BACK TO PASCAL:

RETRO BUT NOT BACKWARDS

Abstract

The debate over which language to use when teaching beginning programming has persisted for decades. One institution has elected to give Pascal a second chance. The rationale for this change and some early experiences are outlined. The difficulties experienced when this institution used C++ are described and the requirements for a first language are listed. Pascal satisfies most of the requirements as specified. Opposition to the change was noted and countered. Now, after the first semester of use, some clear advantages as well as disadvantages are evident.

Introduction

"You can't teach beginning programmers top-down design because they don't know which way is up."C.A.R.Hoare

No, this is not another paper about whether or not to teach OO in first year. Our department uses a structured, imperative paradigm in our first course and this was not open for debate at this time. Even though we previously taught C++ in CS101, we did no OO until the second course.

While many CS departments, having already decided to switch to an "OO first" approach, debate the relative merits of introducing applets and GUIs in the first course, ours has quietly moved backwards. At least, that is how many perceive a move back to Pascal as a first language. Describing the decision to adopt Pascal as the programming language used in our CS101 as "going back to Pascal" is indeed accurate. It is however, not a regression – rather it is more of a fundamentalist reaffirmation – "back to basics", as it were.

The first course in a CS program is, of course, key in determining the students' approach towards the remainder of their program. It is destined to leave a lasting impression on its subjects for better or

worse. As instructors, we are well aware of its importance. We all have strongly held beliefs about what's best for our beginning students and we want to ensure that our favorite topic or technique is included. We feel very passionate about this. However, the choice of approach in a course must be governed by something other than passion, so to that end, we must outline clearly defined goals, rationales and desired outcomes that can be measured and where progress can be assessed. There is something to be said for keeping both our "pedagogical heritage and history" in view as we do this. [KAY 96]

1. Background

The ACM/IEEE Computing Curriculum 2001 [CC2001] offers 6 different approaches for the first course. Three involve programming first, and one of those involves the imperative paradigm. It remains one of the most popular approaches. At our institution we have tried to combine a programming-first approach with a broad introduction to problem solving and design, as well as giving students some breadth. As a result, we have rarely covered all aspects of a particular programming language, regardless of which one. In the last 10 or so years we have tried: Miranda, C, C++, and now finally Pascal (again).

The move to Miranda, a functional language, was abandoned after five years. We tried a functional paradigm because it was felt that it would enable instructors to spend less time on syntax and more on problem solving. We ended up spending considerable time with the functional paradigm and the problem solving aspects were not as easily transferred to an imperative or OO approach by the students as we had hoped. Students did fine with the functional paradigm, but that didn't help when it came to learning C/C++. They had to start from scratch again.

For a few years we used 'C' in the first course, but this was quickly abandoned in favour of C++ as it was felt we needed something with more market appeal.

Up until this year we used C++ in both our first courses. The approach we have always used involves covering the introductory sequence in three semesters. In the latest ACM Curriculum, this is now recognized as the CS101-102-103 sequence. After three years of teaching with C++ we felt that students were unnecessarily frustrated by the language itself, and this was not merely the result of inexperience in

using it as a tool. As it turns out, our students weren't the only ones to experience this frustration. [DING00] [KAY 96]

Late in 2000 discussions began to pave the way for another language change. The proposal was to move to Pascal in CS101 and Java in CS102 from the outset, and most were in favour of switching to Java as an OO language. A few did have difficulty with the move to Java. One felt it was impossible to teach OO without access to multiple inheritance. A few others were concerned that students would be left with no experience in C/C++. The latter was addressed by offering to introduce C in the final weeks of CS102 and by spending more time with it in our required 2nd year architecture course. The proposal to switch to Pascal met with very mixed response, including some who were in severe opposition.

2. Problem Specification

When deciding which tools to use in the first course, the first step is to decide what is fundamental. Our institution has always taken a fairly traditional approach, which has usually agreed with the ACM Curriculum. Having decided what to teach, we can then decide how this is best accomplished. The choice of language cannot be made in isolation. A number of other factors must be kept in mind, such as what is to be covered in subsequent courses, whether or not this will make life easier or harder for students later (and whether or not we care). Will students be able to easily transfer their knowledge to a new language? Is this even one of the goals of the first course? The length of the program at your institution also plays a role: a four-year program will have more flexibility in terms of pedagogy than a two year program.

These days it is quite widely accepted [BRUC96] that experience with multiple languages is crucial to success in CS. In some cases, it is a requirement of accreditation, which was the case for us. By choosing the three course introductory sequence suggested by the ACM (CS101-102-103), we are afforded more flexibility to spend time on fundamentals and provide more breadth than we would with a

two course introductory (CS1-CS2) sequence, while still providing a solid introduction to programming. In our case, our eventual goal is that CS101 become a remedial course, to be covered in the high school curriculum. For the immediate future, a primary goal was to free up class time to allow for discussion of things not directly related to the language being taught, such as general problem solving, design issues, even other topics in CS.

In order to provide the impetus for the language shift, it was necessary to identify the current problems in the affected courses using C++. Chief among these problems was that students complete first year with very vague notions about the fundamental concepts (variables, lists, scope & lifetime, OO). They get embroiled in syntax issues that prevent them from concentrating on algorithmic issues [SOLO86]. At the same time, they can get away with undisciplined code because the language permits almost anything. In large classes such as ours, those who mark the assignments do not often have the luxury of a detailed analysis of student submissions, which could correct this difficulty. Different compilers produce different results, which is a serious problem for beginners who do not yet have the means to comprehend, and therefore compensate for, these differences.

Our students get a slow start to programming: usually week #5 (out of 13). This is due to an effort to cover low-level details well enough to give students the foundation required to cope with the programming bugs they are likely to generate.

When first learning, students are dumb-founded by unhelpful compiler messages and programs that compile but still don't run right (due to uninitialized variables, array bounds errors, '=' vs '==', side-effects in 'while', 'for', and 'if'). Most beginning students firmly believe that once their program compiles, they are almost done. Having a program that runs but does something bizarre (while it may be a great pastime for some of us) can be very disheartening for a beginner. [CLOS00]

Since our CS101-102 sequence use the same programming language and the second course is essentially a continuation of the first, IB/AP and other advanced students wishing to begin their program with the second introductory course must still take the first if they don't know C++. These students are bored, and some of the brighter ones go elsewhere as a result. In a department undergoing expansion with

a desire to improve the overall quality of our students, this is considered a serious problem.

Learning one language for a full year results in students who are fairly fluent but also very reluctant to learn a new language. They feel they know quite a lot about C++. The prospect of learning that much about another language seems quite painful. For many of us, learning that second language was nearly as difficult as learning the first. It was felt that having them learn a new language after the first semester would force students to cross this barrier sooner, and generally improve the students' ability to learn even more languages.

3. Requirements Analysis

In order to become properly positioned to advocate Pascal in the introductory course, it was necessary to clearly define what the requirements of a language used to teach beginners should be. A first programming language must provide a protected programming environment with enforced structured organization of program parts. We wanted compiler, or at least run-time generated warnings for students about common "dangers" (such as: array bounds violations, uninitialized variables, etc.). The language syntax must be simple and unambiguous, and it must have the capabilities for the following: subprograms, pointers, structures, file I/O, some form of conditional switch, ability to define new types, strong typing, and recursion. Finally, the language must enforce some of the style issues we are trying to teach.

4. How Pascal Fits the Bill

One of the key factors influencing the final acceptance of the Pascal had to be the realization that the audience is assumed to have no experience with programming. The over-riding concern was to be able to move beginning programmers to the intermediate stage as smoothly as possible. The argument was made that Pascal does fulfill the requirements outlined above. It is known to be easier to learn for beginners [DING00]. It also possesses a number of additional advantages. Perhaps chief among them is

the fact that the **entire** syntax can be presented formally on 2 sheets of paper. With Pascal we have an opportunity to introduce the concept of formal languages in first year - something we couldn't do with C++. No parts of the language are "off limits" to beginners so there is less to complicate the concepts. Standard Pascal compiles the same way **everywhere**. Pascal warns about uninitialized variables, won't permit array bound violations, implicit type conversions, and distinguishes between real and integer division. While an experienced programmer is likely to find these "features" restrictive, all of them are useful for beginners. Pascal has clearly defined parameter passing mechanisms (no defaults). It permits no statement side-effects which is typically a difficult concept for beginners. Since the language is so restrictive, it will allow us to concentrate more on algorithms & problem solving and less time on syntax, thus students can begin creating substantial programs sooner. Using Pascal, we can still cover all the major concepts including pointers and recursion (which we didn't have time for with C++). Pascal compilers typically have more meaningful error messages and the restrictive nature of the language helps to foster more disciplined programming [DING00].

When using C++ it is necessary to discuss binary representation in depth so students are armed with the background necessary to comprehend some of the weird bugs they will encounter. There is less need to discuss this with Pascal since we can't easily do bit bashing (not even by accident). With Pascal first, we can postpone the in-depth treatment till CS102 (fits in better with Java), or possibly even till the second year architecture course. As a result we will spend less time on it in CS101 - allowing us to start writing programs sooner.

A potential side-effect of the change is that it will encourage non-majors to take the first year service course instead (which could change to C++, or whichever language currently seems to be the most desirable). The advantage is that we will end up with a higher % of 'our guys' in CS101, as currently about 50% of the students enrolled in CS101 are non-majors. Our institution does not yet support direct entry and as a result we are not permitted to restrict enrollment into our first year courses. Non-majors are typically not as dedicated to the discipline and less willing to spend the necessary time to learn the material. As a result, instructors and lab teachers spend time helping them (both in and out of class) that

could be better spent with our majors.

5. Opposition

Not surprisingly, reaction to the proposed change ranged from curiosity over how it might work, to amusement at the prospect of turning back to a language other institutions are abandoning in droves, to laughter. A review was made, as far as possible of the first language used in post-secondary institutions [REID00]. Pascal still retained a substantial following, including a few of the "big" names. This helped to lend our proposal some validity. There was, however, plenty of other opposition. The major points are outlined and addressed in this section.

Pascal is viewed as old, not commercially marketable. Its purpose in first year is that of **teaching tool** (like writing in pseudo-code but you get to test and run it). Since we are not in the business of producing marketable workers after the first year, whether or not they learn a "useful" language as their first is irrelevant.

Pascal syntax is different from C/C++/Java/Perl... Students will be forced to deal with many forms of notation anyways in Computer Science - this is just part of the discipline. Industry is moving towards multiple languages for different purposes now and learning more than one language will help them prepare.

Pascal is seen as a step backwards. Pascal is just the ***first*** of a number of tools students will learn. By the end of the first year they will have been exposed to: Pascal, Java, C, various UNIX utilities, BNF, syntax diagrams, and UML. Further, we are moving towards making CS101 a remedial course. Eventually, it will be assumed that students entering Computer Science will all have programming experience. By treating Java in CS102 as a 'new' language and syntax for all in the second course, we become more flexible. IB and AP students as well as those with other prior experience can begin with CS102 easily (not currently the case unless they already know C++). This will help to attract more good students, and simplify transferability. CS101 challenge exams can be language independent.

Many currently popular languages have a C-like syntax rather than a

Pascal-like syntax. Since students must become comfortable dealing with many forms of notation this is not really an issue. Until very recently, most of our students graduated essentially knowing only one language. They became so attached to the one language that learning any other was avoided. This is becoming a serious problem in industry where people are expected to be flexible. In terms of syntax, the biggest potential confusion is easy to avoid: i.e. the use of '{' '}' in Pascal as comment delimiters. We can easily use just the original delimiters: '(*' '*)'. We can also use the 'C'-style for arrays.

Students won't learn C++ in first year. Students will however learn Java and C in CS102; with experience in these two languages, picking up C++ won't be difficult. We have since added a new course (elective) in second year (as a follow-up to CS103) that will use C++, among other things. This course could also be used as a pre-requisite for the graphics courses, which seem to have the greatest need for C++.

Too many languages are confusing. This complaint warrants a more detailed response.

6. Too Many Languages are Confusing

Throughout most of the late 1980's and well into the 1990's the emphasis on programming skills focused heavily on a single language: C++. Previous to that there had been a plethora of both special and general-purpose languages with new ones being designed almost monthly. Many had excellent design and wonderful features but most died out, largely due to resource issues. The language was wonderful but we couldn't do anything useful in it on the machines we had available at the time. As a result, C flourished, and when OO became all the rage, many shops switched to C++ while still essentially programming in C. Over the last ten years the advances in hardware have once again made interpreted languages a feasible choice. Now a programming project often begins with a consideration of which language would be the best choice for the task. As stated before, industry is moving in this direction already, so with the change of languages in first year, our program will more closely match current industry requirements than a unilingual program does.

Accreditation actually **requires** that students know multiple languages. Before the proposed

language change we had difficulty meeting this requirement.

As an added benefit, learning multiple languages allows students to see common concepts and capabilities, so programming becomes a higher-level process and they are less bound by specifics of syntax [APPL01]. When most students know only C++, everything they do is organized in terms of C++. They have great difficulty seeing another way to approach a problem. In my third year class, some students would rather write a 1500-line C++ program than learn enough Perl to do it in 20. Experience with multiple languages allows students to see first hand the value of using the appropriate tool for a task.

As any person fluent in multiple natural languages can attest, the more languages they learn, the easier it becomes to learn more. When students are exposed to numerous languages in junior courses, we won't have to spend so much lecture time on syntax in senior courses. It will also give us greater flexibility in senior courses, allowing us to let students use whichever language or tool is appropriate to the task at hand.

7. **Experiences**

We have recently completed the first run of the Pascal course and are now mid-way through the next semester. The basic framework of the course follows that outlined in CC2001 as CS101. Overall, the students experienced noticeably less frustration and reported fewer problems learning the introductory concepts using Pascal than they did learning with C++. Since our classes tend to be large and our demands high in the first year, we have set up a "Tutor Room" specifically for our CS101-102 students. It provides a place where students can go for one-on-one help with any concept or topic covered in class as well as with their assignments. This facility is normally available 25 hours per week and while in previous semesters the tutors had been requesting both additional hours and tutors in order to handle the demand, this past semester, they found they often saw no students at all. Students reported noticeably fewer frustrations when it came to dealing with compile-time errors. Questions dealing with syntax issues were almost non-existent.

What we did notice is that we were not able to get through the fundamental concepts any faster. It took students just as long to get comfortable with the concept of arrays, for example, as it did when we used C++. It would appear that there really is no short cut to teaching the fundamental concepts. They still need to be learned and they still require time to sink in [GHAF01]. Difficulties with syntax did not seem to affect this.

As instructors, we were able to spend considerably less time during lectures talking about syntax. In Pascal, we could simply present the class with the BNF for a particular construct, show them the syntax diagrams, or both, and then go on to use it in various examples. Using Pascal as the vehicle for teaching introductory concepts has indeed allowed us to place more emphasis on problem solving.

In terms of resources, the compiler used (GNU Pascal) was felt to be unacceptable as it had implemented too many C-like features, and some of the protection we had sought from the restrictive nature of the language was lost by an overly extended compiler. We are still looking for an adequate compiler that we can use on our Solaris systems.

The students were told about the language change at the start of the semester and given the rationale. Very few complained about the choice of language, and seemed willing to accept the choice of Pascal for pedagogical reasons.

The transition from Pascal to Java has exposed some unpredicted difficulties that need to be addressed. The first assignment in CS102 was merely a code translation exercise. It seemed to accomplish our goals but the students clearly needed more practice with the new syntax than they had received before going on to deal with the new concepts of the new paradigm [TUTT01]. Shifting from solutions enclosed entirely in a single file to separate classes, each in its own file is a major conceptual leap. This seems to be connected to the general difficulty associated with the shift in paradigms rather than the specific languages employed, as this was also the case in previous semesters. When C++ was used in both courses, students still had difficulty when it came to program organization in terms of seperate components (i.e. objects). Although further work is required to verify this, it would appear that the use of Java instead of C++ seems to accentuate this shortcoming without actually solving it.

Conclusions & Future Improvements

We may need to admit we won't be able to accomplish all we claimed we could when the change to Pascal was first proposed. It would seem that, while it does offer some sound pedagogical advantages, a move back to Pascal will not suddenly simplify the task of introducing new students to Computer Science. Some of the benefits won't show themselves for a few more years, when these students start to deal with advanced topics at the senior levels. Some of the benefits are difficult to measure: how does one demonstrate the acquisition of a "gestalt" of programming languages?

Having discovered that although Pascal does indeed simplify the mechanics of program writing it still doesn't simplify the process of learning to program, the next step will be to develop lectures and lab exercises that capitalize on our newfound freedom to spend more time on design. The reintroduction of a set of syntax exercises developed at this institution many years ago is in progress [BECK83].

Better quality resources need to be developed to help students through the transition and guide them into the OO paradigm. A series of graduated exercises to help students learn the new syntax is being designed as well as one to take students from programs written as a single unit to those designed as components.

While the outcome of the change back to Pascal was not as remarkable as had first been hoped, the use of Pascal has indeed addressed and resolved enough of the problems initially outlined to be deemed success at this institution.

References

[APPL01] Applin, Anne Gates, **"Second language acquisition and CS1"** Technical Symposium on Computer Science Education - Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education 2001, Charlotte, North Carolina, United States Pages: 174 - 178

[BECK83] Becker, Katrin, **"Teaching Syntax in an Introductory Programming Course"** CIPS Conference '83 Converging Technologies May 16-20 1983, Ottawa Canada, Proceedings, pp183-195

[BRUC96] Bruce, Kim **"Thoughts on Computer Science Education"** ACM Computing Surveys Vol. 28A No. 4 (December 1996)

[CC2001] ACM/IEEE **"Computing Curricula 2001" Final Report Dec. 15 2001** The Joint Task Force on computing Curricula, IEEE Computer Society, Association for Computing Machinery.

[CLOS00] Close, Richard, Danny Kopec, Jim Aman **"CS1: perspectives on programming languages and the breadth-first approach"** Consortium for Computing in Small Colleges - Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges 2000, Ramapo College of New Jersey, Mahwah, New Jersey, United States Pages: 228 – 234

[DING00] Dingle, Adair, Carol Zander **"Assessing the ripple effect of CS1 language choice"** Consortium for Computing in Small Colleges - Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference 2000 , Oregon Graduate Institute, Beaverton, Oregon, United States Pages: 85 – 93

[GHAF00] Ghafarian, Ahmad "**Teaching design effectively in the introductory programming courses"** Proceedings of the fourteenth annual consortium on Small Colleges Southeastern conference 2000, Roanoke College, Salem, Virginia, United States Pages: 201 – 208

[KAY 96] Kay, David G, "**Bandwagons Considered Harmful or The Past as prologue in Curriculum Change"** SIGSCE Bulletin, Volume 28, Number 4, December 1996 Pages: 55-58,64

[REID00] "First-Course Language for Computer Science Majors – The List" http://www.case.wvu.edu/~vanscoy/reid.htm

[SOLO86] Soloway, Elliot, **"Learning to Program = Learning to Construct Mechanisms and Explanations"**, Communications of the ACM, Vol.29 #9, Sept. 1986 p850-858

[TUTT01] Tuttle, Sharon, **"¡Yo Quiero Java! Teaching Java as A Second Programming Language"**, Consortium for Computing in Small Colleges - Proceedings of the Third Annual CCSC on Computing in Small Colleges Northwestern Conference 2001, Pacific Lutheran University, Tacoma, Washington, United States Pages: 33 –44

Filename:	Pascal-submission.rtf
Directory:	C:\X-PC\Publications\X-Old-Papers\Pascal
Template:	C:\Users\becker\AppData\Roaming\Microsoft\Templates\Normal.dot
Title:	Teaching Advanced Data Structures to Computer Science Students
Subject:	
Author:	becker
Keywords:	
Comments:	
Creation Date:	3/11/2002 8:43:00 AM
Change Number:	4
Last Saved On:	3/11/2002 8:52:00 AM
Last Saved By:	becker
Total Editing Time:	1 Minute
Last Printed On:	4/5/2009 4:43:00 PM

As of Last Complete Printing

Number of Pages:	12
Number of Words:	3,924 (approx.)
Number of Characters:	22,367 (approx.)