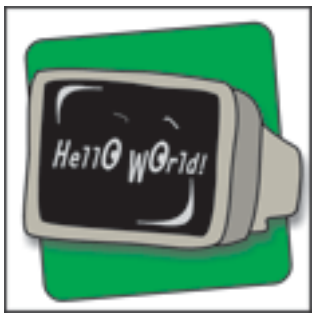
Crossroads
The ACM Student Magazine



What is a Good First Programming Language?

by ***Diwaker Gupta***

Introduction

Programming is an art. As with any other art, it is important to use the right medium. In programming, this translates to the choice of programming language. But why should one pay so much attention to one's first programming language? When there exists a plethora of programming languages and visual editors to make programming so easy, why does it matter which programming language you start with? There are so many books, on-line tutorials, and code samples out there that you could practically learn any language in one day.

Your first programming language will have a profound impact. It affects your programming style, coding technique, and code quality in many subtle ways. In fact, this issue is so important that there is a surprisingly large amount of literature available on the topic [**8**, **10**].

Why do novice programmers find it so difficult to learn programming languages? One might expect the healthy history of programming languages to include languages easy for beginners to learn. Certainly, efforts have been made. Consider **LOGO**, **GRAIL** and BASIC. As it happens, the development of programming languages is not driven by the pedagogical requirements of novice programmers! The designers of these languages

are domain experts in the field of programming, and thus, are not able to fully appreciate the problems faced by a novice programmer. This article aims to discuss some of the requirements of a good introductory programming language, and the shortcomings of contemporary languages that make them unsuitable for beginners.

Things to Consider

Simplicity

A beginner can be easily overwhelmed by code that is unintuitive or hard to read. Consequently, languages with unusual syntax and a counter-intuitive control flow are difficult to learn. Prime examples include functional languages like ML and Scheme. Most procedural programming languages follow natural semantics of control flow and hence are easy to understand. Consider the famous "Hello, World!" program in Prolog [**6**]:

```
hello :- printstring("HELLO WORLD!!!!").
printstring[].
printstring[H|T] :- put(H), printstring(T).
```

and the same in C:

```
int main()
{
   printf("Hello, World!\n");
}
```

To someone who is new to both C and Prolog, the C code will immediately look more understandable.

Orthogonality

By **orthogonality**, we mean that the language should offer a feature set which exhibits linear independence in some sense. In other words, there should not be too many ways of doing the same thing, such as list traversal or array access. While this may seem to be severely restrictive on a language's expressibility, a beginner can be intimidated by the sheer power of the language. A case in point is Perl; from the

beginning, the Perl creed has been "there is more than one way!" The following lines, for instance, are all equivalent [**11**]:

```
if ($x == 0) {$y = 10;}  else {$y = 20;}
$y = $x==0 ? 10 : 20;
$y = 20;  $y = 10 if $x==0;
unless ($x == 0) {$y=20} else {$y=10}
if ($x)  {$y=20} else {$y=10}
$y = (10,20)[$x != 0];
```

As another example, C lets you refer to the `i`th element of an array in several ways: `a[i]`, `*(a + i)`, `*++a`. Such a feature is called a *syntactic synonym*. A related concept is a *syntactic homonym*, in which two or more different constructs are syntactically same, but semantically different (contextual dependence). LISP and **Turing** are examples of languages with these constructs. Although being able to do things in various ways gives the programmer a lot of flexibility, one needs to be experienced in the "art" of programming to be able to use this flexibility in a productive fashion. Novice programmers often end up with unreadable, inefficient, and even incorrect code. An orthogonal feature set keeps things simple, and helps one to focus on the essential concepts.

Target Audience

This is an aspect which is often not given much importance while considering a first programming language. Here is a simple example: suppose you want to teach sixth graders about problem solving using computers: Which language would you suggest? C, **qbasic**, or **JAVA**? Bear in mind that qbasic is fast to pickup, extremely easy to use, and comes with a great development environment with plenty of reference documentation. The programs don't have to be compiled into object code. They can just "run", and there are no pointers! If the focus is on problem solving rather than implementation aspects in programming languages or data structures, then qbasic may be more appropriate than C or JAVA. On the other hand, for incoming college freshmen, perhaps C or JAVA would be a better choice, because the students would be able to better appreciate constructs like pointers and concepts like object orientation.

Coverage

The language should cover all common semantic and syntactic constructs. In particular, it should have constructs for conditionals (if..then..else, switch..case), loops (while, for), aggregates (structures, classes, objects), and some form of exception handling. Specialized domain specific languages like Prolog (logic) and ML (functional programming) fail terribly in this regard. Older languages such as Pascal lack support for many newer concepts like polymorphism and function overloading. However, most contemporary procedural languages fare well in this area.

Regularity

Regularity means that the language should maintain consistency, both syntactically and semantically. This may include invocations with unexpected side-effects or violation of expected semantics. A notorious example from C is:

```
if(x = 1) {
   // do something;
}
```

The above condition will always evaluate to true, irrespective of the value of `x`! C has the uncanny ability to typecast almost any data type into an integer, without notifying the programmer in most cases, and so the above error goes unnoticed. Strongly typed languages like JAVA enforce that only a Boolean value is provided, thus avoiding this problem. Equally notorious is the default prototype for undeclared functions, which assumes the data type of parameters and return value to be integer, unless specified otherwise. This has been corrected in ANSI C.

Another aspect of consistency comes with regard to data types. Taking another example from C, on some platforms, a C `int` will take 4 bytes, while on some it will take two bytes. On the other hand, JAVA guarantees that data types behave *exactly* identically on all platforms.

Turnaround Time and Debugging Support

The most common pattern observed among beginners is an initial coding phase followed by short sequences of code-compile cycles. Novice programmers make a lot of small errors, and they debug programs by making small changes at a time, recompiling the code, and observing the execution. Hence, the turnaround time for a language

environment becomes important. A short debug-compile-execute cycle is preferable to a program that requires multi-pass preprocessing or complex debug-compile cycles. Another extremely important factor is the debugging support provided by the language environment. Although this is not really a language dependent feature, the compiler support is usually tightly coupled with programming language design. As an example, the error messages returned by the JAVA compiler are usually more meaningful than the error messages returned by the gcc compiler for similar errors in C code.

The Sins

I refer the reader to the *Seven Deadly Sins of Introductory Programming Language Design* [**2**] for an excellent discussion on design flaws in most common programming languages that make then unsuitable for beginners. This section outlines some common and important shortcomings:

Excess Brevity

While excess brevity looks very clever, and might be very useful in some cases, it is not always helpful to the novice programmer because it can often become less understandable. The beginner is more interested in learning concepts, and not in learning how these concepts can be expressed in the most concise manner. Excess brevity should be avoided both with regard to syntax and semantics. For instance, languages like **Lisp** and **Scheme** have just one primary data structure: the list. While this abstraction is simple to explain, it results in unreadable code with very little structure. Consider arithmetic expressions in Common Lisp, which must be written in prefix notation (e.g., `(* 2 3 4)` is equal to 24). A novice programmer would preferably use the common infix notation. The Scheme language and LISP (and its variants) are also notorious for little code and greater confusion. The provision of a single data structure (the list) and influence from functional programming paradigm leads to code that is highly unreadable.

Excess Verbosity

On the other hand, excess verbosity is also not desired. The beginner might feel lost trying to grasp the syntax and may miss the conceptual learning. Consider the standard "Hello, World!" program in JAVA. Most beginners will not understand why the `main()` function has to be declared static and void in JAVA, but not in C:

```
public class Hello{
   public static void main(String args[]{
      System.out.println("Hello, World!");
   }
}
```

A strongly typed language such as JAVA imposes a lot of restrictions on the programmer and requires the programmer to understand advanced concepts like inheritance and polymorphism. I/O can be tedious in JAVA, frustrating novice programmers as I/O does not directly aid the programmer in solving a problem using an algorithm. The language COBOL, designed to closely mimic the English language for use by managers, is also excessively verbose. Here is our "Hello, World!" program like in COBOL [**6**]:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.     HELLOWORLD.
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
001200
100000 PROCEDURE DIVISION.
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100500     DISPLAY "HELLO, WORLD.".
100600     STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800 EXIT.
```

Complex Grammar

The problems in this category arise from the presence of *syntactic homonyms* and *synonyms* discussed earlier. Another very common cause of confusion is the *silent*

assumption on parameter types and return values that are not explicitly defined. This is known as *elision* [2]. The distinction between various parameter passing mechanisms (e.g., pass by value, pass by reference, pass by name, pass by result, and pass by value-result) is often complex for a beginner. If not properly explained, this confusion can easily lead to incorrect programs that are very difficult to debug. For example, **JAVA** always uses a pass-by-value mechanism, but many beginners are confused by this since **JAVA** uses object references everywhere. The point to note is that in the case of parameters that are object instances, it is their reference which is passed by value.

Variable Data Types

Beginners must navigate numerous (often redundant) data types. In fact, many experienced programmers also have difficulty identifying the specific data type for a particular application domain, and usually end up using the most general data type. This problem is most common with the design and implementation of basic numerical and character string data types. An example is the different floating point types in C: `float` and `double`. The `float` type is notorious because of its single precision overflow behavior, so most people end up using `double` even when double precision is not required. As another example, consider that C/C++ have 32 distinct numerical data types, and the size of these depends on the underlying hardware as well! The standard `int` type, for instance, varies from 16 to 32 bits depending on the machine and runtime implementation. Clearly, when using numerical data types, it is often required that the user be aware of the internal representation used and its limitations. For instance, the following C code never terminates:

```
int main()
{
   for(double i = 0;i != 10;i += .1)
      cout << i << endl;
   return 0;
}
```

The binary representation of 0.1 is non-terminating, so the increment is not in precise steps of 0.1 so the loop condition is never satisfied.

The Balance of Power

Some languages put too much power in the hands of programmers, while others are far too restrictive. This can have many repercussions. For instance, a language that allows references to arbitrary memory addresses through pointers cannot provide automatic garbage collection. Almost anyone who has programmed in C has experienced the "segmentation fault - core dumped" frustration. Most novices have no idea how to examine the dumped core or use a debugger to find out where the problem happened. To them, "segmentation fault" seems a generic blob that subsumes all kinds of error conditions. JAVA stands at the other end of the spectrum. JAVA is a strongly typed language, and its virtual memory environment imposes a restriction on valid memory accesses. Since JAVA completely avoids pointers, it can do automatic garbage collection, which is very convenient for beginners. However, beginners also miss out on understanding pointers. Although pointers are a bit difficult to grasp at first, they are nonetheless a very important and powerful concept, which is critical to the design and implementation of programming languages themselves.

Towards an Easier Language

What would an ideal language for beginners look like? This section makes some suggestions.

Know Your Audience

It is important to know the target audience. As discussed above, the choice of language will be different for sixth graders versus college freshmen. The choice should be a function of the background of the audience, and especially their understanding of some basic concepts like iteration and recursion. For instance, people rooted firmly in lambda calculus might more easily appreciate a functional language such as ML than an object oriented language like **JAVA**.

Readable Syntax and Natural Semantics

Simple, predictable keywords should be used (`if` instead of cond, `head/tail` instead of `car/cdr`, `!=` instead of <>).

Pitch it Right

The language should pose its abstractions at the right level. The abstractions should provide flexibility without compromising ease of use. For instance, Lisp has just one

data structure: the list. Though simple to grasp, it is inconvenient for solving diverse problems.

Small Orthogonal Feature Set

Keeping the feature set small and orthogonal alleviates the difficulties created by homonyms and synonyms, and also makes it easier for the beginner to pick up the constructs fast.

Keep I/O Simple

Keeping the I/O simple allows one to focus on the problem solving techniques and fundamental constructs like loops and conditionals. For example, I/O in JAVA is extremely well designed; it is extensible and powerful. However, one needs to appreciate the difference between character streams and data streams and how the cascading of streams works. In C++, I/O is both simple and powerful. One needs only to get hold of `scanf/printf` and `cin/cout` to get started. And for advanced users, C++ streams provide all the features required.

Development Support

The development support for a particular language goes a long way towards its acceptance in the programming community, especially for beginners. Development support includes language reference, API documentation, debuggers, and IDE's. **JAVA** took API documentation to a totally new level, in the process enforcing good programming practices such as how to comment the source code (most JAVA API documentation is automatically generated from commented source code).

Keep Up

As an old saying goes, "change is inevitable". As with all things, programming languages also change with time. Even the education patterns change, so the background of novice programmers also changes over time. These changes have to be kept in mind, and our choice should be continuously evaluated.

As an example, at the **Indian Institute of Technology, Kanpur** (the author's undergraduate school), the introductory programming language used to be Pascal until the mid-1990s. Then C took over, because of the advantages it had over Pascal and its popularity as the language of choice for commercial applications. Later on, as OOP

gained momentum and became the standard in industry software development, JAVA was adopted as the introductory language. No doubt, the initial few courses were bumpy rides for the students, but gradually the courses were modified to suit the new language and the students' needs.

So What is it Going to Be?

Where does all this discussion leave us? Can we say anything conclusive about which programming languages are or are not suitable for an introductory course? Brilliant and Wiseman [1] present an excellent treatment of this problem. This section is organized by programming paradigm, and briefly discusses some languages for each paradigm.

Procedural

The procedural paradigm for programming still continues to be the most popular choice among teachers for an introductory course. Much legacy software is written in procedural languages. Having learned these languages in an introductory course aids in jobs and internships. The procedural approach is easy to understand. Furthermore, it is straightforward to convert intuitive algorithms into code. In contrast, object oriented programming requires one to look beyond just the algorithm, and actually go into the structure and design of the system and how its components interact with each other.

Pascal

Pascal used to be the most commonly used programming language in introductory courses until recently. One reason behind the popularity of Pascal among students was the development environment. Most Pascal environments came with a lot of support and documentation. As object orientation, function overloading, and polymorphism became more common, universities gradually started shifting to teach other languages such as C/C++ and JAVA.

C

The C programming language has had an interesting history, and has come to be known as the language of "hackers." The use of C in an introductory course has attracted a lot of controversy and there are compelling arguments both for and against it [7].

On one hand, the learning curve for C is not very steep. C compilers are available on most systems, and getting started with a C program usually does not take too long. Also, a lot of commercial software has been written in C, and hence, knowledge of C is a prerequisite for many professional positions. On the other hand, C gives programmers a lot of freedom and power; most new programmers do not need this kind of raw power and do not understand how to use it appropriately. Perhaps the most common complaint against use of the C language is pointers. The fact that pointers are required even for very basic things (such as implementing a linked list) coupled with the fact that pointers are not easy to teach or to understand adds to the level of confusion. C also faces a lot of criticism for poor support for data encapsulation and information hiding.

Object Oriented Programming

Object orientation is not a new concept. Smalltalk, in some sense, was the first object oriented language, and it came out around 1970. However, the mainstream acceptance of OOP as a programming paradigm did not happen until C++ and JAVA became popular. These two languages truly launched OOP into the forefront of software development and led to widespread industry acceptance of the methodology. At first glance, object orientation seems very simple and natural. However, to build a real system using OO concepts is a subtle art. C++, JAVA, and Ada all are vast complex languages, with an extremely rich set of APIs. This size can be daunting to beginners. Often, a subset of the language in question is used for teaching purposes. While this idea works well in practice, its success is hampered by the fact that reference books are not written keeping this in mind. Most reference text will try to cover as much material as possible, and the student faces a hard time trying to filter out what is needed and ignoring all of the other information. [**3**]

C++

C++ derives both its popularity and problems from C, since it is so deeply rooted in it. C++ adds several powerful features to C, such as templates and object orientation, but these are seldom of relevance to the novice programmer. However, C++ does get rid of many of C's problems. Consequently, very often C++ is used in place of C to teach procedural programming.

JAVA

JAVA is relatively new but it has made substantial leaps in recent history. JAVA was designed bottom up to be a safe object oriented language with a very strong focus on "what to do" rather than "how to do." Consequently, JAVA is highly standardized, strongly typed, and has a rich set of APIs that make it easy to write programs. However, JAVA is quite verbose. Further, one needs to really understand object oriented programming to make the most effective use of JAVA. For people used to programming in C/C++, JAVA might seem restrictive since it does not use pointers at all, and it does not provide the "raw" power of C/C++ [**5**].

Functional Programming

The functional paradigm is by far the least popular for introductory courses. This is not really unexpected, because to appreciate functional programming fully, one should already be familiar with at least the procedural programming model and also have the requisite theoretical background (e.g., lambda calculus). Another problem that functional languages face is that they are not "real-world" languages. The most popular functional languages are Scheme, Lisp variants, and ML, none of which are commonly used in industry.

Looking Beyond

Although important, choosing the appropriate language is just the first step in designing an introductory course [**4**]. From a pedagogical point of view, there are a number of other factors to be considered. These factors affect not only the choice of the language, but also the structure and design of an introductory course. In *Teaching Programming To Beginners: Choosing the Language is Just the First Step* [**4**], the authors describe how the curriculum at their institute shifted from a procedural language to an object oriented language. They further describe the redesign and restructuring of their introductory course to enable the students to "explore the art, science, and pleasure of programming." There are many ways to structure a course. A reasonable choice seems to focus on **problem based learning**, allowing students to focus on techniques rather than on the language syntax itself. Object oriented languages have an upper hand in this regard, because they allow modularity and code reuse so that students can use simple pre-built components to make more sophisticated applications. On the flip side, they require some understanding of OOP on the part of the student.

Another issue is that of class size. In general, problem-based learning does not go too

well with large group lecturing. However, with the provision of comprehensive text and "learn-at-your-pace" philosophy, it can be avoided [4]. The authors also discuss structuring of labs, tutorials, and effective assessment techniques.

Conclusion

In light of this discussion, can we settle on a choice of the best first programming language? Clearly, a number of factors need to be taken into consideration when making this decision. The language should be in the happy medium that allows for beginners to grasp the basics, but still be able to eventually use the more advanced concepts. It should also be intuitive enough that beginners will not give up in frustration. Finally, all of these different factors need to be evaluated by considering the target audience and structuring the language and the course to fulfill the needs of the students.

qbasic is a very good choice for sixth graders, for a number of reasons: it has a well integrated environment with extensive online support, the debug-run cycle is extremely short because it involves no intermediate object code compilation, it has a rich API that students can use to solve problems, and it is pitched at just about the right level; not as high level as object oriented, or as low level as pointer manipulations.

However, for college freshmen, C is a better choice, because it introduces older students to the more complex concepts. However, this choice is made with the understanding that the instructor will focus on problem solving rather than exploiting the "raw" power of C. In that regard, pointers need to be introduced right at the start, rather than being avoided or deferred until late in the course. Pointers are imperative to the proper understanding of more advanced programming language concepts, and essential for many efficient data structure implementations. C is simple enough that students can immediately begin to write simple programs while being flexible enough to allow students to gradually use it to create more complex ones. Moreover, it introduces the basic elements common to most widely used "real-world" programming languages, and provides a good foundation for learning other languages.

References

1

Brilliant, S. S. and Wiseman, T. R. "The first programming paradigm and language dilemma". *Proceedings of the 27th SIGCSE Technical Symposium on*

Computer Science Education. Philadelphia, Pennsylvania. 1996. pp. 338-342.

2

Conway, D. and McIver, L. *Seven Deadly Sins of Introductory Programming Language Design*. Department of Computer Science, Monash University. <**http://www.csse.monash.edu.au/~damian/papers/PDF/SevenDeadlySins.pdf**>.

3

Decker, R. and Hirshfield, S. "The Top 10 Reasons Why Object-oriented Programming Can't be Taught in CS1". *Proceedings of the 25th SIGCSE Symposium on Computer Science Education*. Phoenix, Arkansas. 1994. pp. 51-55.

4

Duke, R., Salzman, E., Burmeister, J., Poon, J., and Murray, L. "Teaching Programming to Beginners: Choosing the Language is Just the First Step". In *Proceedings of the Australasian Conference on Computing Education*. Melbourne, Australia. 2000. pp. 79-86.

5

Hadjerrouit, S. "JAVA as First Programming Language: A Critical Evaluation". *ACM SIGCSE Bulletin, Volume 30, Number 2*. 1998. pp. 43-47.

6

Hello, World! Project. <**http://www2.latech.edu/~acm/HelloWorld.shtml**>.

7

Johnson, L. F. "C in the First Course Considered Harmful." *Communications of the ACM, Volume 38, Number 5*. 1995. pp. 99-101.

8

McIver, L. *The Effect of Programming Language on Error Rates of Novice Programmers*. School of Computer Science, Monash University.

9

The Google Directory. "Programming Language," *Comparison and Review* <**http://directory.google.com/Top/Computers/Programming/Languages/Comparison_and_Review/**>.

10

Wexelblat, R. L. "The Consequence of One's First Programming Language." In *Proceedings of the 3rd ACM SIGSMALL Symposium and the First SIGPC Symposium on Small Systems.* Palo Alto, California, 1980. pp. 52-55.

11

World of Perl. *Introduction to Perl.* University of Missouri - Columbia (20 April 1999) <**www.cclabs.missouri.edu/things/instruction/perl**>.

Biography

Diwaker Gupta (**dgupta@cs.ucsd.edu**) is a graduate student in the Computer Science and Engineering department at the University of California, San Diego. He is interested in systems and networking, especially computer networks and distributed systems. In his spare time, he enjoys reading, swimming, and playing cricket.