this short program:

```
main    PROC
$LN4:
        sub     rsp, 40         ; 00000028H

        xor     edx, edx        ; EDX = 0
        test    ecx, ecx     ; System passes ARGC in ECX register
        setne   dl           ; If ECX==0, sets EDX=1, else EDX=0

        lea     rcx, OFFSET FLAT:$SG7886 ; Zero flag unchanged!
        call    printf       ; printf parm1 in RCX, parm2 in EDX

; Return 0;
        xor     eax, eax

        add     rsp, 40                 ; 00000028H
        ret     0
main    ENDP
```

As you can see in the 80x86 assembly output, only three machine instructions (involving no expensive branches) are needed to convert zero/nonzero to `0`/`1`.

13.4.3 Forcing Short-Circuit Evaluation in an if Statement

Although it's useful to be able to force complete Boolean evaluation on occasion, needing to force short-circuit evaluation is probably more common. Consider the following Pascal statement:

```
if( (ptrVar <> NIL) AND (ptrVar^ < 0) ) then begin

    ptrVar^ := 0;

end;
```

The Pascal language definition leaves it up to the compiler writer to decide whether to use complete Boolean evaluation or short-circuit evaluation. In fact, the writer is free to use both schemes as desired. Thus, it's quite possible that the same compiler could use complete Boolean evaluation for the previous statement in one section of the code and short-circuit evaluation in another.

You can see that this Boolean expression will fail if `ptrVar` contains the NIL pointer value and if the compiler uses complete Boolean evaluation. The only way to get this statement to work properly is by using short-circuit Boolean evaluation.

Simulating short-circuit Boolean evaluation with the AND operator is actually quite simple. All you have to do is create a pair of nested `if` statements and place each subexpression in each one. For example, you could guarantee short-circuit Boolean evaluation in the current Pascal example by rewriting it as follows:

```
if( ptrVar <> NIL ) then begin

    if( ptrVar^ < 0 ) then begin

        ptrVar^ := 0;
    end;

end;
```

This statement is semantically identical to the previous one. It should be clear that the second subexpression will not execute if the first expression evaluates to `false`. Even though this approach clutters up the source file a bit, it does guarantee short-circuit evaluation regardless of whether the compiler supports that scheme.

Handling the logical-OR operation is a little more difficult. Guaranteeing that the right operand of a logical-OR does not execute if the left operand evaluates to `true` requires an extra test. Consider the following C code (remember that C supports short-circuit evaluation by default):

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    if( i < g(y) || k > f(x) )
    {
        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the machine code that the Microsoft Visual C++ compiler produces:

```
main    PROC

$LN8:
        sub     rsp, 40             ; 00000028H

        mov     ecx, DWORD PTR y
        call    g
        cmp     DWORD PTR i, eax
        jl      SHORT $LN3@main
        mov     ecx, DWORD PTR x
        call    f
        cmp     DWORD PTR k, eax
        jle     SHORT $LN6@main
$LN3@main:

        lea     rcx, OFFSET FLAT:$SG6880
        call    printf
$LN6@main:

        xor     eax, eax

        add     rsp, 40                 ; 00000028H
        ret     0
main    ENDP
_TEXT   ENDS
```

Here's a version of the C program that implements short-circuit evaluation without relying on the C compiler to do so (not that this is necessary for C, as its language definition guarantees short-circuit evaluation, but you could use this approach in any language):

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    int temp;

        // Compute left subexpression and
        // save.

    temp = i < g(y);

        // If the left subexpression
        // evaluates to false, then try
        // the right subexpression.

    if( !temp )
    {
        temp = k > f(x);
    }

        // If either subexpression evaluates
        // to true, then print "Hello"

    if( temp )
    {
        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the corresponding MASM code emitted by the Microsoft Visual C++ compiler:

```
main    PROC

$LN9:
        sub     rsp, 40         ; 00000028H

        mov     ecx, DWORD PTR y
        call    g
        xor     ecx, ecx
        cmp     DWORD PTR i, eax
        setl    cl
        test    ecx, ecx

        jne     SHORT $LN7@main

        mov     ecx, DWORD PTR x
        call    f
        xor     ecx, ecx
        cmp     DWORD PTR k, eax
        setg    cl
        test    ecx, ecx

        je      SHORT $LN5@main
$LN7@main:

        lea     rcx, OFFSET FLAT:$SG6881
        call    printf
$LN5@main:

        xor     eax, eax

        add     rsp, 40            ; 00000028H
        ret     0
main    ENDP
```

As you can see, the code the compiler emits for the second version of the routine, which manually forces short-circuit evaluation, isn't quite as good as that emitted by the C compiler for the first example. However, if you need the semantics for short-circuit evaluation so the program will execute correctly, you'll have to live with possibly less efficient code than you'd get if the compiler supported this scheme directly.

If speed, minimal size, and short-circuit evaluation are all necessary, and you're willing to sacrifice a little readability and maintainability in your code to achieve them, then you can destructure the code and create something comparable to what the C compiler produces using short-circuit evaluation. Consider the following C code and the resulting output:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    if( i < g(y)) goto IntoIF;
    if( k > f(x) )
    {
      IntoIF:

        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the MASM output from Visual C++:

```
main    PROC

$LN8:
        sub     rsp, 40         ; 00000028H

        mov     ecx, DWORD PTR y
        call    g
        cmp     DWORD PTR i, eax
        jl      SHORT $IntoIF$9

        mov     ecx, DWORD PTR x
        call    f
        cmp     DWORD PTR k, eax
        jle     SHORT $LN6@main
$IntoIF$9:

        lea     rcx, OFFSET FLAT:$SG6881
        call    printf
$LN6@main:

        xor     eax, eax

        add     rsp, 40         ; 00000028H
        ret     0
main    ENDP
```

If you compare this code to the MASM output for the original C example (which relies on short-circuit evaluation), you'll see that this code is just as efficient. This is a classic example of why there was considerable resistance to structured programming in the 1970s among some programmers—sometimes it leads to less efficient code. Of course, readability and maintainability are usually more important than a few bytes or machine cycles. But never forget that if performance is paramount for a small section of code, destructuring that code can improve efficiency in some special cases.

13.5 The switch/case Statement

The `switch` (or `case`) high-level control statement is another conditional statement found in HLLs. As you've seen, an `if` statement tests a Boolean expression and executes one of two different paths in the code based on the result of the expression. A `switch/case` statement, on the other hand, can branch to one of several different points in the code based on the result of an ordinal (integer) expression. The following examples demonstrate the `switch` and `case` statements in C/C++, Pascal, and HLA. First, the C/C++ `switch` statement:

```
switch( expression )
{
  case 0:
    << statements to execute if the
       expression evaluates to 0 >>
    break;

  case 1:
    << statements to execute if the
       expression evaluates to 1 >>
    break;

  case 2:
    << statements to execute if the
       expression evaluates to 2>>
    break;

  <<etc>>

  default:
    << statements to execute if the expression is
       not equal to any of these cases >>
}
```

Java and Swift provide a similar syntax to C/C++ for the `switch` statement, although Swift's version has many additional features. We'll explore some of those additional features in the section "The Swift `switch` Statement" on page 500.

Here's an example of a Pascal `case` statement:

```
case ( expression ) of
  0: begin
    << statements to execute if the
```

```
      expression evaluates to 0 >>
    end;

  1: begin
    << statements to execute if the
      expression evaluates to 1 >>
    end;

  2: begin
    << statements to execute if the
      expression evaluates to 2>>
    end;

  <<etc>>

  else
    << statements to execute if
      REG32 is not equal to any of these cases >>

end; (* case *)
```

And finally, here's the HLA switch statement:

```
switch( REG32 )

  case( 0 )
    << statements to execute if
      REG32 contains 0 >>

  case( 1 )
    << statements to execute
      REG32 contains 1 >>

  case( 2 )
    << statements to execute if
      REG32 contains 2>>

  <<etc>>

  default
    << statements to execute if
      REG32 is not equal to any of these cases >>

endswitch;
```

As you can tell by these examples, these statements all share a similar syntax.

13.5.1 Semantics of a switch/case Statement

Most beginning programming classes and textbooks teach the semantics of the `switch`/`case` statement by comparing it with a chain of `if..else..if` statements; this introduces the `switch`/`case` statement using a concept the student already understands. Unfortunately, this approach can be misleading. To see why, consider the following code, which an introductory Pascal programming book might claim is equivalent to our Pascal case statement:

```
if( expression = 0 ) then begin

  << statements to execute if expression is 0 >>

end
else if( expression = 1 ) then begin

  << statements to execute if expression is 1 >>

end
else if( expression = 2 ) then begin

  << statements to execute if expression is 2 >>

end
else
  << statements to execute if expression is not 1 or 2 >>

end;
```

Although this particular sequence will achieve the same result as the `case` statement, there are several fundamental differences between the `if..then..elseif` sequence and the Pascal `case` implementation. First, the case labels in a `case` statement must all be constants, but in an `if..then..elseif` chain you can actually compare variables and other nonconstant values against the control variable. Another limitation of the `switch`/`case` statement is that you can compare only the value of a single expression against a set of constants; you cannot compare one expression against a constant for one case and a separate expression against a second constant, as you can with an `if..then..elseif` chain. The reason for these limitations will become clear in a moment, but the takeaway here is that an `if..then..elseif` chain is semantically different from—and more powerful than—a `switch`/`case` statement.

13.5.2 Jump Tables vs. Chained Comparisons

Although it is arguably more readable and convenient than an `if..then..elseif` chain, the `switch`/`case` statement was originally added to HLLs for efficiency, not readability or convenience. Consider an `if..then..elseif` chain with 10 separate expressions to test. If all the cases are mutually exclusive and equally likely, then on average the program will execute five comparisons before encountering an expression that evaluates to `true`. In assembly language, it's possible to transfer control to one of several different locations in a fixed amount of time, independent of the number of cases, by using a table lookup and an indirect jump. Effectively, such code uses the value of the `switch`/`case` expression as an index into a table of addresses and then jumps (indirectly) to the statement specified by the table entry. When you have more than three or four cases, this scheme is typically faster and consumes less memory than the corresponding `if..then..elseif` chain. Consider the following simple implementation of a `switch/case` statement in assembly language:

```
// Conversion of
//    switch(i)
//    { case 0:...case 1:...case 2:...case 3:...}
// into assembly

static
  jmpTable: dword[4] :=
    [ &label0, &label1, &label2, &label3 ];
      .
      .
      .
    // jmps to address specified by jmpTable[i]

    mov( i, eax );
    jmp( jmpTable[ eax*4 ] );

label0:
    << code to execute if i = 0 >>
    jmp switchDone;

label1:
    << code to execute if i = 1 >>
    jmp switchDone;

label2:
    << code to execute if i = 2 >>
    jmp switchDone;

label3:
    << code to execute if i = 3 >>

switchDone:
  << Code that follows the switch statement >>
```

To see how this code operates, we'll step through it one instruction at a time. The `jmpTable` declaration defines an array of four double-word pointers, one pointer for each case in our `switch` statement emulation. Entry 0 in the array holds the address of the statement to jump to when the `switch` expression evaluates to `0`, entry 1 contains the address of the statement to execute when the `switch` expression evaluates to `1`, and so on. Note that the array must have one element whose index matches each of the possible cases in the `switch` statement (`0` through `3` in this particular example).

The first machine instruction in this example loads the value of the `switch` expression (variable `i`'s value) into the EAX register. Because this code uses the `switch` expression's value as an index into the `jmpTable` array, this value must be an ordinal (integer) value in an 80x86 32-bit register. The next instruction (`jmp`) does the real work of the `switch` statement emulation: it jumps to the address specified by the entry in the `jmpTable` array, indexed by EAX. If EAX contains 0 upon execution of this jmp statement, the program fetches the double word from jmpTable[0] and transfers control to that address; this is the address of the first instruction following the label0 label in the program code. If EAX contains 1, then the jmp instruction fetches the double word at address jmpTable + 4 in memory (note that the *4 scaled-index addressing mode is used in this code; see "Indexed Addressing Mode" on page 34 for more details). Likewise, if EAX contains 2 or 3, then the jmp instruction transfers control to the double-word address held at jmpTable + 8 or jmpTable + 12 (respectively). Because the jmpTable array is initialized with the addresses of label0, label1, label2, and label3, at respective offsets 0, 4, 8, and 12, this particular indirect jmp instruction will transfer control to the statement at the label corresponding to i's value (label0, label1, label2, or label3, respectively).

The first point of interest about this switch statement emulation is that it requires only two machine instructions (and a jump table) to transfer control to any of the four possible cases. Contrast this with an if..then..elseif implementation, which requires at least two machine instructions for each case. Indeed, as you add more cases to the if..then..elseif implementation, the number of compare and conditional branch instructions increases, yet the number of machine instructions for the jump table implementation remains fixed at two (even though the size of the jump table increases by one entry for each case). Accordingly, the if..then..elseif implementation gets progressively slower as you add more cases, while the jump table implementation takes a constant amount of time to execute (regardless of the number of cases). Assuming your HLL compiler uses a jump table implementation for switch statements, a switch statement will typically be much faster than an if..then..elseif sequence if there are a large number of cases.

The jump table implementation of switch statements does have a couple of drawbacks, though. First, because the jump table is an array in memory, and accessing (noncached) memory can be slow, accessing the jump table array could possibly impair system performance.

Another downside is that you must have one entry in the table for every possible case between the largest and the smallest case values, including those values for which you haven't actually supplied an explicit case. In the example up to this point, this hasn't been an issue because the case values started with 0 and were contiguous through 3. However, consider the following Pascal case statement:

```
case( i ) of

  0: begin
      << statements to execute if i = 0 >>
     end;

  1: begin
      << statements to execute if i = 1 >>
     end;
  5: begin
      << statements to execute if i = 5 >>
     end;

  8: begin
      << statements to execute if i = 8 >>
     end;

end; (* case *)
```

We can't implement this case statement with a jump table containing four entries. If the value of i were 0 or 1, then it would fetch the correct address. However, for case 5, the index into the jump table would be 20 (5 × 4), not the third (2 x 4 = 8) entry in the jump table. If the jump table contained only four entries (16 bytes), indexing into the jump table using the value 20 would grab an address beyond the end of the table and likely crash the application. This is exactly why in the original definition of Pascal, the results were undefined if the program supplied a case value that wasn't present in the set of labels for a particular case statement.

To solve this problem in assembly language, you must make sure there are entries for each of the possible case labels as well as all values in between them. In the current example, the jump table would need nine entries to handle all the possible case values, 0 through 8:

```
// Conversion of
//    switch(i)
//    { case 0:...case 1:...case 5:...case 8:}
// into assembly

static
  jmpTable: dword[9] :=
          [
            &label0, &label1, &switchDone,
            &switchDone, &switchDone,
            &label5, &switchDone, &switchDone,
            &label8
          ];
      .
      .
      .
    // jumps to address specified by jmpTable[i]

    mov( i, eax );
    jmp( jmpTable[ eax*4 ] );

label0:
    << code to execute if i = 0 >>
    jmp switchDone;

label1:
    << code to execute if i = 1 >>
    jmp switchDone;
```

```
label5:
    << code to execute if i = 5 >>
    jmp switchDone;

label8:
    << code to execute if i = 8 >>

switchDone:
  << Code that follows the switch statement >>
```

Notice that if `i` is equal to `2`, `3`, `4`, `6`, or `7`, then this code transfers control to the first statement beyond the `switch` statement (the standard semantics for C's `switch` statement and the `case` statement in most modern variants of Pascal). Of course, C will also transfer control to this point in the code if the `switch`/`case` expression value is greater than the largest case value. Most compilers implement this feature with a comparison and conditional branch immediately before the indirect jump. For example:

```
// Conversion of
//    switch(i)
//    { case 0:...case 1:...case 5:...case 8:}
// into assembly, that automatically
// handles values greater than 8.

static
  jmpTable: dword[9] :=
          [
            &label0, &label1, &switchDone,
            &switchDone, &switchDone,
            &label5, &switchDone, &switchDone,
            &label8
          ];
      .
      .
      .
    // Check to see if the value is outside the range
    // of values allowed by this switch/case stmt.

    mov( i, eax );
    cmp( eax, 8 );
    ja switchDone;

    // jmps to address specified by jmpTable[i]

    jmp( jmpTable[ eax*4 ] );

      .
      .
      .

switchDone:
  << Code that follows the switch statement >>
```

You may have noticed another assumption that this code is making—that the case values start at 0. Modifying the code to handle an arbitrary range of case values is simple. Consider the following example:

```
// Conversion of
//    switch(i)
//    { case 10:...case 11:...case 12:...case 15:...case 16:}
// into assembly, that automatically handles values
// greater than 16 and values less than 10.

static
  jmpTable: dword[7] :=
          [
            &label10, &label11, &label12,
            &switchDone, &switchDone,
            &label15, &label16
          ];
      .
      .
      .
    // Check to see if the value is outside the
    // range 10..16.

    mov( i, eax );
    cmp( eax, 10 );
    jb switchDone;
    cmp( eax, 16 );
    ja switchDone;

    // The "- 10*4" part of the following expression
    // adjusts for the fact that EAX starts at 10
    // rather than 0, but we still need a zero-based
    // index into our array.

    jmp( jmpTable[ eax*4 - 10*4] );

      .
      .
      .

switchDone:
  << Code that follows the switch statement >>
```

There are two differences between this example and the previous one. First, this example compares the value in EAX against the range `10..16` and, if the value falls outside this range, branches to the `switchDone` label (in other words, there is no case label for the value in EAX). Second, the `jmpTable` index has been modified to be `[eax*4 - 10*4]`. Arrays at the machine level always begin at index `0`; the "`- 10*4`" component of this expression adjusts for the fact that EAX actually contains a value starting at `10` rather than `0`. Effectively, this expression makes `jmpTable` start 40 bytes earlier in memory than its declaration states. Because EAX is always 10 or greater (40 bytes or greater because of the `eax*4` component), this code begins accessing the table at its declared beginning location. Note that HLA subtracts this offset from the address of `jmpTable`; the CPU doesn't actually perform this subtraction at runtime. Hence, there is no additional efficiency loss to create this zero-based index.

Notice that a fully generalized `switch`/`case` statement actually requires six instructions to implement: the original two instructions plus four instructions to test the range.[3] This, plus the fact that an indirect jump is slightly more expensive to execute than a conditional branch, is why the break-even point for a `switch`/`case` statement (versus an `if..then..elseif` chain) is around three to four cases.

As mentioned earlier, one serious drawback to the jump table implementation of the `switch`/`case` statement is the fact that you must have one table entry for every possible value between the smallest case and the largest case. Consider the following C/C++ `switch` statement:

```
switch( i )
{
  case 0:
      << statements to execute if i == 0 >>
      break;

  case 1:
      << statements to execute if i == 1 >>
      break;

  case 10:
      << statements to execute if i == 10 >>
      break;

  case 100:
      << statements to execute if i == 100 >>
      break;

  case 1000:
      << statements to execute if i == 1000 >>
      break;

  case 10000:
      << statements to execute if i == 10000 >>
      break;
}
```

If the C/C++ compiler implements this `switch` statement using a jump table, that table will require 10,001 entries (that is, 40,004 bytes of memory on a 32-bit processor). That's quite a chunk of memory for such a simple statement! Although the wide separation of the cases has a major effect on memory usage, it has only a minor effect on the execution speed of the `switch` statement. The program executes the same four instructions it would

3. Actually, with a little assembly language trickery, a good programmer or compiler can reduce this from four to three machine instructions (with only a single branch).

execute if the values were all contiguous (only four instructions are necessary because the case values start at 0, so there's no need to check the switch expression against a lower bound). Indeed, the only reason there's a performance difference at all is because of the effects of the table size on the cache (it's less likely you will find a particular table entry in the cache when the table is large). Speed issues aside, the memory usage by the jump table is difficult to justify for most applications. Therefore, if your particular compiler emits a jump table for all switch/case statements (which you can determine by looking at the code it produces), you should be careful about creating switch/case statements whose cases are widely separated.

13.5.3 Other Implementations of switch/case

Because of the issue with jump table sizes, some HLL compilers do not implement switch/case statements using jump tables. Some compilers will simply convert a switch/case statement into the corresponding if..then..elseif chain (Swift falls into this category). Obviously, such compilers tend to produce low-quality code (from a speed point of view) whenever a jump table would be appropriate. Many modern compilers are relatively smart about their code generation. They'll determine the number of cases in a switch/case statement as well as the spread of the case values. Then the compiler will choose a jump table or if..then..elseif implementation based on some threshold criteria (code size versus speed). Some compilers might even use a combination of the techniques. For example, consider the following Pascal case statement:

```
case( i ) of
  0: begin
     << statements to execute if i = 0 >>
    end;

  1: begin
     << statements to execute if i = 1 >>
    end;

  2: begin
     << statements to execute if i = 2 >>
    end;

  3: begin
     << statements to execute if i = 3 >>
    end;

  4: begin
     << statements to execute if i = 4 >>
    end;

  1000: begin
     << statements to execute if i = 1000 >>
       end;
end; (* case *)
```

A good compiler will recognize that the majority of the cases work well in a jump table, with the exception of only one (or a few) cases. It will translate this code to a sequence of instructions that combine the if..then and jump table implementation. For example:

```
    mov( i, eax );
    cmp( eax, 4 );
    ja try1000;
    jmp( jmpTable[ eax*4 ] );
      .
      .
      .
try1000:
    cmp( eax, 1000 );
    jne switchDone;
    << code to do if i = 1000 >>
switchDone:
```

Although the switch/case statement was originally created to allow the use of an efficient jump table transfer mechanism in an HLL, there are few language definitions that require a specific implementation for a control structure. Therefore, unless you stick with a specific compiler and you know how that compiler generates code under all circumstances, there's absolutely no guarantee that your switch/case statements will compile to a jump table, an if..then..elseif chain, some combination of the two, or something else entirely. For example, consider the following short C program and the resulting assembly output:

```
extern void f( void );
extern void g( void );
extern void h( void );
int main( int argc, char **argv )
{
    int boolResult;

    switch( argc )
    {
        case 1:
            f();
            break;

        case 2:
            g();
            break;

        case 10:
            h();
            break;

        case 11:
            f();
            break;
    }
    return 0;
}
```

Here's the 80x86 output from the (older) Borland C++ v5.0 compiler:

```
_main   proc    near
?live1@0:
   ;
   ;    int main( int argc, char **argv )
   ;
@1:
    push      ebp
    mov       ebp,esp
   ;
   ;    {
   ;        int boolResult;
   ;
   ;        switch( argc )
   ;

; Is argc == 1?

    mov       eax,dword ptr [ebp+8]
    dec       eax
    je        short @7

; Is argc == 2?

    dec       eax
    je        short @6

; Is argc == 10?

    sub       eax,8
    je        short @5

; Is argc == 11?

    dec       eax
    je        short @4

; If none of the above

    jmp       short @2
   ;
   ;        {
   ;            case 1:
   ;                f();
   ;
@7:
    call      _f
   ;
   ;                break;
   ;
```

```
    jmp       short @8
   ;
   ;
   ;           case 2:
   ;               g();
   ;
@6:
    call      _g
   ;
   ;               break;
   ;
    jmp       short @8
   ;
   ;
   ;           case 10:
   ;               h();
   ;
@5:
    call      _h
   ;
   ;               break;
   ;
    jmp       short @8
   ;
   ;
   ;           case 11:
   ;               f();
   ;
@4:
    call      _f
   ;
   ;               break;
   ;
   ;         }
   ;         return 0;
   ;
@2:
@8:
    xor       eax,eax
   ;
   ;    }
   ;
@10:
@9:
    pop       ebp
    ret
_main   endp
```

As you can see at the beginning of the main program, this code compares the value in `argc` against the four values (`1`, `2`, `10`, and `11`) sequentially. For a `switch` statement as small as this one, this isn't a bad implementation.

When there are a fair number of cases and a jump table would be too large, many modern optimizing compilers generate a binary search tree to test the cases. For example, consider the following C program and the corresponding output:

```
#include <stdio.h>

extern void f( void );
int main( int argc, char **argv )
{
    int boolResult;

    switch( argc )
    {
        case 1:
            f();
            break;

        case 10:
            f();
            break;

        case 100:
            f();
            break;

        case 1000:
            f();
            break;

        case 10000:
            f();
            break;

        case 100000:
            f();
            break;

        case 1000000:
            f();
            break;

        case 10000000:
            f();
            break;

        case 100000000:
            f();
            break;

        case 1000000000:
            f();
            break;
    }
    return 0;
}
```

Here's the 64-bit MASM output from the Visual C++ compiler. Note how Microsoft's compiler generates a partial binary search through each of the 10 cases:

```
main    PROC

$LN18:
        sub     rsp, 40                                 ; 00000028H

; >+ 100,000?
        cmp     ecx, 100000                             ; 000186a0H
        jg      SHORT $LN15@main
        je      SHORT $LN10@main

; handle cases where argc is less than 100,000
;
; Check for argc = 1

        sub     ecx, 1
        je      SHORT $LN10@main

; check for argc = 10

        sub     ecx, 9
        je      SHORT $LN10@main

;check for argc = 100

        sub     ecx, 90                                 ; 0000005aH
        je      SHORT $LN10@main

; check for argc = 1000

        sub     ecx, 900                                ; 00000384H
        je      SHORT $LN10@main

; check for argc = 1000
        cmp     ecx, 9000                               ; 00002328H

        jmp     SHORT $LN16@main
$LN15@main:

; Check for argc = 100,000

      cmp     ecx, 1000000                              ; 000f4240H
        je      SHORT $LN10@main

; check for argc = 1,000,000
        cmp     ecx, 10000000                           ; 00989680H
        je      SHORT $LN10@main
```

```
; check for argc = 10,000,000
        cmp     ecx, 100000000                          ; 05f5e100H
        je      SHORT $LN10@main

; check for argc = 100,000,000

        cmp     ecx, 1000000000                         ; 3b9aca00H
$LN16@main:
        jne     SHORT $LN2@main
$LN10@main:

        call    f
$LN2@main:

        xor     eax, eax

        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
```

Interestingly enough, when compiling to 32-bit code, Visual C++ produces a true binary search. Here's the MASM32 output from the 32-bit version of Visual C++:

```
_main   PROC

        mov     eax, DWORD PTR _argc$[esp-4] ; argc is passed on stack in 32-bit code

; Start with >100,000, = 100,000, or < 100,000

        cmp     eax, 100000                 ; 000186a0H
        jg      SHORT $LN15@main            ; Go if >100,000
        je      SHORT $LN4@main             ; Match if equal

; Handle cases where argc < 100,000
;
; Divide it into >100 and < 100

        cmp     eax, 100                    ; 00000064H
        jg      SHORT $LN16@main            ; Branch if > 100
        je      SHORT $LN4@main             ; = 100

; Down here if < 100

        sub     eax, 1
        je      SHORT $LN4@main             ; branch if it was 1

        sub     eax, 9                      ; Test for 10
        jmp     SHORT $LN18@main

; Come down here if >100 and <100,000
```

```
$LN16@main:

        cmp     eax, 1000                     ; 000003e8H
        je      SHORT $LN4@main               ; Branch if 1000
        cmp     eax, 10000                    ; 00002710H
        jmp     SHORT $LN18@main              ; Handle =10,000 or not in range

; Handle > 100,000 here.

$LN15@main:
        cmp     eax, 100000000                ; 05f5e100H
        jg      SHORT $LN17@main              ; > 100,000,000
        je      SHORT $LN4@main               ; = 100,000

; Handle < 100,000,000 and > 100,000 here:

        cmp     eax, 1000000                  ; 000f4240H
        je      SHORT $LN4@main               ; =1,000,000
        cmp     eax, 10000000                 ; 00989680H

        jmp     SHORT $LN18@main              ; Handle 10,000,000 or not in range

; Handle > 100,000,000 here
$LN17@main:
; check for 1,000,000,000
        cmp     eax, 1000000000               ; 3b9aca00H
$LN18@main:
        jne     SHORT $LN2@main
$LN4@main:

        call    _f
$LN2@main:

        xor     eax, eax

        ret     0
_main   ENDP
```

Some compilers, especially those for some microcontroller devices, generate a table of *2-tuples* (paired records/structures), with one element of the tuple being the value of the case and the second element being the address to jump to if the value matches. Then the compiler emits a loop that scans through this little table searching for the current `switch`/`case` expression value. If this is a linear search, this implementation is even slower than the `if..then..elseif` chain. If the compiler emits a binary search, the code may be faster than an `if..then.elseif` chain but probably not as fast as a jump table implementation.

Here's a Java example of a `switch` statement, along with the Java bytecode the compiler produces:

```
public class Welcome
{
    public static void f(){}
    public static void main( String[] args )
    {
        int i = 10;
        switch (i)
        {
            case 1:
                f();
                break;

            case 10:
                f();
                break;

            case 100:
                f();
                break;

            case 1000:
                f();
                break;

            case 10000:
                f();
                break;

            case 100000:
                f();
                break;

            case 1000000:
                f();
                break;

            case 10000000:
                f();
                break;

            case 100000000:
                f();
                break;

            case 1000000000:
                f();
                break;

        }
    }
}

// JBC output:

Compiled from "Welcome.java"
public class Welcome extends java.lang.Object{
public Welcome();
```

```
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void f();
  Code:
   0:   return

public static void main(java.lang.String[]);
  Code:
   0:   bipush  10
   2:   istore_1
   3:   iload_1
   4:   lookupswitch{ //10
        1: 96;
        10: 102;
        100: 108;
        1000: 114;
        10000: 120;
        100000: 126;
        1000000: 132;
        10000000: 138;
        100000000: 144;
        1000000000: 150;
        default: 153 }
   96:  invokestatic    #2; //Method f:()V
   99:  goto    153
   102: invokestatic    #2; //Method f:()V
   105: goto    153
   108: invokestatic    #2; //Method f:()V
   111: goto    153
   114: invokestatic    #2; //Method f:()V
   117: goto    153
   120: invokestatic    #2; //Method f:()V
   123: goto    153
   126: invokestatic    #2; //Method f:()V
   129: goto    153
   132: invokestatic    #2; //Method f:()V
   135: goto    153
   138: invokestatic    #2; //Method f:()V
   141: goto    153
   144: invokestatic    #2; //Method f:()V
   147: goto    153
   150: invokestatic    #2; //Method f:()V
   153: return

}
```

The `lookupswitch` bytecode instruction contains a table of 2-tuples. As described earlier, the first value of the tuple is the case value, and the second is the target address where the code transfers on a match. Presumably, the bytecode interpreter does a binary search on these values rather than a linear search (one would hope!). Notice that the Java compiler generates a separate call to method f() for each of the cases; it doesn't optimize them to a single call as GCC and Visual C++ do.

NOTE *Java also has a* `tableswitch` *VM instruction that executes a table-driven* `switch` *operation. The Java compiler chooses between the* `tableswitch` *and* `lookupswitch` *instructions based on the density of the case values.*

Sometimes, compilers resort to some code tricks to generate marginally better code under certain circumstances. Consider again the short switch statement that led the Borland compiler to produce a linear search:

```
switch( argc )
    {
        case 1:
            f();
            break;

        case 2:
            g();
            break;

        case 10:
            h();
            break;

        case 11:
            f();
            break;

    }
```

Here's the code that the Microsoft Visual C++ 32-bit compiler generates for this switch statement:

```
; File t.c
; Line 13
;
; Use ARGC as an index into the $L1240 table,
; which returns an offset into the $L1241 table:

    mov eax, DWORD PTR _argc$[esp-4]
    dec eax         ; --argc, 1=0, 2=1, 10=9, 11=10
    cmp eax, 10     ; Out of range of cases?
    ja  SHORT $L1229
    xor ecx, ecx
    mov cl, BYTE PTR $L1240[eax]
    jmp DWORD PTR $L1241[ecx*4]

    npad    3
$L1241:
    DD  $L1232  ; cases that call f
    DD  $L1233  ; cases that call g
    DD  $L1234  ; cases that call h
    DD  $L1229  ; Default case

$L1240:
    DB  0   ; case 1 calls f
    DB  1   ; case 2 calls g
    DB  3   ; default
    DB  3   ; default
    DB  3   ; default
    DB  3   ; default
    DB  3   ; default
    DB  3   ; default
    DB  3   ; default
    DB  2   ; case 10 calls h
    DB  0   ; case 11 calls f

; Here is the code for the various cases:

$L1233:
; Line 19
    call    _g
; Line 31
    xor eax, eax
; Line 32
    ret 0

$L1234:
; Line 23
    call    _h
; Line 31
    xor eax, eax
; Line 32
    ret 0

$L1232:
; Line 27
    call    _f
$L1229:
; Line 31
    xor eax, eax
; Line 32
    ret 0
```

The trick in this 80x86 code is that Visual C++ first does a table lookup to make an `argc` value in the range `1..11` to a value in the range `0..3` (which corresponds to the three different code bodies appearing in the cases, plus a default case). This code is shorter than a jump table, with the corresponding double-word entries mapping to the default case, although it's a little slower than a jump table because it needs to access two different tables in memory. (As for how the speed of this code compares with a binary search or linear search, that research is left to you; the answer will probably vary by processor.) Note, however, that when producing 64-bit code, Visual C++ reverts to the linear search:

```
main    PROC

$LN12:
        sub     rsp, 40                                 ; 00000028H

; ARGC is passed in ECX

        sub     ecx, 1
        je      SHORT $LN4@main  ; case 1
        sub     ecx, 1
        je      SHORT $LN5@main  ; case 2
        sub     ecx, 8
        je      SHORT $LN6@main  ; case 10
        cmp     ecx, 1
        jne     SHORT $LN10@main ; case 11
$LN4@main:

        call    f
$LN10@main:

        xor     eax, eax

        add     rsp, 40                                 ; 00000028H
        ret     0
$LN6@main:

        call    h

        xor     eax, eax

        add     rsp, 40                                 ; 00000028H
        ret     0
$LN5@main:

        call    g

        xor     eax, eax

        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
```

Few compilers give you the option of explicitly specifying how the compiler will translate a specific `switch`/`case` statement. For example, if you really want the `switch` statement with cases 0, 1, 10, 100, 1,000, and 10,000 given earlier to generate a jump table, you'll have to write the code in assembly language or use a specific compiler whose code generation traits you understand. Any HLL code you've written that depends on the compiler generating a jump table won't be portable to other compilers, however, because few languages specify the actual machine code implementation of high-level control structures.

Of course, you don't have to totally rely on the compiler to generate decent code for a `switch`/`case` statement. Assuming your compiler uses the jump table implementation for all `switch`/`case` statements, you can help it produce better code when modifications to your HLL source code would generate a huge jump table. For example, consider the `switch` statement given earlier with the cases 0, 1, 2, 3, 4, and 1,000. If your compiler generates a jump table with 1,001 entries (consuming a little more than 4KB of memory), you can improve its output by writing the following Pascal code:

```
if( i = 1000 ) then begin

  << statements to execute if i = 1000 >>

end
else begin

  case( i ) of
    0: begin
         << statements to execute if i = 0 >>
       end;

    1: begin
         << statements to execute if i = 1 >>
       end;

    2: begin
         << statements to execute if i = 2 >>
       end;

    3: begin
         << statements to execute if i = 3 >>
       end;

    4: begin
         << statements to execute if i = 4 >>
       end;
  end; (* case *)
end; (* if *)
```

By handling case value `1000` outside the `switch` statement, the compiler can produce a short jump table for the main cases, which are contiguous.

Another possibility (which is arguably easier to read) is the following C/C++ code:

```
switch( i )
{
  case 0:
      << statements to execute if i == 0 >>
      break;

  case 1:
      << statements to execute if i == 1 >>
      break;

  case 2:
      << statements to execute if i == 2 >>
      break;

  case 3:
      << statements to execute if i == 3 >>
      break;

  case 4:
      << statements to execute if i == 4 >>
     break;

  default:
    if( i == 1000 )
    {
      << statements to execute if i == 1000 >>
    }
    else
    {
      << Statements to execute if none of the cases match >>
    }
}
```

What makes this example slightly easier to read is that the code for the case when `i` is equal to `1000` has been moved into the `switch` statement (thanks to the default clause), so it doesn't appear to be separate from all the tests taking place in the `switch`.

Some compilers simply won't generate a jump table for a `switch/case` statement. If you're using such a compiler and you want to generate a jump table, there's little you can do—short of dropping into assembly language or using nonstandard C extensions.

Although jump table implementations of `switch/case` statements are generally efficient when you have a fair number of cases and they're all equally likely, remember that an `if..then..elseif` chain can be faster if one or two cases are far more likely than the others. For example, if a variable has the value `15` more than half the time, the value `20` about a quarter of the time, and one of several different values the remaining 25 percent of the time, it's probably more efficient to implement the multiway test using an `if..then..elseif` chain (or a combination of `if..then..elseif` and a `switch/case` statement). By testing the most common case(s) first, you can often reduce the average time the multiway statement needs to execute. For example:

```
if( i == 15 )
{
  // If i = 15 better than 50% of the time,
  // then we only execute a single test
  // better than 50% of the time:
}
else if( i == 20 )
{
  // if i == 20 better than 25% of the time,
  // then we only execute one or
  // two comparisons 75% of the time.
}
else if etc....
```

If `i` is equal to `15` more often than not, then most of the time this code sequence will execute the body of the first `if` statement after executing only two instructions. Even in the best `switch` statement implementation, you're going to need more instructions than this.

13.5.4 The Swift switch Statement

Swift's `switch` statement is semantically different from most other languages. There are four major differences between Swift's `switch` and the typical C/C++ `switch` or Pascal `case` statement:

- Swift's `switch` provides a special `where` clause that lets you apply a conditional to a `switch`.
- Swift's `switch` allows you to use the same value in more than one `case` statement (differentiated by the `where` clause).
- Swift's `switch` allows the use of nonintegral/ordinal data types, such as tuples, strings, and sets, as the selection value (with appropriately typed case values).
- Swift's `switch` statement supports pattern matching for case values.

Check out the Swift language reference manual for more details. The purpose of this section is not to provide the syntax and semantics of the Swift `switch`, but rather to discuss how Swift's design affects its implementation.

Because it allows arbitrary types as the `switch` selector value, there's no way that Swift could use a jump table to implement the `switch` statement. A jump table implementation requires an ordinal value (something you can represent as an integer) that the compiler can use as an index into the jump table. A string selector, for example, couldn't be used as an index into an array. Furthermore, Swift allows you to specify the same case value twice,[4] creating a consistency problem with the same jump table entry mapping to two separate sections of code (which is impossible for a jump table).

4. If you supply two identical cases, you would normally use the `where` clause to differentiate between the two. If there is no `where` clause, or if the two `where` clauses both evaluate `true`, Switch executes the first case it encounters.

Given the design of the Swift `switch` statement, then, the only solution is a linear search (effectively, the `switch` statement is equivalent to a chain of `if..else if..else if..etc.` statements). The bottom line is that there is no performance benefit to using the `switch` statement over a set of `if` statements.

13.5.5 Compiler Output for switch Statements

Before you run off to help your compiler produce better code for `switch` statements, you might want to examine the actual code it produces. This chapter has described several of the techniques that various compilers use for implementing `switch`/`case` statements at the machine code level, but there are several additional implementations that this book could not cover. Although you can't assume that a compiler will always generate the same code for a `switch`/`case` statement, observing its output can help you see the different implementations that compiler authors use.

13.6 For More Information

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Essex, UK: Pearson Education Limited, 1986.

Barrett, William, and John Couch. *Compiler Construction: Theory and Practice*. Chicago: SRA, 1986.

Dershem, Herbert, and Michael Jipping. *Programming Languages, Structures and Models*. Belmont, CA: Wadsworth, 1990.

Duntemann, Jeff. *Assembly Language Step-by-Step*. 3rd ed. Indianapolis: Wiley, 2009.

Fraser, Christopher, and David Hansen. *A Retargetable C Compiler: Design and Implementation*. Boston: Addison-Wesley Professional, 1995.

Ghezzi, Carlo, and Jehdi Jazayeri. *Programming Language Concepts*. 3rd ed. New York: Wiley, 2008.

Hoxey, Steve, Faraydon Karim, Bill Hay, and Hank Warren, eds. *The PowerPC Compiler Writer's Guide*. Palo Alto, CA: Warthman Associates for IBM, 1996.

Hyde, Randall. *The Art of Assembly Language*. 2nd ed. San Francisco: No Starch Press, 2010.

Intel. "Intel 64 and IA-32 Architectures Software Developer Manuals." Updated November 11, 2019. *https://software.intel.com/en-us/articles/intel-sdm*.

Ledgard, Henry, and Michael Marcotty. *The Programming Language Landscape*. Chicago: SRA, 1986.

Louden, Kenneth C. *Compiler Construction: Principles and Practice*. Boston: Cengage, 1997.

Louden, Kenneth C., and Kenneth A. Lambert. *Programming Languages: Principles and Practice.* 3rd ed. Boston: Course Technology, 2012.

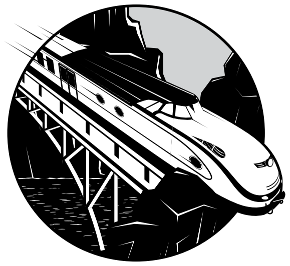
Parsons, Thomas W. *Introduction to Compiler Construction.* New York: W. H. Freeman, 1992.

Pratt, Terrence W., and Marvin V. Zelkowitz. *Programming Languages, Design and Implementation.* 4th ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Sebesta, Robert. *Concepts of Programming Languages.* 11th ed. Boston: Pearson, 2016.

14

ITERATIVE CONTROL STRUCTURES



Most programs spend the majority of their time executing machine instructions within a loop. Therefore, if you want to improve your applications' execution speed, first you should see if you can improve the performance of the loops in your code. This chapter will describe the following varieties of loops:

- `while` loops
- `repeat..until`/`do..while` loops
- `forever` (infinite) loops
- `for` (definite) loops

14.1 The while Loop

The while loop is perhaps the most general-purpose iterative statement that HLLs provide, so compilers generally work hard at emitting optimal code for it. The while loop tests a Boolean expression at the top of the loop body and, if the expression evaluates to true, executes the loop body. When the loop body completes execution, control transfers back to the test and the process repeats. When the Boolean control expression evaluates to false, the program transfers control to the first statement beyond the loop's body. This means that if the Boolean expression evaluates to false when the program first encounters the while statement, the program immediately skips over all statements in the loop body without executing any of them. The following example demonstrates a Pascal while loop:

```
while( a < b ) do begin

  << Statements to execute if a is less than b.
     Presumably, these statements modify the value
     of either a or b so that this loop ultimately
     terminates. >>

end; (* while *)
<< statements that execute when a is not less than b >>
```

You can easily simulate a while loop in an HLL by using an if statement and a goto statement. Consider the following C/C++ while loop and the semantically equivalent code using an if and a goto:

```
// while loop:

while( x < y )
{
  arr[x] = y;
  ++x;
}

// Conversion to an if and a goto:

whlLabel:
if( x < y )
{
  arr[x] = y;
  ++x;
  goto whlLabel;
}
```

Assume for the sake of this example that x is less than y when the if/goto combination first executes. This being true, the body of the loop (the then portion of the if statement) executes. At the bottom of the loop body, the goto statement transfers control back to just before the if statement. This means that the code will test the expression again, just as the while loop does. Whenever the `if` expression evaluates to `false`, control will transfer to the first statement after the `if` (which transfers control beyond the `goto` statement in this code).

Although the `if`/`goto` arrangement is semantically identical to the `while` loop, that's not to suggest that the `if`/`goto` scheme presented here is more efficient than what a typical compiler would generate. It's not. The following assembly code shows what you'd get from a mediocre compiler for the previous `while` loop:

```
  // while( x < y )

whlLabel:
    mov( x, eax );
    cmp( eax, y );
    jnl exitWhile;  // jump to exitWhile label if
                    // x is not less than y

    mov( y, edx );
    mov( edx, arr[ eax*4 ] );
    inc( x );
    jmp whlLabel;
exitWhile:
```

A decent compiler will improve upon this slightly by using a technique known as *code movement* (or *expression rotation*). Consider this slightly more efficient implementation of the previous `while` loop:

```
// while( x < y )

    // Skip over the while loop's body.

    jmp testExpr;

whlLabel:
    // This is the body of the while loop (same as
    // before, except moved up a few instructions).

    mov( y, edx );
    mov( edx, arr[ eax*4 ] );
    inc( x );

// Here is where we test the expression to
// determine if we should repeat the loop body.

testExpr:
    mov( x, eax );
    cmp( eax, y );
    jl whlLabel;    // Transfer control to loop body if x < y.
```

This example has exactly the same number of machine instructions as the previous example, but the test for loop termination has been moved to the bottom of the loop. To preserve the semantics of a `while` loop (so that we don't execute the loop body if the expression evaluates to `false` upon first encountering the loop), the first statement in this sequence is a `jmp` statement that transfers control down to the code that tests the loop termination expression. If that test evaluates to `true`, the program transfers control to the body of the `while` loop (immediately after `whlLabel`).

Although this code has the same number of statements as the previous example, there's a subtle difference between the two implementations. In this latter example, the initial `jmp` instruction executes only once—the very first time the loop executes. For each iteration thereafter, the code skips the execution of this statement. In the original example, the corresponding `jmp` statement is at the bottom of the loop's body, and it executes on each iteration of the loop. Therefore, if the loop body executes more than once, the second version runs faster (on the other hand, if the `while` loop rarely executes the loop body even once, then the first version is slightly more efficient). If your compiler does not generate the best code for a `while` statement, consider getting a different compiler. As Chapter 13 discussed, attempting to write optimal code in an HLL by using `if` and `goto` statements will produce difficult-to-read spaghetti code and, more often than not, `goto` statements in your code will actually impair the compiler's ability to produce decent output.

NOTE *When this chapter discusses the* `repeat..until`/`do..while` *loop, you'll see an alternative to the* `if..goto` *scheme that will produce more structured code that the compiler may be able to handle. Still, if your compiler cannot make a simple transformation like this one, chances are the efficiency of the compiled* `while` *loops is among the least of your problems.*

Compilers that do a decent job of optimizing `while` loops typically make certain assumptions about the loop, the biggest one being that the loop has exactly one entry point and one exit point. Many languages provide statements allowing the premature exit of a loop (for example, `break`, as discussed in "Restricted Forms of the `goto` Statement" on page 459). Of course, many languages provide some form of the `goto` statement that will allow you to enter or exit the loop at an arbitrary point. However, keep in mind that using such statements, while probably legal, may severely affect the compiler's ability to optimize the code. So use them with caution.[1] The `while` loop is one area where you should let the compiler do its job rather than trying to optimize the code yourself (actually, this applies for all loops, as compilers generally do a good job of optimizing loops).

14.1.1 Forcing Complete Boolean Evaluation in a while Loop

The execution of a `while` statement depends upon the semantics of Boolean expression evaluation. As with the `if` statement, sometimes the correct execution of a `while` loop depends upon whether the Boolean expression

1. It is a paradox that many programmers use multiple entries or exits within a loop in an effort to optimize their code, yet their hard work often destroys the very thing they are trying to achieve.

uses complete evaluation or short-circuit evaluation. This section describes ways to force a while loop to use full Boolean evaluation, and the following section will demonstrate ways to force short-circuit evaluation.

At first blush, you might guess that forcing complete Boolean evaluation in a while loop is done the same way as in an if statement. However, if you look back at the solutions given for the if statement (see "Forcing Complete Boolean Evaluation in an if Statement" on page 465), you'll realize that the approaches we used for the if statement (nesting ifs and temporary calculations) won't work for a while statement. We need a different approach.

14.1.1.1 Using Functions the Easy but Inefficient Way

One easy way to force complete Boolean evaluation is to write a function that computes the result of the Boolean expression and use complete Boolean evaluation within that function. The following C code implements this idea:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

/*
** Complete Boolean evaluation
** for the expression:
** i < g(y) || k > f(x)
*/

int func( void )
{
    int temp;
    int temp2;

    temp = i < g(y);
    temp2 = k > f(x);
    return temp || temp2;
}

int main( void )
{
    /*
    ** The following while loop
    ** uses complete Boolean evaluation
    */

    while( func() )
    {
      IntoIF:

        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the code that GCC (x86) emits for this C code (with a little cleanup to remove superfluous lines):

```
func:
.LFB0:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    y(%rip), %eax
        movl    %eax, %edi
        call    g
        movl    %eax, %edx
        movl    i(%rip), %eax
        cmpl    %eax, %edx
        setg    %al
        movzbl  %al, %eax
        movl    %eax, -8(%rbp)
        movl    x(%rip), %eax
        movl    %eax, %edi
        call    f
        movl    %eax, %edx
        movl    k(%rip), %eax
        cmpl    %eax, %edx
        setl    %al
        movzbl  %al, %eax
        movl    %eax, -4(%rbp)
        cmpl    $0, -8(%rbp)
        jne     .L2
        cmpl    $0, -4(%rbp)
        je      .L3
.L2:
        movl    $1, %eax
        jmp     .L4
.L3:
        movl    $0, %eax
.L4:
        leave
        ret
.LFE0:
        .size   func, .-func
        .section        .rodata
.LC0:
        .string "Hello"
        .text
        .globl  main
        .type   main, @function
```

```
main:
.LFB1:
        pushq   %rbp
        movq    %rsp, %rbp
        jmp     .L7
.L8:
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
.L7:
        call    func
        testl   %eax, %eax
        jne     .L8
        movl    $0, %eax
        popq    %rbp
        ret
```

As the assembly code demonstrates, the problem with this approach is that this code must make a function call and return (both of which are slow operations) in order to compute the value of the expression. For many expressions, the overhead of the call and return is more expensive than the actual computation of the expression's value.

14.1.1.2 Using Inline Functions

The previous approach definitely doesn't yield the greatest code you could obtain, in terms of either space or speed. If your compiler supports inline functions, you can produce a much better result by inlining `func()` in this example:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

inline int func( void )
{
    int temp;
    int temp2;

    temp = i < g(y);
    temp2 = k > f(x);
    return temp || temp2;
}

int main( void )
{
    while( func() )
    {
      IntoIF:

        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the conversion to (32-bit) x86 Gas assembly by the GCC compiler:

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %ecx
        andl    $-16, %esp
        .p2align 2,,3
.L2:
        subl    $12, %esp

; while( i < g(y) || k > f(x) )
;
; Compute g(y) into %EAX:

        pushl   y
        call    g
        popl    %edx
        xorl    %ebx, %ebx
        pushl   x

; See if i < g(y) and leave Boolean result
; in %EBX:

        cmpl    %eax, i
        setl    %bl

; Compute f(x) and leave result in %EAX:

        call    f                 ; Note that we call f, even if the
        addl    $16, %esp         ; above evaluates to true

; Compute k > f(x), leaving the result in %EAX.

        cmpl    %eax, k
        setg    %al

; Compute the logical OR of the above two expressions.

        xorl    %edx, %edx
        testl   %ebx, %ebx
        movzbl  %al, %eax
        jne     .L6
        testl   %eax, %eax
        je      .L7
.L6:
        movl    $1, %edx
.L7:
        testl   %edx, %edx
        je      .L10
.L8:

; Loop body:

        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        jmp     .L2
.L10:
        xorl    %eax, %eax
        movl    -4(%ebp), %ebx
        leave
        ret
```

As this example demonstrates, GCC compiles the function directly into the `while` loop's test, sparing this program the overhead associated with the function call and return.

14.1.1.3 Using Bitwise Logical Operations

In the C programming language, which supports Boolean operations on bits (also known as *bitwise logical operations*), you can use the same trick employed for the `if` statement to force complete Boolean evaluation—just use the bitwise operators. In the special case where the left and right operands of the `&&` or `||` operators are always `0` or `1`, you can use code like the following to force complete Boolean evaluation:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    // Use "|" rather than "||"
    // to force complete Boolean
    // evaluation here.

    while( i < g(y) | k > f(x) )
    {
        printf( "Hello" );
    }

    return( 0 );
}
```

Here's the assembly code that Borland C++ generates for this C source code:

```
_main   proc    near
?live1@0:
   ;
   ;    int main( void )
   ;
@1:
        push      ebx
        jmp       short @3 ; Skip to expr test.
   ;
   ;    {
   ;            while( i < g(y) | k > f(x) )
   ;            {
   ;                    printf( "Hello" );
   ;
@2:
        ; Loop body.

        push      offset s@
        call      _printf
        pop       ecx

; Here's where the test of the expression
; begins:

@3:
        ; Compute "i < g(y)" into ebx:

        mov       eax,dword ptr [_y]
        push      eax
        call      _g
        pop       ecx
        cmp       eax,dword ptr [_i]
        setg      bl
        and       ebx,1

        ;  Compute "k > f(x)" into EDX:

        mov       eax,dword ptr [_x]
        push      eax
        call      _f
        pop       ecx
        cmp       eax,dword ptr [_k]
        setl      dl
        and       edx,1
        ; Compute the logical OR of
        ; the two results above:

        or       ebx,edx

        ; Repeat loop body if true:

        jne      short @2
   ;
   ;          }
   ;
   ;          return( 0 );
   ;
        xor      eax,eax
   ;
   ;    }
   ;
@5:
@4:
        pop      ebx
        ret
_main   endp
```

As you can see in this 80x86 output, the compiler generates semantically equivalent code when using the bitwise logical operators. Just keep in mind that this code is valid only if you use `0` and `1` for the Boolean values `false` and `true`, respectively.

14.1.1.4 Using Unstructured Code

If you don't have inline function capability or if bitwise logical operators aren't available, you can use unstructured code to force complete Boolean evaluation as a last resort. The basic idea is to create an infinite loop and then write code to explicitly exit the loop if the condition fails. Generally, you'd use a `goto` statement (or a limited form of the `goto` statement like C's `break` or `continue` statements) to control loop termination. Consider the following example in C:

```
#include <stdio.h>

static int i;
static int k;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    int temp;
    int temp2;
    for( ;; )                //Infinite loop in C/C++
    {
        temp = i < g(y);
        temp2 = k > f(x);
        if( !temp && !temp2 ) break;
        printf( "Hello" );
    }

    return( 0 );
}
```

By using an infinite loop with an explicit break, we were able to compute the two components of the Boolean expression using separate C statements (hence, forcing the compiler to execute both subexpressions). Here's the code that the MSVC++ compiler produces:

```
main    PROC
; File c:\users\rhyde\test\t\t\t.cpp
; Line 16
$LN9:
        sub     rsp, 56                                 ; 00000038H

; Infinite loop jumps here:

$LN2@main:
; Line 21
;
; temp = i < g(y);
;
        mov     ecx, DWORD PTR ?y@@3HA                  ; y
        call    ?g@@YAHH@Z                              ; g

; compute i < g(y) and leave result in eax:

        cmp     DWORD PTR ?i@@3HA, eax
        jge     SHORT $LN5@main
        mov     DWORD PTR tv67[rsp], 1
        jmp     SHORT $LN6@main
$LN5@main:
        mov     DWORD PTR tv67[rsp], 0

$LN6@main:

; temp2 = k > f(x);

        mov     ecx, DWORD PTR ?x@@3HA                  ; x
        call    ?f@@YAHH@Z                              ; f

; compute k > f(x) and leave result in eax:

        cmp     DWORD PTR ?k@@3HA, eax
        jle     SHORT $LN7@main
        mov     DWORD PTR tv71[rsp], 1
        jmp     SHORT $LN8@main
$LN7@main:
        mov     DWORD PTR tv71[rsp], 0
$LN8@main:

; if( !temp && !temp2 ) break;

        or      ecx, eax
        mov     eax, ecx
        test    eax, eax
        je      SHORT $LN3@main
; Line 23
        lea     rcx, OFFSET FLAT:$SG6924
        call    printf

; Jump back to beginning of for(;;) loop.
;
; Line 24
        jmp     SHORT $LN2@main


$LN3@main:
; Line 26
        xor     eax, eax
; Line 27
        add     rsp, 56                                ; 00000038H
        ret     0
main    ENDP
```

As you can see, this program always evaluates both parts of the original Boolean expression (that is, you get complete Boolean evaluation).

You should be careful using unstructured code in this way. Not only is the result harder to read, but it's difficult to coerce the compiler into producing the code you want. Furthermore, code sequences that produce good code on one compiler won't produce comparable code on other compilers.

If your particular language doesn't support a statement like `break`, you can always use a `goto` statement to break out of the loop and achieve the same result. Although injecting `goto`s into your code isn't a great idea, in some cases it's your only option.

14.1.2 Forcing Short-Circuit Boolean Evaluation in a while Loop

Sometimes you need to guarantee short-circuit evaluation of the Boolean expression in a `while` statement even if the language (such as BASIC or Pascal) doesn't implement that scheme. For the `if` statement, you can force short-circuit evaluation by rearranging the way you compute the loop-control expression in your program. Unlike in the `if` statement, you can't use nested `while` statements or preface your `while` loop with other statements to force short-circuit evaluation, but it's still possible to do in most programming languages.

Consider the following C code fragment:

```
while( ptr != NULL && ptr->data != 0 )
{
    << loop body >>
    ptr = ptr->Next; // Step through a linked list.
}
```

This code could fail if C didn't guarantee short-circuit evaluation of the Boolean expression.

As with forcing complete Boolean evaluation, the easiest approach in a language like Pascal is to write a function that computes and returns the Boolean result using short-circuit Boolean evaluation. However, this scheme is relatively slow because of the high overhead of a function call. Consider the following Pascal example:[2]

```
program shortcircuit;
{$APPTYPE CONSOLE}
uses SysUtils;
var
    ptr     :Pchar;

    function shortCir( thePtr:Pchar ):boolean;
    begin

        shortCir := false;
        if( thePtr <> NIL ) then begin

            shortCir := thePtr^ <> #0;

        end; //if

    end;  // shortCircuit

begin

    ptr := 'Hello world';
    while( shortCir( ptr )) do begin

        write( ptr^ );
        inc( ptr );

    end; // while
    writeln;

end.
```

2. Delphi enables you to choose short-circuit or complete Boolean evaluation, so you wouldn't need to use this scheme with Delphi in reality. However, Delphi will compile this code, hence the use of its compiler for this example (though Free Pascal also works).

And now consider this 80x86 assembly code produced by Borland's Delphi compiler (and disassembled with IDAPro):

```
; function shortCir( thePtr:Pchar ):boolean
;
; Note: thePtr is passed into this function in
; the EAX register.

sub_408570  proc near

            ; EDX holds function return
            ; result (assume false).
            ;
            ; shortCir := false;

            xor     edx, edx

            ; if( thePtr <> NIL ) then begin

            test    eax, eax
            jz      short loc_40857C    ; branch if NIL

            ; shortCir := thePtr^ <> #0;

            cmp     byte ptr [eax], 0
            setnz   dl  ; DL = 1 if not #0

loc_40857C:

            ; Return result in EAX:

            mov     eax, edx
            retn
sub_408570  endp



; Main program (pertinent section):
;
; Load EBX with the address of the global "ptr" variable and
; then enter the "while" loop (Delphi moves the test for the
; while loop to the physical end of the loop's body):

                mov     ebx, offset loc_408628
                jmp     short loc_408617
; --------------------------------------------------------

loc_408600:
                ; Print the current character whose address
                ; "ptr" contains:

                mov     eax, ds:off_4092EC  ; ptr pointer
                mov     dl, [ebx]           ; fetch char
                call    sub_404523          ; print char
                call    sub_404391
                call    sub_402600

                inc     ebx               ; inc( ptr )

; while( shortCir( ptr )) do ...

loc_408617:
                mov     eax, ebx        ; Pass ptr in EAX
                call    sub_408570      ; shortCir
                test    al, al          ; Returns true/false
                jnz     short loc_408600 ; branch if true
```

The `sub_408570` procedure contains the function that will compute the short-circuit Boolean evaluation of an expression similar to the one appearing in the earlier C code. As you can see, the code that dereferences `thePtr` never executes if `thePtr` contains NIL (`0`).

If a function call is out of the question, then about the only reasonable solution is to use an unstructured approach. The following is a Pascal version of the `while` loop in the earlier C code that forces short-circuit Boolean evaluation:

```
    while( true ) do begin

        if( ptr = NIL ) then goto 2;
        if( ptr^.data = 0 ) then goto 2;
        << loop body >>
        ptr := ptr^.Next;

    end;
2:
```

Again, producing unstructured code, like the code in this example, is something you should do only as a last resort. But if the language (or compiler) you're using doesn't guarantee short-circuit evaluation and you need those semantics, unstructured code or inefficient code (using a function call) might be the only solution.

14.2 The repeat..until (do..until/do..while) Loop

Another common loop that appears in most modern programming languages is `repeat..until`. This loop tests for its terminating condition at the bottom of the loop. This means that the loop's body always executes at least once, even if the Boolean control expression evaluates to `false` on the first iteration of the loop. Although the `repeat..until` loop is a little less broadly applicable than the `while` loop, and you won't use it anywhere near as often, there are many situations where the `repeat..until` loop is the best choice of control structure for the job. Perhaps the classic example is reading input from the user until the user inputs a certain value. The following Pascal code fragment is very typical:

```
repeat

      write( 'Enter a value (negative quits): ');
      readln( i );
      // do something with i's value

until( i < 0 );
```

This loop always executes the body once. This, of course, is necessary because you must execute the loop's body to read the user-entered value, which the program checks to determine when loop execution is complete.

The `repeat..until` loop terminates when its Boolean control expression evaluates to `true` (rather than `false`, as for the `while` loop), as implied by the word *until*. Note, however, that this is a minor syntactical issue; the C/C++/Java/Swift languages (and many languages that share a C heritage) provide a `do..while` loop that repeats execution of the loop's body as long as the loop condition evaluates to `true`. From an efficiency point of view, there's absolutely no difference between these two loops, and you can easily convert one loop termination condition to the other by using your language's logical NOT operator. The following examples demonstrate the syntax of the Pascal, HLA, and C/C++ `repeat..until` and `do..while` loops. Here's the Pascal `repeat..until` loop example:

```
repeat

    (* Read a raw character from the "input" file, which in this case is the keyboard *)

    ch := rawInput( input );

    (* Save the character away. *)

    inputArray[ i ] := ch;
    i := i + 1;

    (* Repeat until the user hits the enter key *)

until( ch = chr( 13 ));
```

Now here's the C/C++ `do..while` version of the same loop:

```
do
{
    /* Read a raw character from the "input" file, which in this case is the keyboard */

    ch = getKbd();

    /* Save the character away. */
    inputArray[ i++ ] = ch;

    /* Repeat until the user hits the enter key */
}
while( ch != '\r' );
```

And here is the HLA `repeat..until` loop:

```
repeat

    // Read a character from the standard input device.

    stdin.getc();

    // Save the character away.

    mov( al, inputArray[ ebx ] );
    inc( ebx );

    // Repeat until the user hits the enter key.

until( al = stdio.cr );
```

Converting the `repeat..until` (or `do..while`) loop into assembly language is relatively easy and straightforward. All the compiler needs to do is substitute code for the Boolean loop control expression and branch back to the beginning of the loop's body if the expression evaluates affirmative (`false` for `repeat..until` or `true` for `do..while`). Here's the straightforward pure assembly implementation of the earlier HLA `repeat..until` loop (compilers for C/C++ and Pascal would generate nearly identical code for the other examples):

```
rptLoop:

    // Read a character from the standard input.

    call stdin.getc;

    // Store away the character.

    mov( al, inputArray[ ebx ] );
    inc( ebx );

    // Repeat the loop if the user did not hit
    // the enter key.

    cmp( al, stdio.cr );
    jne rptLoop;
```

As you can see, the code that a typical compiler generates for a `repeat..until` (or `do..while`) loop is usually a bit more efficient than the code you'll get for a regular `while` loop. Thus, you should consider using the repeat..until/do..while form if semantically possible. In many programs, the Boolean control expression always evaluates to true on the first iteration of some loop constructs. For example, it's not that uncommon to find a loop like the following in an application:

```
i = 0;
while( i < 100 )
{
      printf( "i: %d\n", i );
      i = i * 2 + 1;
      if( i < 50 )
      {
            i += j;
      }
}
```

This while loop is easily converted to a do..while loop as follows:

```
i = 0;
do
{
      printf( "i: %d\n", i );
      i = i * 2 + 1;
      if( i < 50 )
      {
            i += j;
      }
} while( i < 100 );
```

This conversion is possible because we know that i's initial value (0) is less than 100, so the loop's body always executes at least once.

As you've seen, you can help the compiler generate better code by using the more appropriate repeat..until/do..while loop rather than a regular while loop. Keep in mind, however, that the efficiency gain is small, so make sure you're not sacrificing readability or maintainability by doing so. Always use the most logically appropriate loop construct. If the body of the loop always executes at least once, you should use a repeat..until/do..while loop, even if a while loop would work equally well.

14.2.1 Forcing Complete Boolean Evaluation in a repeat..until Loop

Because the test for loop termination occurs at the bottom of the loop on a repeat..until (or do..while) loop, you force complete Boolean evaluation for it, similarly to how you do for an if statement. Consider the following C/C++ code:

```
extern int x;
extern int y;
extern int f( int );
extern int g( int );
extern int a;
extern int b;
int main( void )
{

    do
        {
            ++a;
            --b;
        }while( a < f(x) && b > g(y));

    return( 0 );
}
```

Here's the GCC output for the PowerPC (using short-circuit evaluation, which is standard for C) for the `do..while` loop:

```
L2:
        // ++a
        // --b

        lwz r9,0(r30)  ; get a
        lwz r11,0(r29) ; get b
        addi r9,r9,-1  ; --a
        lwz r3,0(r27)  ; Set up x parm for f
        stw r9,0(r30)  ; store back into a
        addi r11,r11,1 ; ++b
        stw r11,0(r29) ; store back into b

        ; compute f(x)

        bl L_f$stub    ; call f, result to R3

        ; is a >= f(x)? If so, quit loop

        lwz r0,0(r29)  ; get a
        cmpw cr0,r0,r3 ; Compare a with f's value
        bge- cr0,L3

        lwz r3,0(r28)  ; Set up y parm for g
        bl L_g$stub    ; call g

        lwz r0,0(r30)  ; get b
        cmpw cr0,r0,r3 ; Compare b with g's value
        bgt+ cr0,L2    ; Repeat if b > g's value
L3:
```

This program skips over the test for `b > g(y)` to label `L3` if the expression `a < f(x)` is false (that is, if `a >= f(x)`).

To force complete Boolean evaluation in this situation, our C source code needs to compute the subcomponents of the Boolean expression just prior to the `while` clause (keeping the results of the subexpressions in temporary variables) and then test only the results in the `while` clause:

```
static int a;
static int b;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    int temp1;
    int temp2;

    do
        {
            ++a;
            --b;
            temp1 = a < f(x);
            temp2 = b > g(y);
        }while( temp1 && temp2 );

    return( 0 );
}
```

Here's the conversion to PowerPC code by GCC:

```
L2:
        lwz r9,0(r30)    ; r9 = b
        li r28,1         ; temp1 = true
        lwz r11,0(r29)   ; r11 = a
        addi r9,r9,-1    ; --b
        lwz r3,0(r26)    ; r3 = x (set up f's parm)
        stw r9,0(r30)    ; Save b
        addi r11,r11,1   ; ++a
        stw r11,0(r29)   ; Save a
        bl L_f$stub      ; Call f
        lwz r0,0(r29)    ; Fetch a
        cmpw cr0,r0,r3   ; Compute temp1 = a < f(x)
        blt- cr0,L5      ; Leave temp1 true if a < f(x)
        li r28,0         ; temp1 = false
L5:
        lwz r3,0(r27)    ; r3 = y, set up g's parm
        bl L_g$stub      ; Call g
        li r9,1          ; temp2 = true
        lwz r0,0(r30)    ; Fetch b
        cmpw cr0,r0,r3   ; Compute b > g(y)
        bgt- cr0,L4      ; Leave temp2 true if b > g(y)
        li r9,0          ; Else set temp2 false
L4:
        ; Here's the actual termination test in
        ; the while clause:

        cmpwi cr0,r28,0
        beq- cr0,L3
        cmpwi cr0,r9,0
        bne+ cr0,L2
L3:
```

Of course, the actual Boolean expression (`temp1 && temp2`) still uses short-circuit evaluation, but only for the temporary variables created. The loop computes both of the original subexpressions regardless of the result of the first one.

14.2.2 Forcing Short-Circuit Boolean Evaluation in a repeat..until Loop

If your programming language provides a facility to break out of a `repeat..until` loop, such as C's `break` statement, then forcing short-circuit evaluation is fairly easy. Consider the C `do..while` loop from the previous section that forces complete Boolean evaluation:

```
do
{
    ++a;
    --b;
    temp1 = a < f(x);
    temp2 = b > g(y);

}while( temp1 && temp2 );
```

The following shows one way to convert this code so that it evaluates the termination expression using short-circuit Boolean evaluation:

```
static int a;
static int b;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    do
    {
        ++a;
        --b;

        if( !( a < f(x) )) break;
    } while( b > g(y) );

    return( 0 );
}
```

Here's the code that GCC emits for the PowerPC for the `do..while` loop in this code sequence:

```
L2:
        lwz r9,0(r30)   ; r9 = b
        lwz r11,0(r29)  ; r11 = a
        addi r9,r9,-1   ; --b
        lwz r3,0(r27)   ; Set up f(x) parm
        stw r9,0(r30)   ; Save b
        addi r11,r11,1  ; ++a
        stw r11,0(r29)  ; Save a
        bl L_f$stub     ; Call f

        ; break if !(a < f(x)):

        lwz r0,0(r29)
        cmpw cr0,r0,r3
        bge- cr0,L3

        ; while( b > g(y) ):

        lwz r3,0(r28)   ; Set up y parm
        bl L_g$stub     ; Call g
        lwz r0,0(r30)   ; Compute b > g(y)
        cmpw cr0,r0,r3
        bgt+ cr0,L2     ; Branch if true
L3:
```

If `a` is greater than or equal to the value that `f(x)` returns, this code immediately breaks out of the loop (at label `L3`) without testing to see if `b` is greater than the value `g(y)` returns. Hence, this code simulates short-circuit Boolean evaluation of the C/C++ expression `a < f(x) && b > g(y)`.

If the compiler you're using doesn't support a statement equivalent to C/C++'s `break` statement, you'll have to use slightly more sophisticated logic. Here's one way to do that:

```
static int a;
static int b;

extern int x;
extern int y;
extern int f( int );
extern int g( int );

int main( void )
{
    int temp;

    do
    {
        ++a;
        --b;
        temp = a < f(x);
        if( temp )
        {
            temp = b > g(y);
        };
    }while( temp );

    return( 0 );
}
```

And here's the PowerPC code that GCC produces for this example:

```
L2:
        lwz r9,0(r30)   ; r9 = b
        lwz r11,0(r29)  ; r11 = a
        addi r9,r9,-1   ; --b
        lwz r3,0(r27)   ; Set up f(x) parm
        stw r9,0(r30)   ; Save b
        addi r11,r11,1  ; ++a
        stw r11,0(r29)  ; Save a
        bl L_f$stub     ; Call f
        li r9,1         ; Assume temp is true
        lwz r0,0(r29)   ; Set temp false if
        cmpw cr0,r0,r3  ; a < f(x)
        blt- cr0,L5
        li r9,0
L5:
        cmpwi cr0,r9,0  ; If !(a < f(x)) then bail
        beq- cr0,L10    ; on the do..while loop
        lwz r3,0(r28)   ; Compute temp = b > f(y)
        bl L_g$stub     ; using a code sequence
        li r9,1         ; that is comparable to
        lwz r0,0(r30)   ; the above.
        cmpw cr0,r0,r3
        bgt- cr0,L9
        li r9,0
L9:
        ; Test the while termination expression:

        cmpwi cr0,r9,0
        bne+ cr0,L2
L10:
```

Although these examples have been using the conjunction operation (logical AND), using the disjunction operator (logical OR) is just as easy. To close off this section, consider this Pascal sequence and its conversion:

```
repeat

      a := a + 1;
      b := b - 1;

until( (a < f(x)) OR (b > g(y)) );
```

Here's the conversion to force complete Boolean evaluation:

```
repeat

      a := a + 1;
      b := b - 1;
      temp := a < f(x);
      if( not temp ) then begin

            temp := b > g(y);

    end;
until( temp );
```

Here's the code that Borland's Delphi produces for the two loops (assuming you select *complete Boolean evaluation* in the compiler's options):

```
;    repeat
;
;        a := a + 1;
;        b := b - 1;
;
;    until( (a < f(x)) or (b > g(y)));

loc_4085F8:
                inc     ebx                     ; a := a + 1;
                dec     esi                     ; b := b - 1;
                mov     eax, [edi]              ; EDI points at x
                call    locret_408570
                cmp     ebx, eax                ; Set AL to 1 if
                setl    al                      ; a < f(x)
                push    eax                     ; Save Boolean result.

                mov     eax, ds:dword_409288 ; y
                call    locret_408574           ; g(6)

                cmp     esi, eax                ; Set AL to 1 if
                setnle  al                      ; b > g(y)
                pop     edx                     ; Retrieve last value.
                or      dl, al                  ; Compute their OR
                jz      short loc_4085F8        ; Repeat if false.

;    repeat
;
;        a := a + 1;
;        b := b - 1;
;        temp := a < f(x);
;        if( not temp ) then begin
;
;            temp := b > g(y);
;
;        end;
;
;    until( temp );
```

```
loc_40861B:
                inc     ebx                 ; a := a + 1;
                dec     esi                 ; b := b - 1;
                mov     eax, [edi]          ; Fetch x
                call    locret_408570       ; call f
                cmp     ebx, eax            ; is a < f(x)?
                setl    al                  ; Set AL to 1 if so.

            ; If the result of the above calculation is
            ; true, then don't bother with the second
            ; test (that is, short-circuit evaluation)

                test    al, al
                jnz     short loc_40863C

            ; Now check to see if b > g(y)

                mov     eax, ds:dword_409288
                call    locret_408574

            ; Set AL = 1 if b > g(y):

                cmp     esi, eax
                setnle  al

; Repeat loop if both conditions were false:

loc_40863C:
                test    al, al
                jz      short loc_40861B
```

The code that the Delphi compiler generates for this forced short-circuit evaluation is nowhere near as good as the code it would generate if you allowed it to do this job for you. Here's the Delphi code with the *complete Boolean evaluation* option unselected (that is, instructing Delphi to use short-circuit evaluation):

```
loc_4085F8:
                inc     ebx
                dec     esi
                mov     eax, [edi]
                call    nullsub_1 ;f
                cmp     ebx, eax
                jl      short loc_408613
                mov     eax, ds:dword_409288
                call    nullsub_2 ;g
                cmp     esi, eax
                jle     short loc_4085F8
```

While this trick is useful for forcing short-circuit evaluation when the compiler does not support it, this latter Delphi example reiterates that you should use the compiler's facilities if at all possible—you'll generally get better machine code.

14.3 The forever..endfor Loop

The while loop tests for loop termination at the beginning (top) of the loop. The repeat..until loop tests for loop termination at the end (bottom) of the loop. The only place left to test for loop termination is somewhere in the middle of the loop's body. The forever..endfor loop, along with some special loop termination statements, handles this case.

Most modern programming languages provide a while loop and a repeat..until loop (or their equivalents). Interestingly enough, only a few modern imperative programming languages provide an explicit forever..endfor loop.[3] This is especially surprising because the forever..endfor loop (along with a loop termination test) is actually the most general of the three forms. You can easily synthesize a while loop or a repeat..until loop from a single forever..endfor loop.

Fortunately, it's easy to create a simple forever..endfor loop in any language that provides a while loop or a repeat..until/do..while loop. All you need do is supply a Boolean control expression that always evaluates to false for repeat..until or true for do..while. In Pascal, for example, you could use code such as the following:

```
const
    forever = true;
        .
        .
        .
    while( forever ) do begin

        << code to execute in an infinite loop >>

    end;
```

The big problem with standard Pascal is that it doesn't provide a mechanism (other than a generic goto statement) for explicitly breaking out of a loop. Fortunately, many modern Pascals, like Delphi and Free Pascal, provide a statement like break to immediately exit the current loop.

Although the C/C++ language does not provide an explicit statement that creates a forever loop, the syntactically bizarre for(;;) statement has served this purpose since the very first C compiler was written. Therefore, C/C++ programmers can create a forever..endfor loop as follows:

```
for(;;)
{
    << code to execute in an infinite loop >>
}
```

3. Ada provides one, as do C and C++ (the for(;;) loop).

C/C++ programmers can use C's break statement (along with an if statement) to place a loop termination condition in the middle of a loop, like so:

```
for(;;)
{
    << Code to execute (at least once)
       prior to the termination test >>

    if( termination_expression ) break;

    << Code to execute after the loop termination test >>
}
```

The HLA language provides an explicit (high-level) forever..endfor statement (along with a break and a breakif statement) that lets you terminate the loop somewhere in the middle. This HLA forever..endfor loop tests for loop termination in the middle of the loop:

```
forever

    << Code to execute (at least once) prior to
       the termination test >>

    breakif( termination_expression );

    << Code to execute after the loop termination test >>

endfor;
```

Converting a forever..endfor loop into pure assembly language is trivial—all you need is a single jmp instruction that can transfer control from the bottom of the loop back to the top of the loop. The implementation of the break statement is just as simple: it's just a jump (or conditional jump) to the first statement following the loop. The following two code fragments demonstrate an HLA forever..endfor loop (along with a breakif) and the corresponding "pure" assembly code:

```
// High-level forever statement in HLA:

forever

    stdout.put
    (
     "Enter an unsigned integer less than five:"
    );
    stdin.get( u );
    breakif( u < 5);
    stdout.put
    (
      "Error: the value must be between zero and five" nl
    );
```

```
endfor;

// Low-level coding of the forever loop in HLA:

foreverLabel:
    stdout.put
    (
      "Enter an unsigned integer less than five:"
    );
    stdin.get( u );
    cmp( u, 5 );
    jbe endForeverLabel;
    stdout.put
    (
      "Error: the value must be between zero and five" nl
    );
    jmp foreverLabel;

endForeverLabel:
```

Of course, you can also rotate this code to create a slightly more efficient version:

```
// Low-level coding of the forever loop in HLA
// using code rotation:

jmp foreverEnter;
foreverLabel:
        stdout.put
        (
          "Error: the value must be between zero and five"
          nl
        );
    foreverEnter:
        stdout.put
        (
          "Enter an unsigned integer less "
          "than five:"
        );
        stdin.get( u );
        cmp( u, 5 );
        ja foreverLabel;
```

If the language you're using doesn't support a `forever..endfor` loop, any decent compiler will convert a `while(true)` statement into a single jump instruction. If your compiler doesn't do so, then it does a poor job of optimization, and any attempts to manually optimize the code are a lost cause. For reasons you'll soon see, you shouldn't try to create the `forever..endfor` loop using a `goto` statement.

14.3.1 Forcing Complete Boolean Evaluation in a forever Loop

Because you exit from a `forever` loop using an `if` statement, the techniques for forcing complete Boolean evaluation when exiting a `forever` loop are the same as for an `if` statement. See "Forcing Complete Boolean Evaluation in an `if` Statement" on page 465 for details.

14.3.2 Forcing Short-Circuit Boolean Evaluation in a forever Loop

Likewise, because you exit from a `forever` loop using an `if` statement, the techniques for forcing short-circuit Boolean evaluation when exiting a `forever` loop are the same as for a `repeat..until` statement. See "Forcing Short-Circuit Boolean Evaluation in a `repeat..until` Loop" on page 524 for details.

14.4 The Definite Loop (for Loops)

The `forever..endfor` loop is an *infinite* loop (assuming you don't break out of it via a `break` statement). The `while` and `repeat..until` loops are examples of *indefinite* loops because, in general, the program cannot determine how many iterations they will execute when it first encounters them. For a *definite* loop, on the other hand, the program can determine exactly how many iterations the loop will repeat prior to executing the first statement of the loop's body. A good example of a definite loop in a traditional HLL is Pascal's `for` loop, which uses the following syntax:

```
for variable := expr1 to expr2 do
        statement
```

which iterates over the range `expr1..expr2` if `expr1` is less than or equal to `expr2`, or

```
for variable := expr1 downto expr2 do
        statement
```

which iterates over the range `expr1..expr2` if `expr1` is greater than or equal to `expr2`. Here's a typical example of a Pascal `for` loop:

```
for i := 1 to 10 do
    writeln( 'hello world');
```

This loop always executes exactly 10 times; hence, it's a definite loop. However, this doesn't imply that a compiler has to be able to determine the number of loop iterations at compile time. Definite loops also allow the use of expressions that force the program to determine the number of iterations at runtime. For example:

```
write( 'Enter an integer:');
readln( cnt );
for i := 1 to cnt do
    writeln( 'Hello World');
```

The Pascal compiler cannot determine the number of iterations this loop will execute. In fact, because the number of iterations is dependent upon user input, it could vary each time this loop executes in a single execution of the enclosing program. However, the program can determine exactly how many iterations the loop will execute, indicated by the value in the `cnt` variable, whenever it encounters this loop. Note that Pascal (like most languages that support definite loops) expressly forbids code such as the following:

```
for i := 1 to j do begin

    << some statements >>
    i := <<some value>>;
    << some other statements >>

end;
```

You are not allowed to change the value of the loop control variable during the execution of the loop's body. In this example, should you try to change the `for` loop's control variable, a high-quality Pascal compiler will detect that attempt and report an error. Also, a definite loop computes the starting and ending values only once. Therefore, if the `for` loop modifies a variable that appears as the second expression, it does not reevaluate the expression on each iteration of the loop. For example, if the body of the `for` loop in the previous example modifies the value of `j`, this will not affect the number of loop iterations.[4]

Definite loops have certain special properties that allow a (good) compiler to generate better machine code. In particular, because the compiler can determine how many iterations the loop will execute prior to executing the first statement of the loop's body, the compiler can often dispense with complex tests for loop termination and simply decrement a register down to `0` to control the number of loop iterations. The compiler can also use induction to optimize access to the loop control variable in a definite loop (see the description of induction in "Optimization of Arithmetic Statements" on page 397).

C/C++/Java users should note that the `for` loop in these languages is not a true definite loop; rather, it is a special case of the indefinite `while` loop. Most good C/C++ compilers will attempt to determine if a `for` loop is a definite loop and, if so, they'll generate decent code. You can help your compiler by following these guidelines:

- Your C/C++ `for` loops should use the same semantics as the definite (`for`) loops in languages such as Pascal. That is, the `for` loop should initialize a single loop control variable, test for loop termination when that value is less than or greater than some ending value, and increment or decrement the loop control variable by 1.

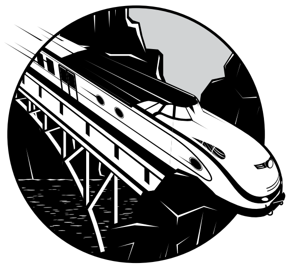
4. Of course, some compilers might actually recompute this on each iteration, but the Pascal language standard doesn't require this; indeed, the standard suggests that these values shouldn't change during the execution of the loop body.

- Your C/C++ `for` loops should not modify the value of the loop control variable within the loop.
- The test for loop termination remains static over the execution of the loop's body. That is, the loop body should not be able to change the termination condition (which, by definition, would make the loop an indefinite loop). For example, if the loop termination condition is `i < j`, the loop body should not modify the value of `i` or `j`.
- The loop body does not pass the loop control variable or any variable appearing in the loop termination condition by reference to a function if that function modifies the actual parameter.

14.5 For More Information

"For More Information" on page 501 applies to this chapter as well. Please see that section for more details.

15
FUNCTIONS AND PROCEDURES



Since the beginning of the structured programming revolution in the 1970s, subroutines (procedures and functions) have been one of the primary tools software engineers use to organize, modularize, and otherwise structure their programs. Because procedure and function calls are used so frequently in code, CPU manufacturers have attempted to make them as efficient as possible. Nevertheless, these calls—and their associated returns—have costs that many programmers don't consider when creating functions, and using them inappropriately can greatly increase a program's size and execution time. This chapter discusses those costs and how to avoid them, covering the following subjects:

- Function and procedure calls
- Macros and inline functions
- Parameter passing and calling conventions
- Activation records and local variables

- Parameter-passing mechanisms
- Function return results

By understanding these topics, you can avoid the efficiency pitfalls that are common in modern programs that make heavy use of procedures and functions.

15.1 Simple Function and Procedure Calls

Let's begin with some definitions. A *function* is a section of code that computes and returns some value—the function result. A *procedure* (or *void function,* in C/C++/Java/Swift terminology) simply accomplishes some action. Function calls generally appear within an arithmetic or logical expression, while procedure calls look like statements in the programming language. For the purpose of this discussion, you can generally assume that a procedure call and a function call are the same, and use the terms *function* and *procedure* interchangeably. For the most part, a compiler implements procedure and function calls identically.

NOTE *Functions and procedures do have some differences, however. Namely, there are some efficiency issues related to function results, which we'll consider in "Function Return Values" on page 590.*

With most CPUs, you invoke procedures via an instruction similar to the 80x86 `call` (`branch` and `link` on the ARM and PowerPC) and return to the caller using the `ret` (return) instruction. The `call` instruction performs three discrete operations:

1. It determines the address of the instruction to execute upon returning from the procedure (this is usually the instruction immediately following `call`).
2. It saves this address (commonly known as the *return address* or *link address*) into a known location.
3. It transfers control (via a jump mechanism) to the first instruction of the procedure.

Execution starts with the first instruction of the procedure and continues until the CPU encounters a `ret` instruction, which fetches the return address and transfers control to the machine instruction at that address. Consider the following C function:

```
#include <stdio.h>

void func( void )
{
    return;
}
```

```
int main( void )
{
    func();
    return( 0 );
}
```

Here's the conversion to PowerPC code by GCC:

```
_func:
        ; Set up activation record for function.
        ; Note R1 is used as the stack pointer by
        ; the PowerPC ABI (application binary
        ; interface, defined by IBM).

        stmw r30,-8(r1)
        stwu r1,-48(r1)
        mr r30,r1

        ; Clean up activation record prior to the return

        lwz r1,0(r1)
        lmw r30,-8(r1)

        ; Return to caller (branch to address
        ; in the link register):

        blr

_main:
        ; Save return address from
        ; main program (so we can
        ; return to the OS):

        mflr r0
        stmw r30,-8(r1) ; Preserve r30/31
        stw r0,8(r1)    ; Save rtn adrs
        stwu r1,-80(r1) ; Update stack for func()
        mr r30,r1       ; Set up frame pointer

        ; Call func:

        bl _func

        ; Return 0 as the main
        ; function result:

        li r0,0
        mr r3,r0
        lwz r1,0(r1)
        lwz r0,8(r1)
        mtlr r0
        lmw r30,-8(r1)
        blr
```

Here's the 32-bit ARM version of this source code compiled by GCC:

```
func:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.

    str fp, [sp, #-4]!  @ Save frame pointer on stack
    add fp, sp, #0
    nop
    add sp, fp, #0
    @ sp needed
    ldr fp, [sp], #4    @ Load FP from stack.
    bx  lr              @ Return from subroutine

main:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0

    push    {fp, lr}    @ Save FP and return address

    add fp, sp, #4      @ Set up FP
    bl  func            @ Call func
    mov r3, #0          @ main return value = 0
    mov r0, r3

    @ Note that popping PC returns to Linux
    pop {fp, pc}
```

And here's the conversion of the same source code to 80x86 code by GCC:

```
func:
.LFB0:
    pushq   %rbp
    movq    %rsp, %rbp
    nop
    popq    %rbp
    ret

main:
.LFB1:
    pushq   %rbp
    movq    %rsp, %rbp
    call    func
    movl    $0, %eax
    popq    %rbp
    ret
```

As you can see, the 80x86, ARM, and PowerPC devote considerable effort to building and managing activation records (see "The Stack Section" on page 179). The important things to see in these two assembly language sequences are the `bl _func` and `blr` instructions in the PowerPC code; `bl func` and `bx lr` instructions in the ARM code; and the `call func` and `ret` instructions in the 80x86 code. These are the instructions that call the function and return from it.

15.1.1 Return Address Storage

But where, exactly, does the CPU store the return address? In the absence of recursion and certain other program control constructs, the CPU could store the return address in any location that is large enough to hold the address and that will still contain that address when the procedure returns to its caller. For example, the program could choose to store the return address in a machine register (in which case the return operation would consist of an indirect jump to the address contained in that register). One problem with using registers, however, is that CPUs generally have a limited number of them. This means every register that holds a return address is unavailable for other purposes. For this reason, on CPUs that save the return address in a register, the applications usually move the return address to memory so they can reuse that register.

Consider the PowerPC and ARM `bl` (branch and link) instruction. This instruction transfers control to the target address specified by its operand and copies the address of the instruction following `bl` into the LINK register. Inside a procedure, if no code modifies the value of the LINK register, the procedure can return to its caller by executing a PowerPC `blr` (branch to LINK register) or ARM `bx` (branch and exchange) instruction. In our trivial example, the `func()` function does not execute any code that modifies the value of the LINK register, so this is exactly how `func()` returns to its caller. However, if this function had used the LINK register for some other purpose, it would have been the procedure's responsibility to save the return address so that it could restore the value prior to returning via a `blr` instruction at the end of the function call.

A more common place to keep return addresses is in memory. Although accessing memory on most modern processors is much slower than accessing a CPU register, keeping return addresses in memory allows a program to have a large number of nested procedure calls. Most CPUs actually use a *stack* to hold return addresses. For example, the 80x86 `call` instruction *pushes* the return address onto a stack data structure in memory, and the `ret` instruction *pops* this return address off the stack. Using a stack of memory locations to hold return addresses offers several advantages:

- Stacks, because of their *last-in, first-out (LIFO)* organization, fully support nested procedure calls and returns as well as recursive procedure calls and returns.
- Stacks are memory efficient because they reuse the same memory locations for different procedure return addresses (rather than requiring a separate memory location to hold each procedure's return address).

- Even though stack access is slower than register access, the CPU can generally access memory locations on the stack faster than separate return addresses elsewhere, because the CPU frequently accesses the stack and the stack contents tend to remain in the cache.
- As discussed in Chapter 7, stacks are also great places to store activation records (such as parameters, local variables, and other procedure state information).

Using a stack also incurs a few penalties, though. Most importantly, maintaining a stack generally requires dedicating a CPU register to keep track of it in memory. This could be a register that the CPU explicitly dedicates for this purpose (for example, the RSP register on the x86-64 or R14/SP on the ARM) or a general-purpose register on a CPU that doesn't provide explicit hardware stack support (for example, applications running on the PowerPC processor family typically use R1 for this purpose).

On CPUs that provide a hardware stack implementation and a `call`/`ret` instruction pair, making a procedure call is easy. As shown earlier in the 80x86 GCC example output, the program simply executes a `call` instruction to transfer control to the beginning of the procedure and then executes a `ret` instruction to return from the procedure.

The PowerPC/ARM approach, using a "branch and link" instruction might seem less efficient than the `call`/`ret` mechanism. While it's certainly true that the "branch and link" approach requires a little more code, it isn't so clear that it's slower than the `call`/`ret` approach. A `call` instruction is a complex instruction (accomplishing several independent tasks with a single instruction) and, as a result, typically requires several CPU clock cycles to execute. The execution of the `ret` instruction is similar. Whether the extra overhead is costlier than maintaining a software stack varies by CPU and compiler. However, a "branch and link" instruction and an indirect jump through the link address, without the overhead of maintaining the software stack, is usually faster than the corresponding `call`/`ret` instruction pair. If a procedure doesn't call any other procedures and can maintain parameters and local variables in machine registers, it's possible to skip the software stack maintenance instructions altogether. For example, the call to `func()` in the previous example is probably more efficient on the PowerPC and ARM than on the 80x86, because `func()` doesn't need to save the LINK register's value into memory—it simply leaves that value in LINK throughout the execution of the function.

Because many procedures are short and have few parameters and local variables, a good RISC compiler can often dispense with the software stack maintenance entirely. Therefore, for many common procedures, this RISC approach is faster than the CISC (`call`/`ret`) approach; however, that's not to imply that it's always better. The brief example in this section is a very special case. In our simple demonstration program, the function that this code calls—via the `bl` instruction—is near the `bl` instruction. In a complete application, `func()` might be *very* far away, and the compiler wouldn't be able to encode the target address as part of the instruction. That's because RISC processors (like the PowerPC and ARM) must encode their entire instruction within a single 32-bit value (which must include both the opcode and the displacement to the function). If `func()` is farther away than can be encoded in the remaining displacement bits (24, in the case of the PowerPC and ARM `bl` instructions), the compiler has to emit a sequence of instructions that will compute the address of the target routine and indirectly transfer control through that address. Most of the time, this shouldn't be a problem. After all, few programs are so large that the functions would be outside this range (64MB, in the case of the PowerPC, ±32MB for the ARM). However, there's a very common case where GCC (and other compilers, presumably) must generate this type of code: when the compiler doesn't know the target address of the function, because it's an external symbol that the linker must merge in after compilation is complete. Because the compiler doesn't know where the routine will be sitting in memory (and also because most linkers work only with 32-bit addresses, not 24-bit displacement fields), the compiler must assume that the function's address is out of range and emit the long version of the function call. Consider the following slight modification to the earlier example:

```
#include <stdio.h>

extern void func( void );

int main( void )
{
    func();

    return( 0 );
}
```

This code declares `func()` as an external function. Now look at the PowerPC code that GCC produces and compare it with the earlier code:

```
.text
        .align 2
        .globl _main
_main:
        ; Set up main's activation record:

        mflr r0
        stw r0,8(r1)
        stwu r1,-80(r1)

        ; Call a "stub" routine that will
        ; do the real call to func():

        bl L_func$stub

        ; Return 0 as Main's function
        ; result:

        lwz r0,88(r1)
        li r3,0
        addi r1,r1,80
        mtlr r0
        blr


; The following is a stub that calls the
; real func() function, wherever it is in
; memory.

        .data
        .picsymbol_stub
L_func$stub:
        .indirect_symbol _func

        ; Begin by saving the LINK register
        ; value in R0 so we can restore it
        ; later.

        mflr r0

        ; The following code sequence gets
        ; the address of the L_func$lazy_ptr
        ; pointer object into R12:

        bcl 20,31,L0$_func      ; R11<-adrs(L0$func)
L0$_func:
        mflr r11
        addis r11,r11,ha16(L_func$lazy_ptr-L0$_func)

        ; Restore the LINK register (used by the
        ; preceding code) from R0:

        mtlr r0

        ; Compute the address of func() and move it
        ; into the PowerPC COUNT register:

        lwz r12,lo16(L_func$lazy_ptr-L0$_func)(r11)
        mtctr r12

        ; Set up R11 with an environment pointer:

        addi r11,r11,lo16(L_func$lazy_ptr-L0$_func)

        ; Branch to address held in the COUNT
        ; register (that is, to func):

        bctr

; The linker will initialize the following
; dword (.long) value with the address of
; the actual func() function:

        .data
        .lazy_symbol_pointer
```

```
L_func$lazy_ptr:
        .indirect_symbol _func
        .long dyld_stub_binding_helper
```

This code effectively winds up calling two functions in order to call `func()`. First, it calls a *stub* function (`L_func$stub`), which then transfers control to the actual `func()` routine. Clearly there is considerable overhead here. Without actually benchmarking the PowerPC code against the 80x86 code, it's probably a safe bet that the 80x86 solution is a bit more efficient. (The 80x86 version of the GCC compiler emits the same code for the main program as in the earlier example, even when compiling in the external reference.) You'll soon see that the PowerPC also generates stub functions for things other than external functions. Therefore, the CISC solution often is more efficient than the RISC solution (presumably, RISC CPUs make up the difference in performance in other areas).

The Microsoft CLR also provides generic call and return functionality. Consider the following C# program with a static function `f()`:

```
using System;

namespace Calls_f
{
     class program
      {
        static void f()
        {
            return;
        }
        static void Main( string[] args)
        {
            f();
        }
      }
}
```

Here's the CIL code that the Microsoft C# compiler emits for functions `f()` and `Main()`:

```
.method private hidebysig static void  f() cil managed
{
  // Code size       4 (0x4)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  br.s       IL_0003
  IL_0003:  ret
} // end of method program::f


.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  call       void Calls_f.program::f()
  IL_0006:  nop
  IL_0007:  ret
} // end of method program::Main
```

As one last example, here's a comparable Java program:

```
public class Calls_f
{
    public static void f()
    {
        return;
    }

    public static void main( String[] args )
    {
        f();
    }
}
```

Here's the Java bytecode (JBC) output:

```
Compiled from "Calls_f.java"
public class Calls_f extends java.lang.Object{
public Calls_f();
  Code:
   0:   aload_0
        //call Method java/lang/Object."<init>":()
   1:   invokespecial   #1;
   4:   return

public static void f();
  Code:
   0:   return

public static void main(java.lang.String[]);
  Code:
   0:   invokestatic    #2; //Method f:()
   3:   return

}
```

Note that the Microsoft CLR and Java VM both have several variants of call and invoke instructions. These simple examples demonstrate calls to static methods.

15.1.2 Other Sources of Overhead

Of course, a typical procedure call and return involve overhead beyond the execution of the actual procedure `call` and `return` instructions. Prior to calling the procedure, the calling code must compute and pass any parameters to it. Upon entry into the procedure, the calling code may also need to complete the construction of the *activation record* (that is, allocate space for local variables). The costs of these operations vary by CPU and compiler. For example, if the calling code can pass parameters in registers rather than on the stack (or some other memory location), this is usually more efficient. Similarly, if the procedure can keep all its local variables in registers rather than in the activation record on the stack, accessing those local variables is much more efficient. This is one area where RISC processors have a considerable advantage over CISC processors. A typical RISC compiler can reserve several registers for passing parameters and local variables. (RISC processors typically have 16, 32, or more general-purpose registers, so setting aside several registers for this purpose is not outrageous.) For procedures that don't call any other procedures (discussed in the next section), there's no need to preserve these register values, so parameter and local variable access is very efficient. Even on CPUs with a limited number of registers (such as the 32-bit 80x86), it's still possible to pass a small number of parameters, or maintain a few local variables, in registers. Many 80x86 compilers, for example, will keep up to three values (parameters or local variables) in the registers. Clearly, though, the RISC processors have an advantage here.[1]

Armed with this knowledge, along with the background on activation records and stack frames from earlier in this book (see "The Stack Section" on page 179), we can now discuss how to write procedures and functions that operate as efficiently as possible. The exact rules are highly dependent upon your CPU and the compiler you're using, but some of the concepts are generic enough to apply to all programs. The following sections assume that you're writing for an 80x86 or ARM CPU (as most of the world's software runs on one of these two CPUs).

15.2 Leaf Functions and Procedures

Compilers can often generate better code for *leaf* procedures and functions—that is, those that don't call other procedures or functions. The metaphor comes from a graphical representation of procedure/function invocations known as a *call tree*. A call tree consists of a set of circles (*nodes*) that represent the functions and procedures in a program. An arrow from one node to another implies that the first node contains a call to the second. Figure 15-1 illustrates a typical call tree.

In this example, the main program directly calls procedure `prc1()` and functions `fnc1()` and `fnc2()`. Function `fnc1()` directly calls procedure `prc2()`. Function `fnc2()` directly calls procedures `prc2()` and `prc3()` as well

1. The 80x86's saving grace is that the CPU runs so much faster than typical RISC devices, so it can afford to execute a few more instructions or execute instructions that take multiple clock cycles, and it will still run faster than contemporary RISC CPUs. This is a paradox because the whole purpose of RISC design in the first place was to create a CPU that could run at a higher clock frequency, even if it took more instructions to accomplish the same thing as a CISC CPU.

as function `fnc3()`. The leaf procedures and functions in this call tree are `prc1()`, `prc2()`, `fnc3()`, and `prc3()`, which do not call any other procedures or functions.

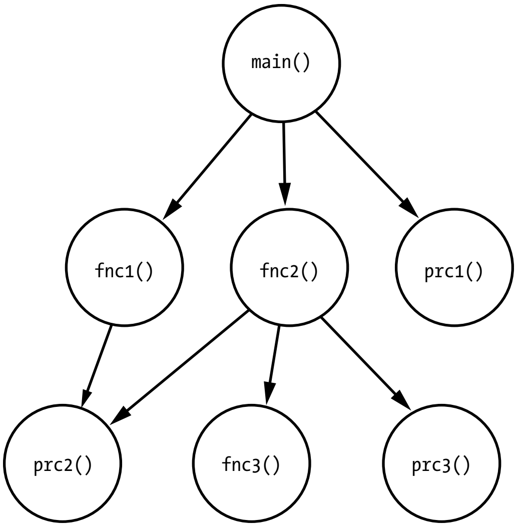


Figure 15-1: A call tree

Working with leaf procedures and functions has an advantage: they do not need to save parameters passed to them in registers or preserve the values of local variables they maintain in registers. For example, if `main()` passes two parameters to `fnc1()` in the EAX and EDX registers, and `fnc1()` passes a different pair of parameters to `prc2()` in EAX and EDX, then `fnc1()` must first save the values it found in EAX and EDX before calling `prc2()`. The `prc2()` procedure, on the other hand, doesn't have to save the values in EAX and EDX prior to some procedure or function call, because it doesn't make such calls. In a similar vein, if `fnc1()` allocates any local variables in registers, then it will need to preserve those registers across a call to `prc2()`, because `prc2()` can use the registers for its own purposes. By contrast, if `prc2()` uses a register for a local variable, it never has to preserve the variable's value, because it never calls any subroutines. Therefore, good compilers tend to generate better code for leaf procedures and functions because they don't have to preserve the register values.

One way to *flatten* the call tree is to take the code associated with procedures and functions in interior nodes and inline it into functions higher in the call tree. In Figure 15-1, for example, if it is practical to move the code for `fnc1()` into `main()`, you don't need to save and restore registers (among other operations). However, be sure you're not sacrificing readability and maintainability when flattening the call tree. You want to avoid writing procedures and functions that simply call other procedures and functions without doing any work on their own, but you don't want to destroy the modularity of your application's design by expanding function and procedure calls throughout your code.

You've already seen that having a leaf function is handy when you're using a RISC processor, like the PowerPC or ARM, that uses a "branch and link" instruction to make a subroutine call. The PowerPC and ARM LINK registers are good examples of registers that you have to preserve across procedure calls. Because a leaf procedure does not (normally) modify the value in the LINK register, no extra code is necessary in a leaf procedure to preserve that register's value. To see the benefits of calling leaf functions on a RISC CPU, consider the following C code:

```
void g( void )
{
    return;
}

void f( void )
{
    g();
    g();
    return;
}

int main( void )
{
    f();
    return( 0 );
}
```

GCC emits the following PowerPC assembly code:

```
; g's function code:

_g:
        ; Set up g's environment
        ; (set up activation record):

        stmw r30,-8(r1)
        stwu r1,-48(r1)
        mr r30,r1

        ; Tear down the activation
        ; record.

        lwz r1,0(r1)
        lmw r30,-8(r1)

        ; Return to caller via LINK:

        blr

; f's function code:

_f:
        ; Set up activation record,
        ; including saving the value
        ; of the LINK register:

        mflr r0         ; R0 = LINK
        stmw r30,-8(r1)
        stw r0,8(r1)    ; Save LINK
        stwu r1,-80(r1)
        mr r30,r1

        ; Call g (twice):

        bl _g
        bl _g

        ; Restore LINK from the
        ; activation record and
        ; then clean up activation
        ; record:

        lwz r1,0(r1)
        lwz r0,8(r1)    ; R0 = saved adrs
        mtlr r0         ; LINK = R0
        lmw r30,-8(r1)

        ; Return to main function:

        blr

; Main function code:

_main:
        ; Save main's return
        ; address into main's
        ; activation record:

        mflr r0
        stmw r30,-8(r1)
        stw r0,8(r1)
        stwu r1,-80(r1)
        mr r30,r1

        ; Call the f function:

        bl _f

        ; Return 0 to whomever
        ; called main:

        li r0,0
        mr r3,r0
        lwz r1,0(r1)
        lwz r0,8(r1)    ; Move saved return
        mtlr r0         ; address to LINK
        lmw r30,-8(r1)
        ; Return to caller:

        blr
```

There's an important difference between the implementations of the `f()` and `g()` functions in this PowerPC code—`f()` has to preserve the value of the LINK register, whereas `g()` does not. Not only does this involve extra instructions, but it also involves accessing memory, which can be slow.

Another advantage to using leaf procedures, which isn't obvious from the call tree, is that constructing their activation record requires less work. On the 80x86, for example, a good compiler doesn't have to preserve the value of the EBP register, load EBP with the activation record address, and then restore the original value by accessing local objects via the stack pointer register (ESP). On RISC processors, which maintain the stack manually, the savings can be significant. For such procedures, the overhead of the procedure call and return and activation record maintenance is greater than the actual work done by the procedure. Therefore, eliminating the activation record maintenance code could nearly double the speed of the procedure. For these and other reasons, you should try to keep your call trees as shallow as possible. The more leaf procedures your program uses, the more efficient it may become when you compile it with a decent compiler.

15.3 Macros and Inline Functions

One offshoot of the structured programming revolution was that computer programmers were taught to write small, modular, and logically coherent functions.[2] A function that is logically coherent does one thing well. All of the statements in such a procedure or function are dedicated to doing the task at hand without producing any side computations or doing any extraneous operations. Years of software engineering research indicate that decomposing a problem into small components, and then implementing those, produces programs that are easier to read, maintain, and modify. Unfortunately, it's easy to get carried away with this process and produce functions like the following Pascal example:

```
function sum( a:integer; b:integer ):integer;
begin

       (* returns sum of a & b as function result *)

        sum := a + b;

end;
      .
      .
      .
sum( aParam, bParam );
```

2. *WGC, Volume 4: Great Design,* covers this subject in greater detail.

On the 80x86, it would probably take about three instructions to compute the sum of two values and store that sum into a memory variable. For example:

```
mov( aParam, eax );
add( bParam, eax );
mov( eax, destVariable );
```

Contrast this with the code necessary to simply *call* the function `sum()`:

```
push( aParam );
push( bParam );
call sum;
```

Within the procedure `sum` (assuming a mediocre compiler), you might expect to find code like the following HLA sequence:

```
// Construct the activation record

push( ebp );
mov( esp, ebp );

// Get aParam's value

mov( [ebp+12], eax );

// Compute their sum and return in EAX

add( [ebp+8], eax );

// Restore EBP's value

pop( ebp );

// Return to caller, cleaning up
// the activation record.

ret( 8 );
```

As you can see, using a function takes three times as many instructions to compute the sum of these two objects as the straight-line (no function call) code. Worse still, these nine instructions are generally slower than the three that make up the inline code. The inline code could run 5 to 10 times faster than the code with the function call.

The one redeeming quality about the overhead associated with a function or procedure call is that it's fixed. It takes the same number of instructions to set up the parameters and the activation record whether the procedure or function body contains 1 or 1,000 machine instructions. Although the overhead of a procedure call is huge when the procedure's body is small, it's inconsequential when the procedure's body is large. Therefore, to reduce the impact of procedure/function call overhead in your programs, try to place larger procedures and functions and write shorter sequences as inline code.

Finding the optimum balance between the benefits of modular structure and the cost of too-frequent procedure calls can be difficult. Unfortunately, good program design often prevents us from increasing the size of our procedures and functions enough that the overhead of the call and return becomes insignificant. Sure, we could combine several functions and procedure calls into a single procedure or function, but this would violate several rules of programming style, and great code usually avoids such tactics. (One problem with the resulting programs is that few people can figure out how they work in order to optimize them.) However, if you can't sufficiently lower the overhead of a procedure's body by increasing the procedure's size, you can still improve overall performance by reducing the overhead in other ways. As you've seen, one option is to use leaf procedures and functions. Good compilers emit fewer instructions for leaf nodes in the call tree, thereby reducing the call/return overhead. However, if the procedure's body is short, you need a way to completely eliminate the procedure call/return overhead. Some languages accomplish this with *macros*.

A *pure* macro expands the body of a procedure or function in place of its invocation. Because there's no call/return to code elsewhere in the program, a macro expansion avoids the overhead associated with those instructions. Furthermore, macros also save considerable expense by using textual substitution for parameters rather than pushing the parameter data onto the stack or moving it into registers. The drawback to a macro is that the compiler expands the macro's body for each invocation of the macro. If the macro body is large and you invoke it in many different places, the executable program can grow by a fair amount. Macros represent the classic time/space tradeoff: faster code at the expense of greater size. For this reason, you should use macros only to replace procedures and functions that have a small number of statements (say, between one and five), except in some rare cases where speed is paramount.

A few languages (like C/C++) provide *inline* functions and procedures, which are a cross between a true function (or procedure) and a pure macro. Most languages that support inline functions and procedures do not guarantee that the compiler will expand the code inline. *Inline expansion*, or a call to an actual function in memory, is done at the compiler's discretion. Most compilers won't expand an inline function if its body is too large or if it has an excessive number of parameters. Furthermore, unlike pure macros, which don't have any associated procedure call overhead, inline functions may still need to build an activation record in order to handle local variables, temporaries, and other requirements. Thus, even if the compiler does expand such a function inline, there may still be some overhead that you wouldn't get with a pure macro.

To see the result of function inlining, consider the following C source file prepared for compilation by Microsoft Visual C++:

```
#include <stdio.h>

// Make geti and getj external functions
// to thwart constant propagation so we
// can see the effects of the following
// code.

extern int geti( void );
extern int getj( void );

// Inline function demonstration. Note
// that "_inline" is the legacy MSVC++ "C" way
// of specifying an inline function (the
// actual "inline" keyword was a C++/C99 feature,
// which this code avoids in order to make
// the assembly output a little more readable).
//
//
// "inlineFunc" is a simple inline function
// that demonstrates how the C/C++ compiler
// does a simple inline macro expansion of
// the function:

_inline int inlineFunc( int a, int b )
{
    return a + b;
}

_inline int ilf2( int a, int b )
{
    // Declare some variable that will require
    // an activation record to be built (that is,
    // register allocation won't be sufficient):

    int m;
    int c[4];
    int d;

    // Make sure we use the "c" array so that
    // the optimizer doesn't ignore its
    // declaration:

    for( m = 0; m < 4; ++m )
    {
        c[m] = geti();
    }
    d = getj();
    for( m = 0; m < 4; ++m )
    {
        d += c[m];
    }
    // Return a result to the calling program:

    return (a + d) - b;
}


int main( int argc, char **argv )
{
    int i;
    int j;
    int sum;
    int result;

    i = geti();
    j = getj();
    sum = inlineFunc( i, j );
    result = ilf2( i, j );
    printf( "i+j=%d, result=%d\n", sum, result );
    return 0;
}
```

Here's the MASM-compatible assembly language code that MSVC emits when you specify a C compilation (versus a C++ compilation, which produces messier output):

```
_main      PROC NEAR
main    PROC
;
; Create the activation record:
;
$LN6:
        mov     QWORD PTR [rsp+8], rbx
        push    rdi
        sub     rsp, 32        ; 00000020H
; Line 66
;
; i = geti();
;
        call    ?geti@@YAHXZ   ; geti -- returns result in EAX
        mov     edi, eax       ; Save i in edi

; Line 67
;
; j = getj();
;
        call    ?getj@@YAHXZ   ; getj -- Returns result in EAX
; Line 69
;
; Inline expansion of inlineFunc()
;
        mov     edx, eax       ; Pass j in EDX
        mov     ecx, edi       ; Pass i in ECX
        mov     ebx, eax       ; Use EBX as "sum" local
        call    ?ilf2@@YAHHH@Z ; ilf2

;       Computes sum = i+j (inline)

        lea     edx, DWORD PTR [rbx+rdi]

; Line 70
;
; Call to printf function:

        mov     r8d, eax
        lea     rcx, OFFSET FLAT:??_C@_0BD@INCDFJPK@i?$CLj?$DN?$CFd?0?5result?$DN?$CFd?6?$AA@
        call    printf
; Line 72
;
; Return from main function
;
        mov     rbx, QWORD PTR [rsp+48]
        xor     eax, eax
        add     rsp, 32        ; 00000020H
        pop     rdi
        ret     0
main    ENDP

?ilf2@@YAHHH@Z PROC            ; ilf2, COMDAT
; File v:\t.cpp
; Line 30
$LN24:
        mov     QWORD PTR [rsp+8], rbx
        mov     QWORD PTR [rsp+16], rsi
        push    rdi
        sub     rsp, 64        ; 00000040H
;
; Extra code to help prevent hacks from messing with
; stack data (clears array data to prevent observing old
; memory data).

        mov     rax, QWORD PTR __security_cookie
        xor     rax, rsp
        mov     QWORD PTR __$ArrayPad$[rsp], rax
        mov     edi, edx
        mov     esi, ecx
; Line 43
; Loop to fill "v" array:
;
        xor     ebx, ebx
$LL4@ilf2:
; Line 45
        call    ?geti@@YAHXZ   ; geti
        mov     DWORD PTR c$[rsp+rbx*4], eax
        inc     rbx
        cmp     rbx, 4
        jl      SHORT $LL4@ilf2

; Line 47
;
; d = getj();
;
        call    ?getj@@YAHXZ   ; getj
; Line 50
;
; Second for loop is unrolled and expanded inline:
;
; d += c[m];

        mov     r8d, DWORD PTR c$[rsp+8]
        add     r8d, DWORD PTR c$[rsp+12]
        add     r8d, DWORD PTR c$[rsp]
        add     r8d, DWORD PTR c$[rsp+4]
;
; return (a+d) - b
;
        add     eax, r8d
; Line 55
        sub     eax, edi
        add     eax, esi
; Line 56
;
; Verify code did not mess with stack before leaving
; (array overflow):
;
        mov     rcx, QWORD PTR __$ArrayPad$[rsp]
        xor     rcx, rsp
        call    __security_check_cookie
        mov     rbx, QWORD PTR [rsp+80]
        mov     rsi, QWORD PTR [rsp+88]
        add     rsp, 64        ; 00000040H
        pop     rdi
        ret     0
?ilf2@@YAHHH@Z ENDP            ; ilf2

?inlineFunc@@YAHHH@Z PROC      ; inlineFunc, COMDAT
; File v:\t.cpp
; Line 26
        lea     eax, DWORD PTR [rcx+rdx]
; Line 27
        ret     0
?inlineFunc@@YAHHH@Z ENDP      ; inlineFunc
```

As you can see in this assembly output, there are no function calls to the `inlineFunc()` function. Instead, the compiler expanded this function in place in the `main()` function, at the point where the main program calls it. Although the `ilf2()` function was also declared inline, the compiler refused to expand it inline and treated it like a normal function (probably because of its size).

15.4 Passing Parameters to a Function or Procedure

The number and type of parameters can also have a big impact on the efficiency of the code a compiler generates for your procedures and functions. Simply put, the more parameter data you pass, the more expensive the procedure or function call becomes. Often, programmers call generic functions (or design generic functions) that require you to pass several optional parameters whose values the function won't use. This scheme can make functions more generally applicable to different applications, but—as you'll see in this section—there's a cost associated with that generality, so you might want to consider using a version of the function specific to your application if space or speed is an issue.

The parameter-passing mechanism (for example, pass-by-reference or pass-by-value) also has an impact on the overhead associated with a procedure call and return. Some languages allow you to pass large data objects by value. (Pascal lets you pass strings, arrays, and records by value, and C/C++ allows you to pass structures by value; other languages vary depending on their design.) Whenever you pass a large data object by value, the compiler must emit machine code that makes a copy of that data into the procedure's activation record. This can be time-consuming (especially when copying large arrays or structures). Furthermore, large objects probably won't fit in the CPU's register set, so accessing such data within a procedure or function is expensive. It's usually more efficient to pass large data objects such as arrays and structures by reference than by value. The extra cost of accessing the data indirectly is usually saved many times over by not having to copy the data into the activation record. Consider the following C code, which passes a large structure by value to a C function:

```c
#include <stdio.h>

typedef struct
{
    int array[256];
} a_t;

void f( a_t a )
{
    a.array[0] = 0;
    return;
}

int main( void )
{
    a_t b;

    f( b );
    return( 0 );
}
```

Here's the PowerPC code that GCC emits:

```
_f:
        li r0,0 ; To set a.array[0] = 0

        ; Note: the PowerPC ABI passes the
        ; first eight dwords of data in
        ; R3..R10. We need to put that
        ; data back into the memory array
        ; here:

        stw r4,28(r1)
        stw r5,32(r1)
        stw r6,36(r1)
        stw r7,40(r1)
        stw r8,44(r1)
        stw r9,48(r1)
        stw r10,52(r1)

        ; Okay, store 0 into a.array[0]:

        stw r0,24(r1)

        ; Return to caller:

        blr

; main function:

_main:

        ; Set up main's activation record:

        mflr r0
        li r5,992
        stw r0,8(r1)

        ; Allocate storage for a:

        stwu r1,-2096(r1)

        ; Copy all but the first
        ; eight dwords to the
        ; activation record for f:

        addi r3,r1,56
        addi r4,r1,1088
        bl L_memcpy$stub

        ; Load the first eight dwords
        ; into registers (as per the
        ; PowerPC ABI):

        lwz r9,1080(r1)
        lwz r3,1056(r1)
        lwz r10,1084(r1)
        lwz r4,1060(r1)
        lwz r5,1064(r1)
        lwz r6,1068(r1)
        lwz r7,1072(r1)
        lwz r8,1076(r1)

        ; Call the f function:

        bl _f

        ; Clean up the activation record
        ; and return 0 to main's caller:

        lwz r0,2104(r1)
        li r3,0
        addi r1,r1,2096
        mtlr r0
        blr

; Stub function that copies the structure
; data to the activation record for the
; main function (this calls the C standard
; library memcpy function to do the actual copy):

        .data
        .picsymbol_stub
L_memcpy$stub:
        .indirect_symbol _memcpy
        mflr r0
        bcl 20,31,L0$_memcpy
L0$_memcpy:
        mflr r11
        addis r11,r11,ha16(L_memcpy$lazy_ptr-L0$_memcpy)
        mtlr r0
        lwz r12,lo16(L_memcpy$lazy_ptr-L0$_memcpy)(r11)
        mtctr r12
        addi r11,r11,lo16(L_memcpy$lazy_ptr-L0$_memcpy)
        bctr
.data
.lazy_symbol_pointer
L_memcpy$lazy_ptr:
        .indirect_symbol _memcpy
        .long dyld_stub_binding_helper
```

As you can see, the call to function `f()` calls `memcpy` to transfer a copy of the data from the `main()` function's local array to the `f()` function's activation record. Again, copying memory is a slow process, and this code amply demonstrates that you should avoid passing large objects by value. Consider the same code when you pass the structure by reference:

```
#include <stdio.h>

typedef struct
```

```
{
    int array[256];
} a_t;

void f( a_t *a )
{
    a->array[0] = 0;
    return;
}

int main( void )
{
    a_t b;

    f( &b );
    return( 0 );
}
```

Here's the conversion of this C source code to 32-bit ARM assembly by GCC:

```
f:
    @ Build activation record:

    str fp, [sp, #-4]!  @ Push old FP on stack
    add fp, sp, #0      @ FP = SP
    sub sp, sp, #12     @ Reserve storage for locals
    str r0, [fp, #-8]   @ Save pointer to 'a'
    ldr r3, [fp, #-8]   @ r3 = a

    @ a->array[0] = 0;

    mov r2, #0
    str r2, [r3]
    nop

    @ Remove locals from stack.

    add sp, fp, #0

    @ Pop FP from stack:

    ldr fp, [sp], #4

    @ Return to main function:

    bx  lr

main:
    @ Save Linux return address and FP:

    push    {fp, lr}
    @ Set up activation record:

    add fp, sp, #4
    sub sp, sp, #1024   @ Reserve storage for b
    sub r3, fp, #1024   @ R3 = &b
    sub r3, r3, #4

    mov r0, r3          @ Pass &b to f in R0
    bl  f               @ Call f

    @ Return 0 result to Linux:

    mov r3, #0
    mov r0, r3
    sub sp, fp, #4      @ Clean up stack frame
    pop {fp, pc}        @ Returns to Linux
```

Depending on your CPU and compiler, it may be slightly more efficient to pass small (scalar) data objects by value rather than by reference. For example, if you're using an 80x86 compiler that passes parameters on the stack, you'll need two instructions to pass a memory object by reference, but only a single instruction to pass that same object by value. So, although trying to pass large objects by reference is a good idea, the reverse is generally true for small objects. However, this is not a hard and fast rule; its validity varies based on the CPU and compiler you're using.

Some programmers may feel that it's more efficient to pass data to a procedure or function via global variables. After all, if the data is already sitting in a global variable that's accessible to the procedure or function, a call to that procedure or function won't require any extra instructions to pass the data to the subroutine, therefore reducing the call overhead. While this seems like a big win, keep in mind that compilers have a difficult time optimizing programs that make excessive use of global variables. Although using globals may reduce the function/procedure call overhead, it may also prevent the compiler from handling other optimizations that would have been otherwise possible. Here's a simple example using Microsoft Visual C++ that demonstrates this problem:

```
#include <stdio.h>

// Make geti an external function
// to thwart constant propagation so we
// can see the effects of the following
// code.

extern int geti( void );

// globalValue is a global variable that
// we use to pass data to the "usesGlobal"
// function:

int globalValue = 0;
```

```
// Inline function demonstration. Note
// that "_inline" is the legacy MSVC++ "C" way
// of specifying an inline function (the
// actual "inline" keyword is a C99/C++ feature,
// which this code avoids in order to make
// the assembly output a little more readable).


_inline int usesGlobal( int plusThis )
{
    return globalValue+plusThis;
}

_inline int usesParm( int plusThis, int globalValue )
{
    return globalValue+plusThis;
}


int main( int argc, char **argv )
{
    int i;
    int sumLocal;
    int sumGlobal;

    // Note: the call to geti in between setting globalValue
    // and calling usesGlobal is intentional. The compiler
    // doesn't know that geti doesn't modify the value of
    // globalValue (and neither do we, frankly), and so
    // the compiler cannot use constant propagation here.

    globalValue = 1;
    i = geti();
    sumGlobal = usesGlobal( 5 );

    // If we pass the "globalValue" as a parameter rather
    // than setting a global variable, then the compiler
    // can optimize the code away:

    sumLocal = usesParm( 5, 1 );
    printf( "sumGlobal=%d, sumLocal=%d\n", sumGlobal, sumLocal );
    return 0;
}
```

Here's the 32-bit MASM source code (with manual annotations) that the MSVC++ compiler generates for this code:

```
_main      PROC NEAR

;   globalValue = 1;

    mov    DWORD PTR _globalValue, 1

;   i = geti();
```

```
;
; Note that because of dead code elimination,
; MSVC++ doesn't actually store the result
; away into i, but it must still call geti()
; because geti() could produce side effects
; (such as modifying globalValue's value).

    call   _geti

;   sumGlobal = usesGlobal( 5 );
;
; Expanded inline to:
;
; globalValue+plusThis

    mov    eax, DWORD PTR _globalValue
    add    eax, 5          ; plusThis = 5

; The compiler uses constant propagation
; to compute:
;   sumLocal = usesParm( 5, 1 );
; at compile time. The result is 6, which
; the compiler directly passes to print here:

    push   6

; Here's the result for the usesGlobal expansion,
; computed above:

    push   eax
    push   OFFSET FLAT:formatString ; 'string'
    call   _printf
    add    esp, 12                  ; Remove printf parameters

; return 0;

    xor    eax, eax
    ret    0
_main      ENDP
_TEXT      ENDS
           END
```

As you can see, the compiler's ability to optimize around global variables can be easily thwarted by the presence of some seemingly unrelated code. In this example, the compiler cannot determine that the call to the external `geti()` function doesn't modify the value of the `globalValue` variable. Therefore, the compiler can't assume that `globalValue` still has the value `1` when it computes the inline function result for `usesGlobal()`. Exercise extreme caution when using global variables to communicate information between a procedure or function and its caller. Code that's unrelated to the task at hand (such as the call to `geti()`, which probably doesn't affect `globalValue`'s value) can prevent the compiler from optimizing code that uses global variables.

15.5 Activation Records and the Stack

Because of how a stack works, the last procedure activation record the software creates will be the first activation record that the system deallocates. Since activation records hold procedure parameters and local variables, a *last-in, first-out (LIFO)* organization is a very intuitive way of implementing activation records. To see how it works, consider the following (trivial) Pascal program:

```
program ActivationRecordDemo;

    procedure C;
    begin

        (* Stack Snapshot here *)

    end;

    procedure B;
    begin

        C;

    end;

    procedure A;
    begin

        B;

    end;

begin (* Main program *)

    A;

end.
```

Figure 15-2 shows the stack layout as this program executes.

When the program begins execution, it first creates an activation record for the main program. The main program calls the `A` procedure (①). Upon entry into the `A` procedure, the code completes the construction of the activation record for `A`, effectively pushing it onto the stack. Once inside procedure `A`, the code calls procedure `B` (②). Note that `A` is still active while the code calls `B`, so `A`'s activation record remains on the stack. Upon entry into `B`, the system builds `B`'s activation record and pushes it onto the top of the stack (③). Once inside `B`, the code calls procedure `C`, and `C` builds its activation record on the stack and arrives at the comment `(* Stack snapshot here *)` (④).

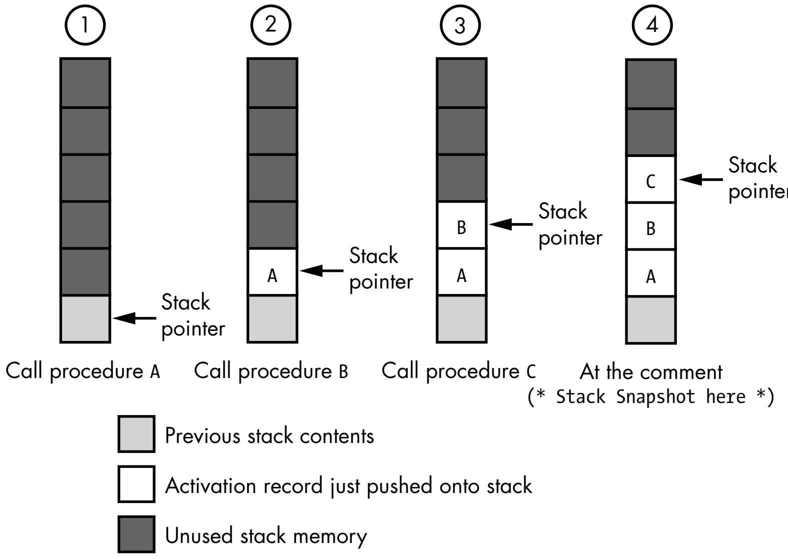


Figure 15-2: Stack layout after three nested procedure calls

Because procedures keep their local variables and parameter values in their activation record, the lifetime of these variables extends from the point the system first creates the activation record until the system deallocates it when the procedure returns to its caller. In Figure 15-2, notice that `A`'s activation record remains on the stack during the execution of the `B` and `C` procedures. Therefore, the lifetime of `A`'s parameters and local variables completely brackets the lifetimes of `B`'s and `C`'s activation records.

Now consider the following C/C++ code with a recursive function:

```
void recursive( int cnt )
{
    if( cnt != 0 )
    {
        recursive( cnt - 1 );
    }
}

int main( int argc, char **argv )
{
    recursive( 2 );
}
```

This program calls the `recursive()` function three times before it begins returning (the main program calls `recursive()` once with the parameter value `2`, and `recursive()` calls itself twice with the parameter values `1` and `0`). Because each recursive call to `recursive()` pushes another activation record before the current call returns, when this program finally hits the `if` statement in this code with `cnt` equal to `0`, the stack looks something like Figure 15-3.

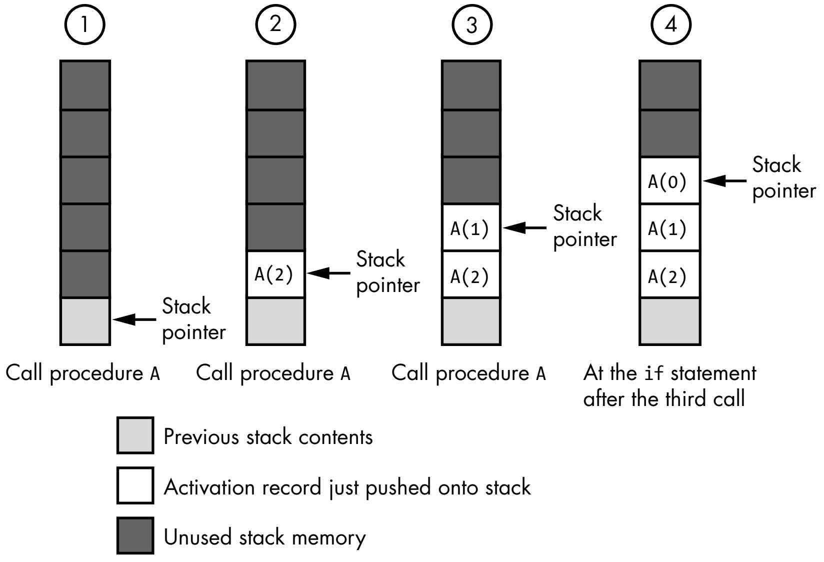


Figure 15-3: Stack layout after three recursive procedure calls

Because each procedure invocation has a separate activation record, each activation of the procedure will have its own copy of the parameters and local variables. While the code for a procedure or function is executing, it will access only those local variables and parameters in the activation record it has most recently created,[3] thereby preserving the values from previous calls.

15.5.1 Breaking Down the Activation Record

Now that you've seen how procedures manipulate activation records on the stack, it's time to take a look at the internal composition of a typical activation record. In this section we'll use a typical activation record layout that you'll see when executing code on an 80x86. Although different languages, different compilers, and different CPUs lay out the activation record differently, these differences, if they exist at all, will be minor.

The 80x86 maintains the stack and activation records using two registers: ESP/RSP (the stack pointer) and EBP/RBP (the frame pointer, which Intel calls the *base pointer*). The ESP/RSP register points at the current top of stack, and the EBP register points at the base address of an activation record.[4] A procedure can access objects within its activation record by using the indexed addressing mode (see "Indexed Addressing Mode" on page 34) and supplying a positive or negative offset from the value in the EBP/RBP register. Generally, a procedure allocates memory storage for

3. The only exception is when a procedure recursively calls itself and passes one of its local variables or parameters by reference to the new invocation.

4. Some people call activation records *stack frames*, which is where the phrase *frame pointer* comes from. Intel chose the name *base pointer* for the EBP register because it points at the base address of the stack frame.

local variables at negative offsets from EBP/RBP's value, and for parameters at positive offsets from EBP/RBP. Consider the following Pascal procedure, which has both parameters and local variables:

```
procedure HasBoth( i:integer; j:integer; k:integer );
var
    a  :integer;
    r  :integer;
    c  :char;
    b  :char;
    w  :smallint;  (* smallints are 16 bits *)
begin
        .
        .
        .
end;
```

Figure 15-4 shows a typical activation record for this Pascal procedure (remember that the stack grows toward lower memory on the 32-bit 80x86).

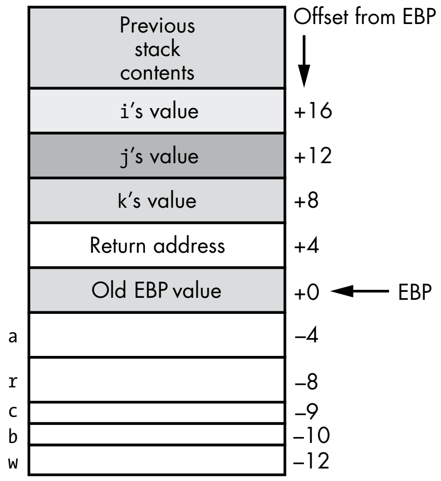


Figure 15-4: A typical activation record

When you see the term *base* associated with a memory object, you might think that the base address is the lowest address of that object in memory. However, there's no such requirement. The base address is simply the address in memory on which you base the offsets to particular fields of that object. As this activation record demonstrates, 80x86 activation record base addresses are actually in the middle of the record.

The activation record is constructed in two phases. The first phase begins when the code calling the procedure pushes the parameters for the call onto the stack. For example, consider the following call to `HasBoth()` in the previous example:

```
HasBoth( 5, x, y + 2 );
```

Here's the HLA/x86 assembly code that might correspond to this call:

```
pushd( 5 );
push( x );
mov( y, eax );
add( 2, eax );
push( eax );
call HasBoth;
```

The three push instructions in this code sequence build the first three double words of the activation record, and the call instruction pushes a *return address* onto the stack, creating the fourth double word in the activation record. After the call, execution continues in the HasBoth() procedure itself, where the program continues to build the activation record.

The first few instructions of the HasBoth() procedure are responsible for finishing the construction of the activation record. Immediately upon entry into HasBoth(), the stack takes the form shown in Figure 15-5.[5]

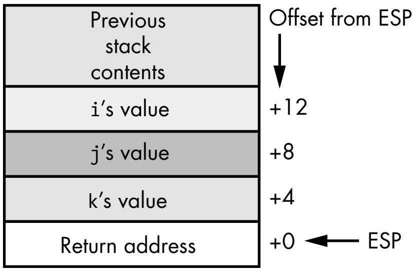


Figure 15-5: Activation record upon entry to HasBoth()

The first thing the procedure's code should do is to preserve the value in the 80x86 EBP register. On entry, EBP probably points at the base address of the caller's activation record. On exit from HasBoth(), EBP needs to contain its original value. Therefore, upon entry, HasBoth() needs to push the current value of EBP on the stack in order to preserve EBP's value. Next, the HasBoth() procedure needs to change EBP so that it points at the base address of the HasBoth() activation record. The following HLA/x86 code takes care of these two operations:

```
// Preserve caller's base address.

        push( ebp );

        // ESP points at the value we just saved. Use its address
        // as the activation record's base address.

        mov( esp, ebp );
```

5. On the x86-64 CPU, the offsets will be slightly different because certain objects (such as the return address) are 64-bit entities rather than 32-bit entities.

Finally, the code at the beginning of `HasBoth()` needs to allocate storage for its local (automatic) variables. As you saw in Figure 15-4, those variables sit below the frame pointer in the activation record. To prevent future pushes from wiping out the values in those local variables, the code has to set ESP to the address of the last double word of local variables in the activation record. To accomplish this, it simply subtracts the number of bytes of local variables from ESP via the following machine instruction:

```
sub( 12, esp );
```

The *standard entry sequence* for a procedure like `HasBoth()` consists of the three machine instructions just considered—`push(ebp);`, `mov(esp, ebp);`, and `sub(12, esp);`—which complete the construction of the activation record inside the procedure. Just before returning, the Pascal procedure is responsible for deallocating the storage associated with the activation record. The *standard exit sequence* usually takes the following form (in HLA) for a Pascal procedure:

```
// Deallocates the local variables
// by copying EBP to ESP.

mov( ebp, esp );

// Restore original EBP value.

pop( ebp );

// Pops return address and
//  12 parameter bytes (3 dwords)

ret( 12 );
```

The first instruction deallocates storage for the local variables shown in Figure 15-4. Note that EBP is pointing at the old value of EBP; this value is stored at the memory address just above all the local variables. By copying the value in EBP to ESP, we move the stack pointer past all the local variables, effectively deallocating them. Now, the stack pointer points at the old value of EBP on the stack; therefore, the `pop` instruction in this sequence restores EBP's original value and leaves ESP pointing at the return address on the stack. The `ret` instruction in the standard exit sequence does two things: it pops the return address from the stack (and, of course, transfers control to this address), and it removes 12 bytes of parameters from the stack. Because `HasBoth()` has three double-word parameters, popping 12 bytes from the stack removes those parameters.

15.5.2 Assigning Offsets to Local Variables

This `HasBoth()` example allocates local (automatic) variables in the order the compiler encounters them. A typical compiler maintains a *current offset* (the initial value of which will be 0) into the activation record for local variables.

Whenever the compiler encounters a local variable, it subtracts the variable's size from the current offset and then uses the result as the offset of the local variable (from EBP/RBP) in the activation record. For example, upon encountering the declaration for variable `a`, the compiler subtracts the size of `a` (4 bytes, assuming `a` is a 32-bit integer) from the current offset (0) and uses the result (–4) as the offset for `a`. Next, the compiler encounters variable `r` (which is also 4 bytes), sets the current offset to –8, and assigns this offset to `r`. This process repeats for each of the local variables in the procedure.

Although this is a typical way that compilers assign offsets to local variables, most languages give compiler implementers free rein to allocate local objects as they see fit. A compiler can rearrange the objects in the activation record if doing so is more convenient. This means you should avoid designing algorithms that depend on the previously mentioned allocation scheme, because some compilers do it differently.

Many compilers try to ensure that all local variables you declare have an offset that is a multiple of the object's size. For example, suppose you have the following two declarations in a C function:

```
char c;
int  i;
```

Normally, you'd expect that the compiler would attach an offset like –1 to the `c` variable and –5 to the (4-byte) `int` variable `i`. However, some CPUs (such as RISC CPUs) require the compiler to allocate double-word objects on a double-word boundary. Even on CPUs that don't require this (for example, the 80x86), it may be faster to access a double-word variable if the compiler aligns it on a double-word boundary. For this reason (and as previous chapters have described), many compilers automatically add padding bytes between local variables so that each variable resides at a *natural* offset in the activation record. In general, bytes may appear at any offset, words are happiest on even address boundaries, and double words should have a memory address that is a multiple of 4.

Although an optimizing compiler might automatically handle this alignment for you, that comes with a cost—those extra padding bytes. As explained earlier, compilers are usually free to rearrange the variables in an activation record, but they don't always do so. Therefore, if you intertwine the definitions for several byte, word, double-word, and other-sized objects in your local variable declarations, the compiler may wind up inserting several bytes of padding into the activation record. You can minimize this problem by attempting to group as many like-sized objects together as is reasonable in your procedures and functions. Consider the following C/C++ code on a 32-bit machine:

```
char c0;
int  i0;
char c1;
int  i1;
char c2;
```

```
int  i2;
char c3;
int  i3;
```

An optimizing compiler may elect to insert 3 bytes of padding between each of these character variables and the (4-byte) integer variable that immediately follows. This means that the preceding code will have about 12 bytes of wasted space (3 bytes for each of the character variables). Now consider the following declarations in the same C code:

```
char c0;
char c1;
char c2;
char c3;
int  i0;
int  i1;
int  i2;
int  i3;
```

In this example, the compiler won't emit any extra padding bytes to the code. Why? Because characters (being 1 byte each) may begin at any address in memory.[6] Therefore, the compiler can place these character variables at offsets –1, –2, –3, and –4 within the activation record. Because the last character variable appears at an address that is a multiple of 4, the compiler doesn't need to insert any padding bytes between `c3` and `i0` (`i0` will naturally appear at offset –8 in the preceding declarations).

As you can see, arranging your declarations so that all like-sized objects are next to one another can help your compiler produce better code. Don't take this suggestion to an extreme, though. If a rearrangement would make your program more difficult to read or maintain, you should carefully consider whether it's worthwhile in your program.

15.5.3 Associating Offsets with Parameters

As noted, compilers are given considerable leeway with respect to how they assign offsets to local (automatic) variables within a procedure. As long as the compiler uses these offsets consistently, the exact allocation algorithm it applies is almost irrelevant; in fact, it could use a different allocation scheme in different procedures of the same program. However, a compiler *doesn't* have free rein when assigning offsets to parameters. It has to live with certain restrictions, because other code outside the procedure accesses those parameters. Specifically, the procedure and the calling code must agree on the layout of the parameters in the activation record so the calling code can build the parameter list. Note that the calling code might not be in the same source file, or even in the same programming language, as the procedure. To ensure interoperability between a procedure and whatever

6. Keep in mind that on some RISC machines, accessing individual bytes in memory can be more expensive than double words. This means that a RISC compiler might allocate 4 (or even 8) bytes for each character variable.

code calls that procedure, then, compilers must adhere to certain *calling conventions*. This section will explore the three common calling conventions for Pascal/Delphi and C/C++.

15.5.3.1 The Pascal Calling Convention

In Pascal (including Delphi) the standard parameter-passing convention is to push the parameters on the stack in the order of their appearance in the parameter list. Consider the following call to the `HasBoth()` procedure from the earlier example:

```
HasBoth( 5, x, y + 2 );
```

The following assembly code implements this call:

```
// Push the value for parameter i:

pushd( 5 );

// Push x's value for parameter j:

push( x );

// Compute y + 2 in EAX and push this as the value
// for parameter k:

mov( y, eax );
add( 2, eax );
push( eax );

// Call the HasBoth procedure with these
// three parameter values:

call HasBoth;
```

When assigning offsets to a procedure's formal parameters, the compiler assigns the highest offset to the first parameter and the lowest offset to the last parameter. Because the old value of EBP is at offset 0 in the activation record and the return address is at offset 4, the last parameter in the activation record (when using the Pascal calling convention on the 80x86 CPU) will reside at offset 8 from EBP. Looking back at Figure 15-4, you can see that parameter `k` is at offset +8, parameter `j` is at offset +12, and parameter `i` (the first parameter) is at offset +16 in the activation record.

The Pascal calling convention also stipulates that it is the procedure's responsibility to remove the parameters the caller pushes when the procedure returns to its caller. As you saw earlier, the 80x86 CPU provides a variant of the `ret` instruction that lets you specify how many bytes of parameters to remove from the stack upon return. Therefore, a procedure that uses the Pascal calling convention will typically supply the number of parameter bytes as an operand to the `ret` instruction when returning to its caller.

15.5.3.2 The C Calling Convention

The C/C++/Java languages employ another very popular calling convention, generally known as the *cdecl calling convention* (or, simply, the *C calling convention*). There are two major differences between the C and Pascal calling conventions. First, calls to functions in C must push their parameters on the stack in the reverse order. That is, the first parameter must appear at the lowest address on the stack (assuming the stack grows downward), and the last parameter must appear at the highest address in memory. The second difference is that C requires the caller, rather than the function, to remove all parameters from the stack.

Consider the following version of `HasBoth()` written in C instead of Pascal:

```
void HasBoth( int i, int j, int k )
{
    int a;
    int r;
    char c;
    char b;
    short w;  /* assumption: short ints are 16 bits */
        .
        .
        .
}
```

Figure 15-6 provides the layout for a typical `HasBoth` activation record (written in C on a 32-bit 80x86 processor).

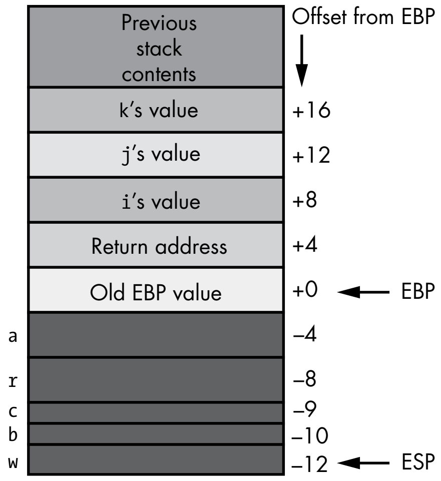


Figure 15-6: `HasBoth()` activation record in C

Looking closely, you'll see the difference between this figure and Figure 15-4. The positions of the `i` and `k` variables are reversed in this activation record (it's only a coincidence that `j` appears at the same offset in both).

Because the C calling convention reverses the order of the parameters and it's the caller's responsibility to remove all parameter values from the stack, the calling sequence for `HasBoth()` is a little different in C than in Pascal. Consider the following call to `HasBoth()`:

```
HasBoth( 5, x, y + 2 );
```

Here's the HLA assembly code for this call:

```
// Compute y + 2 in EAX and push this
// as the value for parameter k

mov( y, eax );
add( 2, eax );
push( eax );

// Push x's value for parameter j

push( x );

// Push the value for parameter i

pushd( 5 );

// Call the HasBoth procedure with
// these three parameter values

call HasBoth;

// Remove parameters from the stack.

add( 12, esp );
```

As a result of using the C calling convention, this code differs in two ways from the assembly code for the Pascal implementation. First, this example pushes the values of the actual parameters in the opposite order of the Pascal code; that is, it first computes `y+2` and pushes that value, then it pushes `x`, and finally it pushes the value `5`. The second difference is the inclusion of the `add(12,esp);` instruction immediately after the call. This instruction removes 12 bytes of parameters from the stack upon return. The return from `HasBoth()` will use only the `ret` instruction, not the `ret n` instruction.

15.5.3.3 Conventions for Passing Parameters in Registers

As you've seen in these examples, passing parameters on the stack between two procedures or functions requires a fair amount of code. Good assembly language programmers have long known that it's better to pass parameters in registers. Therefore, several 80x86 compilers following Intel's ABI (application binary interface) rules may attempt to pass as many as three parameters in the EAX, EDX, and ECX registers.[7] Most RISC processors

7. The number of parameters chosen, three, is not arbitrary. Studies in software engineering strongly suggest that most user-written procedures have three or fewer parameters.

specifically set aside a set of registers for passing parameters between functions and procedures. See "Registers to the Rescue" on page 585 for more information.

Most CPUs require that the stack pointer remain aligned on some reasonable boundary (for example, a double-word boundary), and CPUs that don't absolutely require this may still benefit from it. Furthermore, many CPUs (the 80x86 included) can't easily push certain small-sized objects, like bytes, onto the stack. Therefore, most compilers reserve a minimum number of bytes for a parameter (typically 4), regardless of its actual size. As an example, consider the following HLA procedure fragment:

```
procedure OneByteParm( b:byte ); @nodisplay;
    // local variable declarations
begin OneByteParm;
    .
    .
    .
end OneByteParm;
```

The activation record for this procedure appears in Figure 15-7.

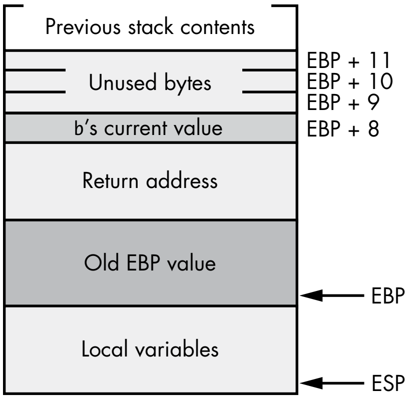


Figure 15-7: `OneByteParm()` activation record

As you can see, the HLA compiler reserves 4 bytes for the `b` parameter even though `b` is only a single-byte variable. This extra padding ensures that the ESP register will remain aligned on a double-word boundary.[8] We can easily push the value of `b` onto the stack in the code that calls `OneByteParm()` using a 4-byte `push` instruction.[9]

Even if your program could access the extra bytes of padding associated with the `b` parameter, doing so is never a good idea. Unless you've explicitly pushed the parameter onto the stack (for example, using assembly

8. Assuming, of course, it was so aligned prior to appearance of the `b` parameter on the stack.

9. The 80x86 does not directly support 1-byte pushes onto the stack, so if the compiler reserved only 1 byte of storage for this parameter, it would take several machine instructions in order to simulate that 1-byte push.

language code), there's no guarantee about the data values that appear in the padding bytes. In particular, they may not contain 0. Nor should your code assume that the padding is present or that the compiler pads such variables out to 4 bytes. Some 16-bit processors may require only a single byte of padding. Some 64-bit processors may require 7 bytes of padding. Some compilers on the 80x86 may use 1 byte of padding, while others use 3 bytes. Unless you're willing to live with code that only one compiler can compile (and code that could break when the next version of the compiler comes along), it's best to ignore these padding bytes.

15.5.4 Accessing Parameters and Local Variables

Once a subroutine sets up the activation record, accessing local (automatic) variables and parameters is easy. The machine code simply uses the indexed addressing mode to access such objects. Consider again the activation record in Figure 15-4. The variables in the `HasBoth()` procedure have the offsets found in Table 15-1.

Table 15-1: Offsets to Local Variables and Parameters in `HasBoth()` (Pascal Version)

Variable	Offset	Addressing mode example
`i`	+16	`mov( [ebp+16], eax );`
`j`	+12	`mov( [ebp+12], eax );`
`k`	+8	`mov( [ebp+8], eax );`
`a`	–4	`mov( [ebp-4], eax );`
`r`	–8	`mov( [ebp-8], eax );`
`c`	–9	`mov( [ebp-9], al );`
`b`	–10	`mov( [ebp-10], al );`
`w`	–12	`mov( [ebp-12], ax );`

The compiler allocates static local variables in a procedure at a fixed address in memory. Static variables do not appear in the activation record, so the CPU accesses static objects using the direct addressing mode.[10] As Chapter 3 discussed, in 80x86 assembly language instructions that use the direct addressing mode need to encode the full 32-bit address as part of the machine instruction. Therefore, instructions that use the direct addressing mode are usually at least 5 bytes long (and often longer). On the 80x86, if the offset from EBP is –128 through +127, then a compiler can encode an instruction of the form `[ebp + *constant*]` in as few as 2 or 3 bytes. Such instructions will be more efficient than those that encode a full 32-bit address. The same principle applies on other processors, even if those CPUs provide different addressing modes, address sizes, and so on. Specifically,

10. Assuming the object is a scalar object. If it is an array, for example, the machine code may use the indexed addressing mode to access elements of the static array.

accessing local variables whose offset is relatively small is generally more efficient than accessing static variables or variables with larger offsets.

Because most compilers allocate offsets for local (automatic) variables as the compiler encounters them, the first 128 bytes of local variables will be the ones with the shortest offsets (at least, on the 80x86; this value may be different for other processors).

Consider the following two sets of local variable declarations (presumably appearing with some C function):

```
// Declaration set #1:

char string[256];
int i;
int j;
char c;
```

Here's a second version of these declarations:

```
// Declaration set #2

int i;
int j;
char c;
char string[256];
```

Although these two declaration sections are semantically identical, there is a big difference in the code a compiler for the 32-bit 80x86 generates to access these variables. In the first declaration, the variable `string` appears at offset –256 within the activation record, `i` appears at offset –260, `j` appears at offset –264, and `c` appears at offset –265. Because these offsets are outside the range –128 through +127, the compiler will have to emit machine instructions that encode a 4-byte offset constant rather than a 1-byte constant. Accordingly, the code associated with these declarations will be larger and may run slower.

Now consider the second declaration. In this example the programmer declares the scalar (non-array) objects first. Therefore, the variables have the following offsets: `i` at –4, `j` at –8, `c` at –9, and `string` at –265. This turns out to be the optimal configuration for these variables (`i`, `j`, and `c` will use 1-byte offsets; `string` will require a 4-byte offset).

This example demonstrates another rule you should try to follow when declaring local (automatic) variables: declare smaller, scalar objects first within a procedure, followed by all the arrays, structures/records, and other large objects.

As explained in "Associating Offsets with Parameters" on page 570, if you declare several local objects with differing sizes adjacent to one another, the compiler may need to insert padding bytes to keep the larger objects aligned at an appropriate memory address. While worrying about a few wasted bytes here and there may seem ridiculous on machines with a gigabyte (or more) of RAM, those few padding bytes could be just enough to push the offsets of certain local variables beyond –128, causing the compiler to emit 4-byte offsets rather than 1-byte offsets for those variables. This is one more reason you should try to declare like-sized local variables adjacent to one another.

On RISC processors, such as the PowerPC or ARM, the range of possible offsets is usually much greater than ±128. This is a good thing, because once you exceed the range of the activation record offset that a RISC CPU can encode directly into an instruction, parameter and local variable access gets very expensive. Consider the following C program:

```
#include <stdio.h>
int main( int argc, char **argv )
{
    int a;
    int b[256];
    int c;
    int d[16*1024*1024];
    int e;
    int f;

    a = argc;
    b[0] = argc + argc;
    b[255] = a + b[0];
    c = argc + b[1];
    d[0] = argc + a;
    d[4095] = argc + b[255];
    e = a + c;
    printf
    (
        "%d %d %d %d %d ",
        a,
        b[0],
        c,
        d[0],
        e
    );
    return( 0 );
}
```

Here's the PowerPC assembly output from GCC:

```
.data
        .cstring
        .align 2
        LC0:
        .ascii "%d %d %d %d %d \0"
        .text

; main function:

        .align 2
        .globl _main
```

```
_main:
        ; Set up main's activation record:

        mflr r0
        stmw r30,-8(r1)
        stw r0,8(r1)
        lis r0,0xfbff
        ori r0,r0,64384
        stwux r1,r1,r0
        mr r30,r1
        bcl 20,31,L1$pb
L1$pb:
        mflr r31

        ; The following allocates
        ; 16MB of storage on the
        ; stack (R30 is the stack
        ; pointer here).

        addis r9,r30,0x400
        stw r3,1176(r9)

        ; Fetch the value of argc
        ; into the R0 register:

        addis r11,r30,0x400
        lwz r0,1176(r11)
        stw r0,64(r30)      ; a = argc

        ; Fetch the value of argc
        ; into r9

        addis r11,r30,0x400
        lwz r9,1176(r11)

        ; Fetch the value of argc
        ; into R0:

        addis r11,r30,0x400
        lwz r0,1176(r11)

        ; Compute argc + argc and
        ; store it into b[0]:

        add r0,r9,r0
        stw r0,80(r30)

        ; Add a + b[0] and
        ; store into c:

        lwz r9,64(r30)
        lwz r0,80(r30)
        add r0,r9,r0
        stw r0,1100(r30)
        ; Get argc's value, add in
        ; b[1], and store into c:

        addis r11,r30,0x400
        lwz r9,1176(r11)
        lwz r0,84(r30)
        add r0,r9,r0
        stw r0,1104(r30)

        ; Compute argc + a and
        ; store into d[0]:

        addis r11,r30,0x400
        lwz r9,1176(r11)
        lwz r0,64(r30)
        add r0,r9,r0
        stw r0,1120(r30)

        ; Compute argc + b[255] and
        ; store into d[4095]:

        addis r11,r30,0x400
        lwz r9,1176(r11)
        lwz r0,1100(r30)
        add r0,r9,r0
        stw r0,17500(r30)

        ; Compute argc + b[255]:

        lwz r9,64(r30)
        lwz r0,1104(r30)
        add r9,r9,r0

; ************************************************
        ; Okay, here's where it starts
        ; to get ugly. We need to compute
        ; the address of e so we can store
        ; the result currently held in r9
        ; into e. But e's offset exceeds
        ; what we can encode into a single
        ; instruction, so we have to use
        ; the following sequence rather
        ; than a single instruction.

        lis r0,0x400
        ori r0,r0,1120
        stwx r9,r30,r0

; ************************************************
        ; The following sets up the
        ; call to printf and calls printf:

        addis r3,r31,ha16(LC0-L1$pb)
        la r3,lo16(LC0-L1$pb)(r3)
        lwz r4,64(r30)
        lwz r5,80(r30)
        lwz r6,1104(r30)
        lwz r7,1120(r30)
        lis r0,0x400
        ori r0,r0,1120
        lwzx r8,r30,r0
        bl L_printf$stub
        li r0,0
        mr r3,r0
        lwz r1,0(r1)
        lwz r0,8(r1)
        mtlr r0
        lmw r30,-8(r1)
        blr


; Stub, to call the external printf function:

        .data
        .picsymbol_stub
L_printf$stub:
        .indirect_symbol _printf
        mflr r0
        bcl 20,31,L0$_printf
L0$_printf:
        mflr r11
        addis r11,r11,ha16(L_printf$lazy_ptr-L0$_printf)
        mtlr r0
        lwz r12,lo16(L_printf$lazy_ptr-L0$_printf)(r11)
        mtctr r12
        addi r11,r11,lo16(L_printf$lazy_ptr-L0$_printf)
        bctr
.data
.lazy_symbol_pointer
L_printf$lazy_ptr:
        .indirect_symbol _printf
        .long dyld_stub_binding_helper
```

This compilation was done under GCC without optimization to show what happens when your activation record grows to the point you can no longer encode activation record offsets into the instruction.

To encode the address of `e`, whose offset is too large, we need these three instructions:

```
lis r0,0x400
ori r0,r0,1120
stwx r9,r30,r0
```

instead of a single instruction that stores `R0` into the `a` variable, such as:

```
stw r0,64(r30)      ; a = argc
```

While two extra instructions in a program of this size might seem insignificant, keep in mind that the compiler will generate these extra instructions for each such access. If you frequently access a local variable with a huge offset, the compiler may generate a significant number of extra instructions throughout your function or procedure.

Of course, in a standard application running on a RISC, this problem seldom occurs because we rarely allocate local storage beyond the range that a single instruction can encode. Also, RISC compilers generally allocate scalar (non-array/non-structure) objects in registers rather than blindly allocating them at the next memory address in the activation record. For example, if you turn on GCC's optimization with the `-O2` command-line switch, you'll get the following PowerPC output:

```
.globl _main
_main:

; Build main's activation record:

        mflr r0
        stw r31,-4(r1)
        stw r0,8(r1)
        bcl 20,31,L1$pb
L1$pb:
        ; Compute values, set up parameters,
        ; and call printf:

        lis r0,0xfbff
        slwi r9,r3,1
        ori r0,r0,64432
        mflr r31
        stwux r1,r1,r0
        add r11,r3,r9
        mr r4,r3
        mr r0,r3
        lwz r6,68(r1)
        add r0,r0,r11 ;c = argc + b[1]
        stw r0,17468(r1)
        mr r5,r9
        add r6,r3,r6
        stw r9,64(r1)
        addis r3,r31,ha16(LC0-L1$pb)
        stw r11,1084(r1)
        stw r9,1088(r1)
        la r3,lo16(LC0-L1$pb)(r3)
        mr r7,r9
        add r8,r4,r6
        bl L_printf$stub

; Clean up main's activation
; record and return 0:

        lwz r1,0(r1)
        li r3,0
        lwz r0,8(r1)
        lwz r31,-4(r1)
        mtlr r0
        blr
```

One thing that you'll notice in this version with optimization enabled is that GCC did not allocate variables in the activation record as they were encountered. Instead, it placed most of the objects in registers (even array elements). Keep in mind that an optimizing compiler may very well rearrange all the local variables you declare.

The ARM processor has similar limitations based on the size of the instruction opcode (32 bits). Here's the (unoptimized) ARM output from GCC:

```
.LC0:
    .ascii  "%d %d %d %d %d \000"

main:

    @ Set up activation record

    push    {fp, lr}
    add fp, sp, #4

    @ Reserve storage for locals.
    @ (2 instructions due to instruction
    @ size limitations).

    add sp, sp, #-67108864
    sub sp, sp, #1056

    @ Store argc (passed in R0)
    @ into a. Three additions
    @ (-67108864, 4, and -1044)
    @ are needed because of ARM
    @ 32-bit instruction encoding
    @ limitations

    add r3, fp, #-67108864
    sub r3, r3, #4
    str r0, [r3, #-1044]

    @ a = argc

    add r3, fp, #-67108864
    sub r3, r3, #4
    ldr r3, [r3, #-1044]    @ r3 = argc
    str r3, [fp, #-8]       @ a = argc

    @ b[0] = argc + argc

    add r3, fp, #-67108864
    sub r3, r3, #4
    ldr r2, [r3, #-1044]    @ R2 = argc
    ldr r3, [r3, #-1044]    @ R3 = argc
    add r3, r2, r3          @ R3 = argc + argc
    str r3, [fp, #-1040]    @ b[0] = argc+argc

    ldr r2, [fp, #-1040]    @ R2 = b[0]
    ldr r3, [fp, #-8]       @ R3 = a
    add r3, r2, r3          @ a + b[0]
    str r3, [fp, #-20]      @ b[255] = a  +b[0]

    ldr r2, [fp, #-1036]    @ R2 = b[1]
    add r3, fp, #-67108864
    sub r3, r3, #4
    ldr r3, [r3, #-1044]    @ R3 = argc
    add r3, r2, r3          @ argc + b[1]
    str r3, [fp, #-12]      @ c = argc + b[1]

    add r3, fp, #-67108864
    sub r3, r3, #4
    ldr r2, [r3, #-1044]    @ R2 = argc
    ldr r3, [fp, #-8]       @ R3 = a
    add r3, r2, r3          @ R3 = argc + a
    add r2, fp, #-67108864
    sub r2, r2, #4
    str r3, [r2, #-1036]    @ d[0] = argc + a

    ldr r2, [fp, #-20]      @ R2 = b[255]
    add r3, fp, #-67108864
    sub r3, r3, #4
    ldr r3, [r3, #-1044]    @ R3 = argc
    add r3, r2, r3          @ R3 = argc + b[255]
    add r2, fp, #-67108864
    sub r2, r2, #4
    add r2, r2, #12288
    str r3, [r2, #3056]     @ d[4095] = argc + b[255]

    ldr r2, [fp, #-8]       @ R2 = a
    ldr r3, [fp, #-12]      @ R3 = c
    add r3, r2, r3          @ R3 = a + c
    str r3, [fp, #-16]      @ e = a + c

    @ printf function call:

    ldr r1, [fp, #-1040]
    add r3, fp, #-67108864
    sub r3, r3, #4
    ldr r3, [r3, #-1036]
    ldr r2, [fp, #-16]
    str r2, [sp, #4]
    str r3, [sp]
    ldr r3, [fp, #-12]
    mov r2, r1
    ldr r1, [fp, #-8]
    ldr r0, .L3
    bl  printf
```

```
    @ return to Linux from function
    mov r3, #0
    mov r0, r3
    sub sp, fp, #4

    pop {fp, pc}

.L3:
    .word   .LC0
```

While this is arguably better than the PowerPC code, there's still considerable ugliness in the address computations because the ARM CPU cannot encode 32-bit constants as part of the instruction opcode. To understand why GCC emits such bizarre constants to compute offsets into the activation record, see the discussion of the ARM immediate operands in the section "The Immediate Addressing Mode" in Appendix C online.

If you find the optimized PowerPC or ARM code a bit hard to follow, consider the following 80x86 GCC output for the same C program:

```
.file   "t.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "%d %d %d %d %d "
        .text
        .p2align 2,,3
        .globl main
        .type   main,@function
main:
        ; Build main's activation record:

        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $67109892, %esp

        ; Fetch ARGC into ECX:

        movl    8(%ebp), %ecx

        ; EDX = 2*argc:

        leal    (%ecx,%ecx), %edx

        ; EAX = a (ECX) + b[0] (EDX):

        leal    (%edx,%ecx), %eax

        ; c (ebx) = argc (ecx) + b[1]:

        movl    %ecx, %ebx
        addl    -1028(%ebp), %ebx
        movl    %eax, -12(%ebp)
```

```
; Align stack for printf call:

andl    $-16, %esp

;d[0] (eax) = argc (ecx) + a (eax);

leal    (%eax,%ecx), %eax

; Make room for printf parameters:

subl    $8, %esp
movl    %eax, -67093516(%ebp)

; e = a + c

leal    (%ebx,%ecx), %eax


pushl   %eax    ;e
pushl   %edx    ;d[0]
pushl   %ebx    ;c
pushl   %edx    ;b[0]
pushl   %ecx    ;a
pushl   $.LC0
movl    %edx, -1032(%ebp)
movl    %edx, -67109896(%ebp)
call    printf
xorl    %eax, %eax
movl    -4(%ebp), %ebx
leave
ret
```

Of course, the 80x86 doesn't have as many registers to use for passing parameters and holding local variables, so the 80x86 code has to allocate more locals in the activation record. Also, the 80x86 provides an offset range of –128 to +127 bytes only around the EBP register, so a larger number of instructions have to use the 4-byte offset rather than the 1-byte offset. Fortunately, the 80x86 does allow you to encode a full 32-bit address as part of the instructions that access memory, so you don't have to execute multiple instructions in order to access a variable stored a long distance away from where EBP points in the stack frame.

15.5.5 Registers to the Rescue

As the examples in the previous section demonstrate, RISC code suffers greatly when it has to deal with parameters and local variables whose offsets are not easy to represent within the confines of the instruction opcode. In real code, however, the situation is not so dire. Compilers are smart enough to use machine registers to pass parameters and hold local variables providing immediate access to those values. This dramatically reduces the number of instructions in typical functions.

Consider the (register-starved) 32-bit 80x86 CPU. As there are only eight general-purpose registers, two of which (ESP and EBP) have special purposes that limit their use, there aren't a lot of registers available for passing parameters or holding local variables. Typical C compilers use EAX, ECX, and EDX to pass up to three parameters to a function. Functions return their result in the EAX register. The function must preserve the values of any other registers (EBX, ESI, EDI, and EBP) it uses. It's fortunate that memory access to local variables and parameters inside the function is very efficient—given the limited register set, the 32-bit 80x86 will need to use memory often for this purpose.

For most applications, the largest architectural improvement to the 64-bit x86-64 over the 32-bit 80x86 was not 64-bit registers (or even addresses), but that the x86-64 added eight new general-purpose registers and eight new XMM registers that compilers could use for passing parameters and holding local variables. The Intel/AMD ABI for the x86-64 allows a compiler to pass up to six different arguments in registers to a function (without the caller explicitly saving those register values before using them). Table 15-2 lists the available registers.

Table 15-2: Ix86-64 Argument Passing via Registers

Register	Usage
RDI	1st argument
RSI	2nd argument
RDX	3rd argument
RCX	4th argument
R8	5th argument
R9	6th argument
XMM0–XMM7	Used to pass floating-point arguments
R10	Can be used to pass static chain pointer
RAX	Used to pass argument count if there are a variable number of parameters

The 32-bit ARM (A32) ABI specifies up to four arguments appearing in registers R0 through R3. As the A64 architecture has twice as many registers (32), the A64 ABI is a bit more generous, passing up to eight 64-bit integer/pointer arguments in R0 through R7 and up to eight additional floating-point parameters in V0 through V7.

The PowerPC ABI, which has 32 general-purpose registers, sets aside R3 through R10 to pass up to eight arguments to a function. It also sets aside the F1 through F8 floating-point registers to pass floating-point arguments to a function.

In addition to setting aside registers to hold function arguments, the various ABIs typically define various registers that a function can use to hold local variables or temporary values (without explicitly preserving the values held in those registers upon entry to the function). For example, the Windows ABI sets aside R11, XMM8 through XMM15, MMX0 through MMX7, the FPU registers, and RAX for temporary/local use. The ARM A32 ABI sets aside R4 through R8 and R10 through R11 for use as locals. The A64 ABI sets aside R9 through R15 for locals and temporaries. The PowerPC sets aside R14 through R30 and F14 through F31 for local variables. Once a compiler exhausts the registers the ABI defines for argument passing, most ABIs expect the calling code to pass additional parameters on the stack. Similarly, once a function uses all the registers set aside for local variables, additional local variable allocation occurs on the stack.

Of course, a compiler can use other registers for local and temporary values as well as those set aside by the CPU's or OS's ABI. However, it's the compiler's responsibility to preserve those register values across the function call.

NOTE *An ABI is a* convention*, not a requirement by the underlying OS or hardware. Compiler writers (and assembly language programmers) who stick to a given ABI can expect that their object code modules will be able to interact with code written in other languages that adhere to that ABI. However, nothing stops a compiler writer from using whatever mechanism they choose.*

15.5.6 Java VM and Microsoft CLR Parameters and Locals

Because the Java VM and Microsoft CLR are both virtual stack machines, programs compiled to those two architectures always push function arguments onto the stack. Beyond that, the two virtual machine architectures diverge. The reason for the divergence is that the Java VM's design supports efficient interpretation of Java bytecodes with JIT compilation improving performance as needed. The Microsoft CLR, on the other hand, does not support interpretation; instead, the CLR code (CIL) design supports efficient JIT compilation to optimized machine code.

The Java VM is a traditional stack architecture, with parameters, locals, and temporaries sitting on the stack. Other than the fact that there are no registers to use for such objects, Java's memory organization is very similar to that of the 80x86/x86-64, PowerPC, and ARM CPUs. During JIT compilation, it can be difficult to figure out which values on the stack can be moved into registers and which local variables the Java compiler allocates on the stack can be allocated in registers. Optimizing such stack allocations to use registers can be very time-consuming, so it's doubtful that the Java JIT compiler does this while the application is running (as doing so would greatly diminish the application's runtime performance).

Microsoft's CLR operates under a different philosophy. CIL is always JIT-compiled into native machine code. Furthermore, Microsoft's intent is to have the JIT compiler produce *optimized* native machine code. While the JIT compiler rarely does as good a job as a traditional C/C++ compiler, it generally does a much better job than the Java JIT compiler. This is because the Microsoft CLR definition explicitly singles out parameter argument and local variable memory accesses. When the JIT compiler sees these special instructions, it can allocate those variables to registers rather than memory locations. As a result, CLR JIT-compiled code is often shorter and faster than Java VM JIT-compiled code (especially on RISC architectures).

15.6 Parameter-Passing Mechanisms

Most high-level languages provide at least two mechanisms for passing actual parameter data to a subroutine: pass-by-value and pass-by-reference.[11] In languages like Visual Basic, Pascal, and C++, declaring and using both types of parameters is so easy that a programmer may conclude that there's little difference in efficiency between the two mechanisms. That's a myth this section intends to dispel.

NOTE *There are other parameter-passing mechanisms besides pass-by-value and pass-by-reference. FORTRAN and HLA, for example, support a mechanism known as pass-by-value/result (or pass-by-value/returned). Ada and HLA support pass-by-result. HLA and Algol support pass-by-name. This book won't discuss these alternative parameter-passing mechanisms further, because you probably won't see them very often. If you'd like more information, consult a good book on programming language design or the HLA documentation.*

15.6.1 Pass-by-Value

Pass-by-value is the easiest parameter-passing mechanism to understand. The code that calls a procedure makes a copy of the parameter's data and passes this copy to the procedure. For small values, passing a parameter by value generally requires little more than a `push` instruction (or, when passing parameters in the registers, an instruction that moves the value into a register). Therefore, this mechanism is often very efficient.

One big advantage of pass-by-value parameters is that the CPU treats them just like a local variable within the activation record. Because you'll rarely have more than 120 bytes of parameter data that you pass to a procedure, CPUs that provide a shortened displacement with the indexed addressing mode will be able to access most parameter values using a shorter (and, therefore, more efficient) instruction.

The one case where passing a parameter by value can be inefficient is when you need to pass a large data structure, such as an array or record. The calling code needs to make a byte-for-byte copy of the actual parameter into the procedure's activation record, as you saw in an earlier example. This can be a very slow process, say, if you decide to pass a million-element array to a subroutine by value. Therefore, you should avoid passing large objects by value unless absolutely necessary.

11. C allows only pass-by-value, but it lets you take an address of some object so that you can easily simulate pass-by-reference. C++ fully supports pass-by-reference parameters.

15.6.2 Pass-by-Reference

The pass-by-reference mechanism passes the address of an object rather than its value. This has a couple of distinct advantages over pass-by-value. First, regardless of the parameter's size, pass-by-reference parameters always consume the same amount of memory—the size of a pointer (which typically fits in a machine register). Second, pass-by-reference parameters allow you to modify the value of the actual parameter—which is impossible with pass-by-value parameters.

Pass-by-reference parameters are not without their drawbacks, though. Usually, accessing a reference parameter within a procedure is more expensive than accessing a value parameter, because the subroutine needs to dereference that address on each access of the object. This generally involves loading a register with the pointer in order to dereference the pointer using a register indirect addressing mode.

For example, consider the following Pascal code:

```
procedure RefValue
 (
    var dest:integer;
    var passedByRef:integer;
        passedByValue:integer
);
begin

    dest := passedByRef + passedByValue;

end;
```

Here's the equivalent HLA/x86 assembly code:

```
procedure RefValue
(
var     dest:int32;
var     passedByRef:int32;
            passedByValue:int32
); @noframe;
begin RefValue;

    // Standard entry sequence (needed because of @noframe).
    // Set up base pointer.
    // Note: don't need SUB(nn,esp) because
    // we don't have any local variables.

    push( ebp );
    mov( esp, ebp );

    // Get pointer to actual value.

    mov( passedByRef, edx );

    // Fetch value pointed at by passedByRef.
```

```
        mov( [edx], eax );

        // Add in the value parameter.

        add( passedByValue, eax );

        // Get address of destination reference parameter.

        mov( dest, edx );

        // Store sum away into dest.

        mov( eax, [edx] );

        // Exit sequence doesn't need to deallocate any local
        // variables because there are none.

        pop( ebp );
        ret( 12 );

    end RefValue;
```

Notice that this code requires two more instructions than a version that uses pass-by-value—specifically, the two instructions that load the addresses of `dest` and `passedByRef` into the EDX register. In general, only a single instruction is needed to access the value of a pass-by-value parameter. However, two instructions are needed to manipulate the value of a parameter when you pass it by reference (one instruction to fetch the address, and one to manipulate the data at that address). So, unless you need the semantics of pass-by-reference, try to use pass-by-value instead.

The issues with pass-by-reference tend to diminish when your CPU has lots of available registers that it can use to maintain the pointer values. In that situation, the CPU can use a single instruction to fetch or store a value via a pointer maintained in the register.

15.7 Function Return Values

Most HLLs return function results in one or more CPU registers. Exactly which register the compiler uses depends on the data type, CPU, and compiler. For the most part, however, functions return their results in registers (assuming the return data fits in a machine register).

On the 32-bit 80x86, most functions that return ordinal (integer) values return their function results in the AL, AX, or EAX register. Functions that return 64-bit values (`long long int`) generally return the function result in the EDX:EAX register pair (with EDX containing the HO double word of the 64-bit value). On 64-bit variants of the 80x86 family, 64-bit compilers return 64-bit results in the RAX register. On the PowerPC, most compilers follow the IBM ABI and return 8-, 16-, and 32-bit values in the R3 register.

Compilers for the 32-bit versions of the PowerPC return 64-bit ordinal values in the R4:R3 register pair (with R4 containing the HO word of the function result). Presumably, compilers running on 64-bit variants of the PowerPC can return 64-bit ordinal results directly in R3.

Generally, compilers return floating-point results in one of the CPU's (or FPU's) floating-point registers. On 32-bit variants of the 80x86 CPU family, most compilers return a floating-point result in the 80-bit ST0 floating-point register. Although the 64-bit versions of the 80x86 family also provide the same FPU registers as the 32-bit members, some operating systems, such as Windows64, typically use one of the SSE registers (XMM0) to return floating-point values. PowerPC systems generally return floating-point function results in the F1 floating-point register. Other CPUs return floating-point results in comparable locations.

Some languages allow a function to return a nonscalar (aggregate) value. The exact mechanism that compilers use to return large function return results varies from compiler to compiler. However, a typical solution is to pass a function the address of some storage where the function can place the return result. As an example, consider the following short C++ program whose `func()` function returns a structure object:

```
#include <stdio.h>

typedef struct
{
    int a;
    char b;
    short c;
    char d;
} s_t;

s_t func( void )
{
    s_t s;

    s.a = 0;
    s.b = 1;
    s.c = 2;
    s.d = 3;
    return s;
}

int main( void )
{
    s_t t;

    t = func();
    printf( "%d %d", t.a, func().a );
    return( 0 );
}
```

Here's the PowerPC code that GCC emits for this C++ program:

```
.text
        .align 2
        .globl _func

; func() -- Note: upon entry, this
;           code assumes that R3
;           points at the storage
;           to hold the return result.

_func:
        li r0,1
        li r9,2
        stb r0,-28(r1) ; s.b = 1
        li r0,3
        stb r0,-24(r1) ; s.d = 3
        sth r9,-26(r1) ; s.c = 2
        li r9,0        ; s.a = 0

        ; Okay, set up the return
        ; result.

        lwz r0,-24(r1) ; r0 = d::c
        stw r9,0(r3)   ; result.a = s.a
        stw r0,8(r3)   ; result.d/c = s.d/c
        lwz r9,-28(r1)
        stw r9,4(r3)   ; result.b = s.b
        blr


        .data
        .cstring
        .align 2
LC0:
        .ascii "%d %d\0"
        .text
        .align 2
        .globl _main
_main:
        mflr r0
        stw r31,-4(r1)
        stw r0,8(r1)
        bcl 20,31,L1$pb
L1$pb:
        ; Allocate storage for t and
        ; temporary storage for second
        ; call to func:

        stwu r1,-112(r1)

        ; Restore LINK from above:

        mflr r31
        ; Get pointer to destination
        ; storage (t) into R3 and call func:

        addi r3,r1,64
        bl _func

        ; Compute "func().a"

        addi r3,r1,80
        bl _func

        ; Get t.a and func().a values
        ; and print them:

        lwz r4,64(r1)
        lwz r5,80(r1)
        addis r3,r31,ha16(LC0-L1$pb)
        la r3,lo16(LC0-L1$pb)(r3)
        bl L_printf$stub
        lwz r0,120(r1)
        addi r1,r1,112
        li r3,0
        mtlr r0
        lwz r31,-4(r1)
        blr

; stub for printf function:

        .data
        .picsymbol_stub
L_printf$stub:
        .indirect_symbol _printf
        mflr r0
        bcl 20,31,L0$_printf
L0$_printf:
        mflr r11
        addis r11,r11,ha16(L_printf$lazy_ptr-L0$_printf)
        mtlr r0
        lwz r12,lo16(L_printf$lazy_ptr-L0$_printf)(r11)
        mtctr r12
        addi r11,r11,lo16(L_printf$lazy_ptr-L0$_printf)
        bctr
        .data
        .lazy_symbol_pointer
L_printf$lazy_ptr:
        .indirect_symbol _printf
        .long dyld_stub_binding_helper
```

Here's the 32-bit 80x86 code that GCC emits for this same function:

```
.file   "t.c"
        .text
        .p2align 2,,3
        .globl func
        .type   func,@function

; On entry, assume that the address
; of the storage that will hold the
; function's return result is passed
; on the stack immediately above the
; return address.

func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp       ; Allocate storage for s.

        movl    8(%ebp), %eax   ; Get address of result
        movb    $1, -20(%ebp)   ; s.b = 1
        movw    $2, -18(%ebp)   ; s.c = 2
        movb    $3, -16(%ebp)   ; s.d = 3
        movl    $0, (%eax)      ; result.a = 0;
        movl    -20(%ebp), %edx ; Copy the rest of s
        movl    %edx, 4(%eax)   ; to the storage for
        movl    -16(%ebp), %edx ; the return result.
        movl    %edx, 8(%eax)
        leave
        ret     $4
.Lfe1:
        .size   func,.Lfe1-func
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "%d %d"

        .text
        .p2align 2,,3
        .globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp       ; Allocate storage for
        andl    $-16, %esp      ; t and temp result.

        ; Pass the address of t to func:

        leal    -24(%ebp), %eax
        subl    $12, %esp
        pushl   %eax
        call    func

        ; Pass the address of some temporary storage
        ; to func:

        leal    -40(%ebp), %eax
        pushl   %eax
        call    func
        ; Remove junk from stack:

        popl    %eax
        popl    %edx

        ; Call printf to print the two values:

        pushl   -40(%ebp)
        pushl   -24(%ebp)
        pushl   $.LC0
        call    printf
        xorl    %eax, %eax
        leave
        ret
```

The takeaway from these 80x86 and PowerPC examples is that functions returning large objects often copy the function result data just prior to returning. This extra copying can take considerable time, especially if the return result is large. Instead of returning a large structure as a function result, as shown here, it's usually better to explicitly pass a pointer to some destination storage to a function that returns a large result and then let the function do whatever copying is necessary. This often saves some time and code. Consider the following C code, which implements this policy:

```
#include <stdio.h>

typedef struct
{
    int a;
    char b;
    short c;
    char d;
} s_t;

void func( s_t *s )
{
    s->a = 0;
    s->b = 1;
    s->c = 2;
    s->d = 3;
    return;
}

int main( void )
{
    s_t s,t;

    func( &s );
    func( &t );
    printf( "%d %d", s.a, t.a );
    return( 0 );
}
```

Here's the conversion to 80x86 code by GCC:

```
        .file   "t.c"
        .text
        .p2align 2,,3
        .globl func
        .type   func,@function
func:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    $0, (%eax)      ; s->a = 0
        movb    $1, 4(%eax)     ; s->b = 1
        movw    $2, 6(%eax)     ; s->c = 2
        movb    $3, 8(%eax)     ; s->d = 3
        leave
        ret
.Lfe1:
        .size   func,.Lfe1-func
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "%d %d"
        .text
        .p2align 2,,3
        .globl main
        .type   main,@function
main:
        ; Build activation record and allocate
        ; storage for s and t:

        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        andl    $-16, %esp
        subl    $12, %esp

        ; Pass address of s to func and
        ; call func:

        leal    -24(%ebp), %eax
        pushl   %eax
        call    func

        ; Pass address of t to func and
        ; call func:

        leal    -40(%ebp), %eax
        movl    %eax, (%esp)
        call    func

        ; Remove junk from stack:

        addl    $12, %esp
```

```
; Print the results:

pushl   -40(%ebp)
pushl   -24(%ebp)
pushl   $.LC0
call    printf
xorl    %eax, %eax
leave
ret
```

As you can see, this approach is more efficient because the code doesn't have to copy the data twice, once to a local copy of the data and once to the final destination variable.

15.8 For More Information

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* 2nd ed. Essex, UK: Pearson Education Limited, 1986.

Barrett, William, and John Couch. *Compiler Construction: Theory and Practice.* Chicago: SRA, 1986.

Dershem, Herbert, and Michael Jipping. *Programming Languages, Structures and Models.* Belmont, CA: Wadsworth, 1990.

Duntemann, Jeff. *Assembly Language Step-by-Step.* 3rd ed. Indianapolis: Wiley, 2009.

Fraser, Christopher, and David Hansen. *A Retargetable C Compiler: Design and Implementation.* Boston: Addison-Wesley Professional, 1995.

Ghezzi, Carlo, and Jehdi Jazayeri. *Programming Language Concepts.* 3rd ed. New York: Wiley, 2008.

Hoxey, Steve, Faraydon Karim, Bill Hay, and Hank Warren, eds. *The PowerPC Compiler Writer's Guide.* Palo Alto, CA: Warthman Associates for IBM, 1996.

Hyde, Randall. *The Art of Assembly Language.* 2nd ed. San Francisco: No Starch Press, 2010.

———. "Webster: The Place on the Internet to Learn Assembly." *http://plantation-productions.com/Webster/index.html.*

Intel. "Intel 64 and IA-32 Architectures Software Developer Manuals." Updated November 11, 2019. *https://software.intel.com/en-us/articles/intel-sdm.*

Ledgard, Henry, and Michael Marcotty. *The Programming Language Landscape.* Chicago: SRA, 1986.

Louden, Kenneth C. *Compiler Construction: Principles and Practice.* Boston: Cengage, 1997.

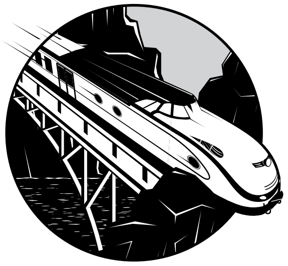
Louden, Kenneth C., and Kenneth A. Lambert. *Programming Languages: Principles and Practice.* 3rd ed. Boston: Course Technology, 2012.

Parsons, Thomas W. *Introduction to Compiler Construction.* New York: W. H. Freeman, 1992.

Pratt, Terrence W., and Marvin V. Zelkowitz. *Programming Languages, Design and Implementation.* 4th ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Sebesta, Robert. *Concepts of Programming Languages.* 11th ed. Boston: Pearson, 2016.

AFTERWORD: ENGINEERING SOFTWARE



The goal of this volume was to help you consider the impact of your high-level coding techniques on the resulting machine code generated by the compiler. Unless you understand the cost tradeoffs of statements and data structures in your HLL programs, you won't be able to produce efficient programs consistently. And if you want to write great code, you can't write inefficient programs. To that end, the first two books of this series, *Understanding the Machine* and *Thinking Low-Level, Writing High-Level,* have addressed efficiency concerns facing modern programmers. However, as noted in Chapter 1, efficiency isn't the only attribute of great code. Thus, the next volume, *Engineering Software*, will change direction and discuss some of the other attributes.

Specifically, Volume 3 begins discussing the personal software engineering aspects of programming. The software engineering field focuses primarily on the management of large software systems. Personal software engineering, on the other hand, covers those topics germane to writing great code at a personal level—craftsmanship, art, and pride in one's work. So, in *Engineering Software*, we'll consider those aspects through discussions on software development metaphors, software developer metaphors, and system documentation, among other topics.

Congratulations on your progress thus far toward writing great code. See you in Volume 3.

GLOSSARY

A

A32 32-bit ARM CPUs.

A64 ARMv8 and later architecture CPUs that support 64-bit registers and operations.

ABI Application binary interface

Activation record A section of memory associated with a subroutine/function invocation that includes a return address, function arguments, local variables, and other function-related information.

Ahead-of-time compilation Translation of a machine-independent bytecode into native machine code prior to execution.

Allocation granularity (memory manager) The minimum-sized block a memory allocator will allocate for a storage request.

AOT Ahead of time. See *Ahead-of-time compilation.*

Attribute A feature associated with some object. Examples include the object's name, type, memory address, and value.

B

Basic block A sequence of machine instructions into and out of which there are no branches except at the beginning and end of the sequence.

Basic multilingual plane The first group of 65,536 Unicode code points (`U+0000` to `U+CFFF` and `U+E000` to `U+FFFF`).

Binding The process of associating an attribute (such as a name, type, value, or address) with some object.

BMP See *Basic multilingual plane.*

BSS Block started by a symbol (uninitialized data section in an object code file).

C

Call tree A graphical diagram of all the calls in a program, or *rooted* at a particular function in an application. Arrows coming out of the root node represent calls made by the function/procedure depicted by that root node. These arrows point at the functions called by the node. Each such node may contain calls of its own to other nodes (represented by arrows leaving the calling node). Technically, a call tree is, more generally, a call *graph* because (indirect) recursive calls could result in arrows to higher nodes in the call tree.

Calling convention The sequence of machine instructions that pass parameters to and invoke the execution of a function/procedure.

Canonical equivalence Two different sequences (such as strings) are canonically equivalent if they produce the same character on an output device. If two strings are canonically equivalent, comparing them for equality should produce `true` even if they have different bytes in their sequences.

CIL Common Intermediate Language (Microsoft .NET)

CISC Complex instruction set computer

CLI Common Language Infrastructure (Microsoft)

CLR Common Language Runtime (Microsoft .NET)

CLS Common Language Specification

Code motion An optimization technique whereby a compiler moves a section of code to some other location where it will execute more efficiently (for example, moving loop-invariant code outside of a loop).

Code plane A set of up to 65,536 different Unicode characters.

Code point A numeric value (in the range 0–65,535) representing a Unicode character (scalar) or a surrogate code point (Unicode character set expansion).

COFF Common Object File Format

Column-major ordering A mechanism for storing elements of a multidimensional array in memory where consecutive memory locations hold elements whose first index changes more rapidly than the other indices into the array.

Common subexpression elimination A compiler optimization that preserves the value of some (sub)expression for later use to avoid recomputing the expression's value.

Complete Boolean evaluation Evaluating all components of a Boolean expression, even if some subcomponents will not affect the final value of the computation.

CTS Common Type System

D

Dangling pointers Pointers that an application continues to use after the application has freed them and made their allocated memory available for other use.

DBCS Double-byte character set

Dead code elimination A compiler optimization that removes all code from an object module that can never execute.

DFA Data flow analysis. Also Deterministic Finite-state Automata.

Display A data structure (typically in an activation record) that provides pointers to access intermediate variables appearing in nested procedures and functions.

Dope vector An array of integers that specifies the bounds of a dynamically allocated array.

E

ECMA European Computer Manufacturers Association

ELF Executable and Linkable File format

Enumerated data type A data type whose (constant) values are a list of symbolic names to which the system associates a unique numeric value (*enumerate* means to count or to list).

F

Filter programs Programs that read a file, process its input, and produce an output file based on that input.

FSF Free Software Foundation

G

Gas GNU Assembler

Glyph A set of strokes that draw a single character on an output device.

GNU Gnu's Not Unix

Grapheme cluster A sequence of one or more Unicode code points that define a single character (glyph). A sequence of Unicode code points that produce a single item most people would recognize as a stand-alone character on an output device.

H

Heap A special memory area where a language's runtime system allocates and deallocates storage for dynamic variables.

HLA High-Level Assembly language

HLL High-level language

HO High order

I

I/O Input/output

IDE Integrated development environment

IL Intermediate Language (Microsoft)

ILAsm Intermediate Language Assembly. The assembly language syntax for the Microsoft CLR.

Indirect recursion A function is indirectly recursive if it calls some other function (which can call some other function, and so on) that ultimately calls the original function before returning to that function.

Inline function A function whose body a compiler expands in place at the point of the function invocation rather than emitting a call to the function.

J

JBC Java bytecode

JIT Just in time (compilation)

L

Leaf procedures and functions Procedures/functions that do not call any other procedures or functions (that is, they are "leaf" nodes in a call tree). See also *call tree.*

LIFO Last-in, first-out (organization of a stack data structure).

LINK register Holds a function return address upon entry into a function (PowerPC). See also *LR.*

LO Low-order

Local variables Those variables whose scope is limited to a sequence of statements associated with some block of code (usually a function or procedure).

Loop invariant A calculation occurring inside a loop whose value does not change on each iteration of the loop.

LR Link register (on the ARM CPU). Holds a function's return address upon entry into the function.

M

Macro A body of text that a compiler substitutes in place of the macro invocation in some source code.

Managed pointers Pointers that have certain restrictions placed on their operations to help eliminate common problems that occur with unmanaged pointers (that is, those that allow arbitrary pointer operations).

Manifest constant A constant value associated with a symbolic name. Upon encountering the symbolic name, the compiler directly replaces it with the constant value.

MASM Microsoft Assembler

Memory leak Continuously allocating memory without ever freeing that memory, even if it is never again used. Such memory becomes inaccessible to the system.

Metadata Data in a file that describes other data in the file.

MSIL Microsoft Intermediate Language

MSVC Microsoft Visual C (++)

Multiple inheritance The ability for a class to inherit attributes (data fields) and behaviors (methods/functions) from multiple parent classes.

N

Namespace pollution Having so many names in a given scope that the name you want to give a new object is already in use elsewhere in your program, potentially leading to a name conflict.

NaN Not a number (floating-point result)

NASM Netwide assembler

Next on stack The value immediately below the top of stack. See also *NOS.*

NOS Next on stack

O

One-address machine See *single-address machine.*

Opaque data type A data type whose internal implementation is not visible to the programmer.

Opcode prefix byte A special machine instruction value that modifies the operation of the instruction immediately following. For example, on the 80x86, an opcode size prefix byte could specify a different memory/register operand size for the following instruction.

P

PC Personal computer

PE/COFF Portable executable, common object file format (Microsoft)

Plain vanilla text An ASCII or Unicode text file that contains only textual information, without any special formatting.

R

Recursion A call to a function from that function itself. See also *indirect recursion.*

RISC Reduced instruction set computer

Row-major ordering A mechanism for storing elements of a multidimensional array in memory where consecutive memory locations hold elements whose last index changes more rapidly than the other indices into the array.

S

SBC Single-board computer

Sentinel A special value that marks some boundary of a data sequence—for example, a zero-terminating byte at the end of a string of characters.

Sequence point The point in a computation where a compiler guarantees that all previous side effects have been computed/completed.

Short-circuit evaluation Ignoring certain parts of a computation that won't affect the overall results of that computation.

Side effect A result from some computation that is not the primary intended result of that computation. Typically, this involves the modification of values (variables) or other program state beyond the primary intent of the calculation.

SIMD Single-instruction, multiple data (instructions for a CPU).

Single-address machine CPUs whose arithmetic and logical instructions have a single operand. This is usually an accumulator-based architecture.

Spaghetti code Code containing lots of control transfer (`goto`) statements that make it difficult to determine the flow of control in the program.

Stack frame See *activation record.*

Stack-based machine A CPU that performs all calculations on a hardware stack (rather than using machine registers).

Static binding Binding (associating attributes with an object) that occurs prior to the execution of a program.

Straight-line code A sequence of instructions that contain no branches, conditionals, function calls, or anything else that causes a transfer of control.

Surrogate code points Special Unicode values that expand the character set beyond 65,536 characters (expansion beyond 16 bits).

T

TASM Turbo assembler (Borland, Embarcadero)

Thrashing Constantly loading values that are not present in the cache (or in a memory page, versus present in virtual storage). Loading data into the cache (or a memory page) may evict data that the application will soon access, leading to even more thrashing.

Three-address machines CPUs whose arithmetic instructions typically have three operands: a destination and two source operands. Most three-address machines are register-based machines.

Tokenized Data that has had words or other lexemes replaced by numeric (typically single-byte) "token" values.

TOS Top of stack

Tuple A list of associated data values. In Swift, a tuple is roughly equivalent to a list of values.

Two-address machines CPUs whose arithmetic instructions typically have two operands: a destination and a source operand (the destination operand doubles as a source operand). Most two-address machines are register-based machines.

U

Unicode A universal standardized character set that supports most known characters.

Unicode normalization Adjusting canonically equivalent Unicode strings so that they have the same (minimal) code points, organized in the same order.

Unrolling (loops) Expanding the code appearing in a loop, once for each iteration of the (fixed iterations) loop. Improves performance by eliminating loop overhead code.

UTF Universal Transformation Format; an encoding scheme for Unicode (UTF-8, UTF-16, and UTF-32 are the three standard Unicode encoding schemes).

V

VB Visual BASIC

VC++ Visual C++ (Microsoft)

VFP Vector floating-point (ARM instructions).

VHLL Very high-level language

VM Virtual machine

W

WGC1 *Write Great Code, Volume 1: Understanding the Machine*

X

x86-64 64-bit variant of the AMD/Intel 80x86 CPU.

Z

Zero-address machine A CPU whose arithmetic and logical instructions do not specify any operands. Typically, this is a stack machine architecture.

ONLINE APPENDIXES

Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level, includes supplementary materials online at *https://nostarch.com/writegreatcode2_2/* and *http://www.writegreatcode.com/*. Among other resources, these five appendixes are published in electronic form so they can be easily updated and downloaded:

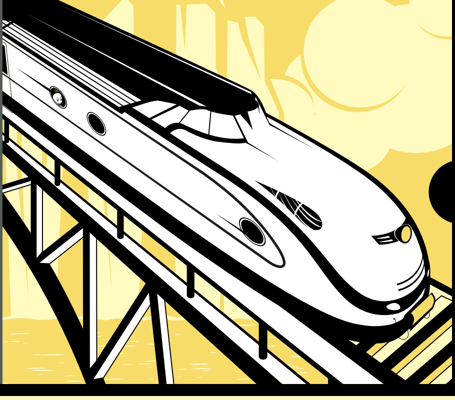
- Appendix A: The Minimal x86 Instruction Set
- Appendix B: PowerPC Assembly for the HLL Programmer
- Appendix C: ARM Assembly for the HLL Programmer
- Appendix D: Java Bytecode Assembly for the HLL Programmer
- Appendix E: CIL Assembly for the HLL Programmer

INDEX

RESOURCES

Visit *https://nostarch.com/writegreatcode2_2/* for resources, errata, and more information.

GET BETTER RESULTS FROM YOUR SOURCE CODE



NO PRIOR KNOWLEDGE OF ASSEMBLY LANGUAGE REQUIRED!

Today's programming languages offer productivity and portability, but also make it easy to write sloppy code that isn't optimized for a compiler. *Thinking Low-Level, Writing High-Level* will teach you to craft source code that results in good machine code once it's run through a compiler.

You'll learn:

- How to analyze the output of a compiler to verify that your code generates good machine code
- The types of machine code statements that compilers generate for common control structures, so you can choose the best statements when writing HLL code
- Enough assembly language to read compiler output
- How compilers convert various constant and variable objects into machine data

With an understanding of how compilers work, you'll be able to write source code that they can translate into elegant machine code.

NEW COVERAGE OF:

- Programming languages like Swift and Java
- Code generation on modern 64-bit CPUs
- ARM processors on mobile phones and tablets
- Stack-based architectures like the Java Virtual Machine
- Modern language systems like the Microsoft Common Language Runtime

ABOUT THE AUTHOR

Randall Hyde is the author of *The Art of Assembly Language* and the three volume *Write Great Code* series (all No Starch Press). He is also the co-author of *The Waite Group's MASM 6.0 Bible*. He has written for *Dr. Dobb's Journal* and *Byte*, and professional and academic journals.

$49.95 ($65.95 CDN)

ISBN 978-1-7185-0038-9
54995

9 781718 500389

SHELVE IN: COMPUTERS/PROGRAMMING