How to Learn a New Programming Language

by **Joe Marasco**
Senior Vice President
Rational Software

Every few years a new programming language arrives on the scene, promising to be the answer to a maiden's prayer and superior to all languages that have gone before it. But how can we discover the true value of a new language without sinking too much valuable time into learning it? This article talks about a cost-effective way to quickly learn the new language in order to gauge its usefulness.

The Problem

The first reaction to a new language, for those of us who have been around the barn a few times, is "What, again?" We're jaded enough to believe that all these languages are more alike than they are different, and we are quick to dispense with the hype surrounding each new introduction.

On the other hand, hope springs eternal; maybe this time there is more gold than dross. Maybe someone has invented a better "do" loop. Whatever. The point is that simply dismissing the new candidate out of hand is not an option for those of us seeking competitive strength wherever we can find it. So we grit our teeth and once more leap into the breach.

Reading books on new programming languages is rarely an uplifting experience. Frankly, most of them are awful. Every now and then there is a gem that not only kick-starts a new generation of programmers but also stands the test of time as well; for example, "*K&R*"[1] has been the classic manual on C programming since the language first became popular. It introduces C and supplies everything you need in a reference manual, all in a short, readable, and (once) inexpensive paperback. (Today that slim

paperback costs $40.) But as I said, this kind of brief, well-organized, and highly informative book is a rarity.

So, after mucking around in a few of the early books available on a new language, most professionals start to learn a new programming language by writing their first program. This enables more learning than simply reading the somewhat contrived and artificial examples that most books set forth to introduce you to the language.

The Problem with the Problem

Once you decide to write a program, you must then get calibrated. Ever since "*K&R*," it has been almost a cliché to write your first program to do nothing more than print or display the message "Hello, world." While this does teach you how to print out a string and use the compiler and linker/loader, it doesn't do much else. You don't get enough return on your investment to consider this even a first effort.

I believe that the first program shouldn't be so trivial. Nothing ventured, nothing gained. On the other hand, we don't want to make this into a huge effort. We want a project that is just hard enough to cause us to learn how to implement things in the idiom of the new language, yet we don't want to get confused by having to do new domain learning.

What this means is that we would like to have a "standard problem," so to speak, so that each time we have to re-implement its solution in a new programming language under evaluation, we have a "calibration." In other words, we don't want to have to invent new science or devise new algorithms. Theoretically, the exercise should get easier each time we do a new implementation in a new language, because the "problem space" is already familiar to us, and we can spend more of our time judging how well the new language articulates the solution. If the new language takes us four times as long as usual to solve the "standard problem," then we might begin to ask questions about the new language and/or its learning curve characteristics.[2]

What Should the Standard Problem Contain?

Glad you asked. Here are a few of the things I like to discover in any new language:

1. How to print out a string. This is useful in terms of prompting the user for input, for example. As mentioned above, this is as far as you get with the "Hello, world" problem.
2. How to accept input from the user. You can start with simple strings, working your way up to formatted numbers. You can do an awful lot by just reading character strings, so that's a good place to start.
3. Simple algorithmic stuff. You should manipulate some data. Nothing fancy here, maybe just some simple assignment, arithmetic operations, and so on. This will expose whether you need to call in math libraries or not, and so on. It's not necessary to test your

ship's rigging under storm conditions at sea, but you do have to get the boat out of the slip.

4. How to store data persistently. This is a big step up, because it asks that you write out a result and store it somewhere that survives the execution of the program. Ideally, you would like your sample problem to both read and write to the permanent store. Generally speaking, this exposes the language's interface to some sort of file system. Note that the file need be nothing more elaborate than a text file.
5. How to implement a "typical" data structure like a linked list. Since these beasts come up over and over again in programming chores, it is good to have one in your sample problem so you can see how this trick works in the new language.
6. How to do simple error handling. What do we do when user input or a data file is not what we expect -- when it's blank, corrupt, or just plain silly?
7. How to evaluate the abstraction and encapsulation capabilities. Will it be easy or hard to react to a change in requirements or problem definition?

Note that number five is really just an extension of number three.

I offer one big caveat here. What we are exploring are "programming in the small" features of the language. While these are important, they do not test the other very important "programming in the large" features -- things like the ability to have public and private interfaces, interactions with other programmatic infrastructures, and, of course, graphics. Nonetheless, those things can be more easily investigated once the programming in the small issues are better understood.

The Animal Game

Here is the program I have been re-coding as my own standard problem since the 1960s.[3] It is called "The Animal Game."

The program is an interactive dialog between a user and the program. The user is prompted to "Think of an animal." The program then begins by asking, "Is your animal a beagle?"[4] If the user was thinking of a beagle, he answers, "Yes"; the program congratulates itself on its perspicacity, thanks the user for playing, and the game is over.

If, on the other hand, the user is not thinking of a beagle, he answers, "No." Downcast, the program responds, "Sorry, I did not guess your animal. Tell me your animal and give me a question that has the answer 'Yes' for your animal, and 'No' for a beagle."

For example, if the person is thinking of a trout, he would enter "trout," followed by the question, "Is it a fish?" The answer is yes for a trout, and no for a beagle.

Having stumped the program and entered his animal and his question, the user is thanked, and the program again terminates.

However, the next time one plays, something different happens. After being prompted to "Think of an animal," the first question asked is, "Is it a fish?" If the user answers "Yes," then the program asks, "Is it a trout?" But if the user answers "No" to "Is it a fish?" then the program asks, "Is it a beagle?" If the user was thinking either of a beagle or a trout, then the program wins by guessing right. On the other hand, if the person was thinking of some other animal, then once again the program admits defeat, and asks for the new animal and a question that will distinguish the new animal from either a beagle or a trout, as appropriate.

So, in the beginning, the program is exceedingly "dumb." But by storing up the new animals and the new questions, it gets "smarter" as it plays. It won't always guess your animal by the shortest possible route, but after a while it can fake intelligence and "guess" your animal almost every time. That's because as its database gets bigger, it appears to "track down your animal" more and more surely.[5]

Does the Animal Game Fit the Criteria?

You betcha. Here they are again:

1. **Output strings**.The program needs to give the user instructions, and to respond to his responses to your questions.
2. **Input**.The program needs to take strings from the user and do something with them. The dialog nature of the problem requires some (but not much) parsing of the input.
3. **Simple algorithm**.The program is going to branch to different questions depending on the answers. Thus, depending on the "Yes" and "No" responses, it will traverse a data structure.
4. **Interaction with a permanent store**.The program needs to keep the current animal and question database file somewhere, and read it in on program invocation. This data will be used to exhaust the question set. If the program does not guess the animal, it will have to update the file with the new animal and the new question. Then it will have to store these for the next user session.
5. **Prototypical data structure.** Once again, the program will traverse some sort of linked list to achieve the result. When the user adds a new animal and a new question, the program will have to add those links and update some of the old links to point to the new information.
6. **Error handling**.The program needs to be able to deal with blank input from the user when actual values are required, and it has to cope with the data file being corrupted. This area has lots of latitude, depending on how user-friendly you want the program to be. For example, what do you do if the user quits halfway through?
7. **Abstraction and encapsulation**.You can easily generalize the problem to be a vegetable game, a mineral game, or a famous

person game. These variations should depend only on loading a different data file; the language should let you do all of them with the same program.

Note that we have kept this very simple. For example, we don't ask you to allow several people to play the game simultaneously; this would vastly complicate the problem. But even for serial interactions by different users, it makes for a nice programming problem.

Does It Pass the "So What?" Test?

Over the years, I have implemented "The Animal Game" in the following languages:[6]

- FORTRAN
- BASIC
- APL
- Pascal
- FORTH[7]
- C
- Ada
- C++

I would have done Java, but[8]...

In general, it takes me a few hours to recall how to make the linked list work and get something on the air. To allow someone else to play the game unsupervised (which means doing some error handling) usually takes me a few days of programming. I could probably go faster if I could ever find legacy code for any of the previous implementations, but I never can. The exercise gets repeated with a periodicity of every four to five years, which is just long enough to lose the last example.

But, you say, why should you need legacy code? Don't you have a design spec or a pseudo-code document lying around that would enable you to do the implementation without referring to the last piece of code you wrote?

The answer is twofold: Yes, I do develop the pseudo-code, but I get to do it over again each time because I can never find that, either. Second, seeing how you did something in the last language is a useful crutch when trying to do the same thing in the new language. If I had my druthers, I'd be able to locate both my last pseudo-code version and my last language implementation. Together, they would give me a quicker start. If only I kept better records. Oh well. Although this would be a grievous infraction for a software development organization, it is somewhat less of a sin for what is, after all, a small, personal, programming effort.

By far the most bizarre implementations I've done were in APL and

FORTH. Practitioners of those languages will probably understand why, and it is futile to try to explain this to non-practitioners.

Absent from the list of likely suspects are LISP, Smalltalk, PL/I, and COBOL. There is nothing to prevent you from trying these; I just never have.

Incidentally, one of the other things you learn each time you do this exercise is the nature of the development environment and the available tools. I learned, for example, that the early C++ debuggers were not that wonderful; on the other hand, the utility of the Rational Environment was apparent, and my Ada implementation was done in record time. When I did it in FORTH, I learned how to reboot my machine every time there was a runtime error.

It's Your Game

The best way to get one's feet wet with a new programming language is to program a "standard problem" that you are already familiar with. This allows you to explore the language and learn "How do I do this?" on a known set of useful questions. My standard problem is one that I have used for many years, one that I believe forces you to confront a minimal but useful set of basic issues.

If you have developed your own standard problem(s), we would love to hear from you.

Notes

1 Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd edition. Prentice Hall, 1988.
2 There is, of course, the possibility that we are losing prowess with the passage of time. Only the true pessimists among us would admit this possibility.
3 I must admit to being proud of having delivered production-quality software in five different decades, starting with the sixties. In the early days, I wrote much of it myself; later on, my colleagues did everything they could to prevent me from writing code. Over that span of time, I have had to deal with many different languages.
4 An homage to Snoopy and his creator, Charles Schultz.
5 There is a flaw here. A user can insert a bad question or otherwise get things bollixed up -- like reversing the roles of "yes" and "no." Don't laugh; I have seen it done. I have even seen a player forget the animal he was supposed to be thinking of in mid-game. Once this happens, the database is contaminated for future use. Historically, we have had more success with third graders than with adults; apparently, the simplicity of the game is perfect for eight-year-olds and daunting for their parents.
6 The list reveals my programming roots: I come from the scientific side of the house, not the business side. Although I did have to manage a group of COBOL programmers once, I was so busy at the time that learning their lingo was way down on the priority list. And, to tell the absolute truth, I still suffered from "language snobbery" at that point in my career.
7 FORTH is the exception to the rule that programming languages are all caps only if they are acronyms. Charles Moore, the inventor of FORTH, wanted it to be a fourth generation language, but the computer he was working on only allowed five letter identifiers, or so goes the folklore. The individual letters of FORTH do not stand for anything themselves.
8 I really am getting too old for this. And to be perfectly evenhanded about it, I will decline the opportunity in C# as well.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!