C as a Portable Intermediate Language

Gene Michael Stover

created Thursday, 2002 July 11
updated Monday, 2006 January 9

Contents

1 Introduction

Currently[1], there is interest in compiling source code to instructions for virtual machines in an attempt to write portable applications. That approach is commonly called Write Once Run Anywhere (WORA), but it's possible to achieve the same goals of portable source code by using C instead of virtual machine byte-codes as the portable intermediate language. What's more, virtual machine byte-codes have disadvantages that C does not.

2 Using C As An Intermediate Language

Here's a description of what I mean when I suggest compiling to C as a portable intermediate language.

Say you want to write programs in some source language *L*. It could be any programming language. Language *L* could be Java, C++, Ada, Lisp, Smalltalk, Perl, Pascal, Algol, Fortran, Cobol, Bourne shell, C itself, or any other programming language. Literally, *L* can be any programming language you can implement.

So you write your program in *L*.

You feed your program to a compiler that compiles *L* to C.

Then you feed the C program to a compiler that produces native object code. You process the object code to produce a natively runnable program in whatever way the native platform requires.

Now you have a natively runnable program compiled from the program you write in *L*. You run that program.

That's it. There's no magic. You compile *L* to native code via C.

3 Portable Application Delivery

One advantage of compiling to C is portable applications.

Let's say you are an open source or free software developer. (I'll discuss shrink wrapped software distribution later.) You can distribute your software to users who don't have compilers for *L*.

To make a distribution file, you compile your program to C & stop there. You package that.

Your user obtains a copy of the distribution archive, expands it, & runs a configuration step, then runs the C compiler on the intermediate C code that

[1] "Currently" is the year 2002.

you compiled & distributed. The user then has a natively executable version of your program.

Okay, you're saying "My users are not technical. I can't require them to run a C compiler; even a configuration step is asking a lot of them."

I reply, "You are absolutely correct. You are also unimaginative." I'll describe ways of hiding the configuration & installation steps on Windows, Mac, & Unix so that your user sees exactly the type of installation process that's usual for his platform. Let's start with Unix.

One common configuration & installation process on Unix is "`./configure; make all check install`". It doesn't take much imagination to see how C as an intermediate language can translate to this just fine. `./configure` determines system settings, as it always has. Then "`make all`" compiles the program, but when you (the programmer) created the distribution archive, you compiled the *L* source files to C, so your user's computer only needs to have a C compiler. In fact, there's precedent for this in the C files derived from (in other words, compiled from) the Bison & Flex source files in Gnu's gcc compiler.

So much for Unix.

Mac & Windows use graphical installation programs, but the same principles apply. During the configuration stage, the installation program can, well, determine the configuration. Then it can run make (or an equivalent program) & a C compiler to produce native executables. The user doesn't need to be aware that the C compiler is running. He just sees the progress bar that tells how much of the process has been done & how much remains.

Okay, now you are legitimately pointing out that the average Mac & Windows computer does not have a C compiler because those operating systems don't ship with C compilers. This is a problem that I hope would be rectified if the idea of C as a portable intermediate language became popular. Notice that the target host only needs a basic C compiler & a simple `make`, not a huge integrated development environment. The alternative of delivering a program compiled to byte-codes for a virtual machine requires that the user has a virtual machine & its complete run-time environment installed, & that's surely larger than a plain C compiler & `make`. Even worse, most Java applications I've seen ship with the entire run-time.

So yes, the technique of C as an intermediate language has a stumbling block of requiring a plain C compiler & a `make` (or equivalent) on those platforms, but it's no more painful than the problems VM-based applications cause. More on this later.

4 Overview of the Technique

I've described the two parts of the technique I'm suggesting. For clarity, Figure 1 shows the steps to the technique again.

1. You write your program in whatever language you want. We're calling that language L.

2. During development, you run your build process with `make` (or whatever conventions you use on your development platform). It compiles your *L* source files to C, then compiles the C to native. You run the program. You edit the *L* source files to fix bugs. In other words, you're programming in *L*, & you don't worry about C.

3. To create a distribution, you compile the *L* source files to C & then archive the *L* source files, the C source files, & whatever else you want. You deliver this archive to your users (or make it available for download or whatever).

4. You users run the exact same type of installation process that is customary on their platform. They must have a basic C compiler & a `make` (or equivalent), but they don't need to be aware that it's running.

5. Your users end up with a native executable.

Figure 1: The steps to distributing a program written in *L* & compiled to C as an intermediate language.

5 Shrink-Wrapped Software

So let's say you deliver shrink-wrapped software, & your source code is proprietary, neither open nor free. How does C as an intermediate language translate to you?

Answer: It translates just fine. You use the same development steps & distribution steps I described for open source & free software developers, except that when you create your distribution archive, you don't include the *L* source files the way the open source & free software developers did. You include the C files & other files required to compile your program, but not the *L* files.

You might be thinking "But my algorithms are in the C source files so that people (programmers at least) can read them, & I need to keep them secret".

Not quite true. Your algorithms were translated to C, true, but it's not a C that's meant for humans to read. When I say that we compiled *L* sources to C, I mean we really *compiled* it. The *L*-to-C compiler sucked up your *L* source files, analyzed them, & produced C code for a C compiler. It didn't produce C code for another human; it's not one of those "please can someone give me a program that converts Pascal to C so I can have all my Pascal programs in C" programs that newbie programmers sometimes request on Usenet. It's a real compiler interested in converting the run-time semantics of your source code to C, but it doesn't make human-readable or human-maintainable C code. Even the symbol names from your *L* sources are lost. (More on this topic later.) So your secrets are safe.

And if you want to distribute executable, binary files (as most shrink-

wrapped software is nowadays, anyway), you still benefit from compiling *L* to C as an intermediate language because you need only one *L*-to-C compiler, which can run on whatever computer type you want. You only need C-to-native compilers on the types of systems you want to support.

6 C Is Ubiquitous

A benefit to compiling to C as an intermediate language is that C is ubiquitous.[2] So if someone implements just one *L*-to-C compiler (hopefully in C), then language *L* is available on nearly every computer in existence.

7 Write Your Compiler Once

There can be just one *L*-to-C compiler. It can be implemented & debugged once. That's Write Once Run Anywhere (WORA).

8 Easy Compiler Implementation

It's fairly easy to write a compiler whose output language is C. It's a lot easier than writing a compiler whose output language is machine code. It is similarly fairly easy to debug a compiler whose output language is C instead of machine code.

9 No Performance Penalty

Because *L* was compiled to C, then to native, your program executes at native speed; it does not have the performance penalty of a virtual machine. Sure sure sure, a virtual machine with a Just In Time (JIT) compiler can convert the VM's byte-codes to native code to overcome the performance penalty of a VM, but why bother? You can compile *L* to C to native & be done with it.

And why bother to implement the JIT, which is just a compiler that outputs native code, when someone has already implemented such a compiler (the C compiler)? Ever hear of "code reuse"?

These days, C compilers produce efficient code. So though an *L*-to-native compiler could produce smaller or faster code than *L*-to-C-to-native, the difference will be minimal. What's more, there is more semantic information in a C program than in VM byte-codes, so the C compiler has a better chance at optimization's than the JIT in the VM.

[2] If you claim C isn't ubiquitous, I won't argue, but if C isn't ubiquitous, no language is.

10 Simpler Language Integration

If all your programs compile to C, it can be easier to integrate different languages because, ultimately, they all use the same calling convention. You might have to twiddle with indirections or support libraries, but all in all, integrating multiple languages that are compiled to C should be simpler than learning & trying to fit together the different language-integration techniques for all those languages that compile to native code in their own way.

11 Disadvantages

11.1 Optimizations Performable at Run-Time Only?

I have heard an anecdote that some optimization's can only be performed at run-time & that such optimization's are better than those which can be performed at compile-time. The story I heard claims that Hewlett-Packard has a run-time optimizer for HPPA that improves performance. In other words, HP has an HPPA native-to-native JIT.

If this is true, then there are some optimization's that can be performed by a JIT but not by a pre-run-time compiler (such as a C compiler).

I have not been able to confirm or refute that anecdote. (I haven't tried very hard, either.)

I find this claim difficult to believe. What optimizations could be performed at run-time rather than compile-time? In a language that uses late binding, maybe a function could be compiled, at run-time, to each actual data type on which it is invoked; the compiled functions could be saved in a dispatch table keyed on actual data types. That would cost a lot of memory, & languages with late binding often (usually?) provide for optimization's which can be specified at compile time. Lisp's `declare` special form is an example.

11.2 Compilation Time

Shut-up & get a sense of perspective.

12 Conclusion

I believe virtual machines, with their run-time interpretation or even with Just In Time compilation (JIT), are the wrong way to achieve portability. They have performance penalties, size penalties, reliability issues, portability issues, & they effectively are just re-implementations of perfectly good functionality that is already in the already-existing C compilers.

Creating a new intermediate language, such as Microsoft's Intermediate Language for Dot-Net, is just as bad. C is public, standardized, documented, understood, widely known, widely ported, & already exists.

Compiling to C as an intermediate language is a better way of achieving application portability than are virtual machines or a new intermediate language. Compiling to C is a good way to implement almost any higher-level language.

A UNCOL

Since originally writing this essay, I have learned a little about UNCOL.

1. UNCOL is the Universal Computer Oriented Language.
2. UNCOL is a mythical universal intermediate language, sought since the mid 1950s. ([1], page 511)
3. Alan Kay believes C is or can be an UNCOL. ([2])

B Debugging

Many people have asked whether this technique would work with interactive debuggers. The question didn't occur to me when I wrote the essay because I don't use debuggers.[3]

There's some small hope that a debugger could use `#define __FILE__` and `#define __LINE__` statements in the C code to redirect it to the original source files.

If that didn't work, I don't know of a way to get run-time debugging unless someone modified the debugger.

The compiler could always inject C code to help debugging without a run-time debugger. It could insert memory- & pointer-validation code, execution traces for a log file, & readable "core dumps" from the virtual machine when things went really bad.

I wonder if, in a really desperate move to get run-time debugging to work, the C code could have comments for each line of L source code so that, if you ran the run-time debugger on the executable, it'd show you the problem place in the C code, & the comments would give enough info about the corresponding location in the L source code.

C Change Log

2005 Apr 13 Added the (Section A) appendix. Improved grammar in a few places. Added the table of contents.

2006 Jan 09 Added (Section B) appendix.

[3] I use test-driven design & print statements instead of a debugger.

D Other File Formats

- This document is available in multi-file HTML format at http://cybertiggyr.com/gene/ick/.
- This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/ick/ick.pdf.

References

[1] Alfred V. Aho Ravi Sethi Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, March 1986. ISBN 0-201-10088-6; often called "the dragon book".

[2] Stuart Feldman. A conversation with alan kay. *Queue*, 2(9):20–30, 2005. http://doi.acm.org/10.1145/1039511.1039523.

[3] Gene Michael Stover. C as a portable intermediate language. *cybertiggyr.com/gene*, July 2002. http://CyberTiggyr.COM/gene/ick/.