Role of the C language in Current Computing Curricula
Part 2 – Beyond Survey Results

Alessio Gaspar,
Alessio@lakeland.usf.edu
University of South Florida Lakeland
3433 Winter Lake Rd, Lakeland 33803, Fl, USA

Abdel Ejnioui
aejnioui@lakeland.usf.edu
University of South Florida Lakeland
3433 Winter Lake Rd, Lakeland 33803, Fl, USA

Naomi Boyer
nboyer@lakeland.usf.edu
University of South Florida Lakeland
3433 Winter Lake Rd, Lakeland 33803, Fl, USA

Role of the C language in Current Computing Curricula Part 2 – Beyond Survey Results

Abstract

In December 2006, a survey hosted on surveymonkey.com was publicized through various ACM mailing lists (SIGCSE, SIGITE). Its purpose was to determine the role of the C language in the various modern computing curricula (CS, IT…). This paper summarizes the results and stresses out the quantitative usage of this language in introductory and intermediate programming courses as well as in upper-level undergraduate courses (e.g. operating systems). We also present the qualitative reasons provided by our respondents for, or against, the adoption of the C language in these various contexts. We then discuss these results and propose an analysis of when in the curriculum the C language might be most useful, how it should be introduced and what specific topics should be covered in such a re-designed "intermediate programming in C" course.

1. Introduction

This second part of our paper revisits the results presented in the previous one and then discusses the C language from the perspective of past efforts to improve the pedagogy of programming courses which have to rely on it. We then discuss the preferential niche for this language in modern curricula. The rest of the paper will therefore be organized as follows; Section #2 will discuss the arguments relative to the inherent complexity of the C language from a technical perspective. We will discuss how these difficulties can become advantages when using C to achieve different pedagogical objectives. Section #3 will then discuss the role of C from a temporal perspective, that is, when is this language best suited to be introduced to students? Section #4 will complete this discussion by revisiting the contents of a C course not meant for beginners. Section #5 will offer some advices on how to address the various issues raised in the previous discussions as well as describe the main objectives of the CLUE project the authors are working on.

2. Complexity of C

The figures from Table 1 revealed that only 23.71% of our participants are using C in either introductory or intermediate programming courses. While this number is sufficiently small to indicate that C is not a beginner's language, it is also paradoxically high when compared to the other surveyed languages. The adoption rates of C are higher in both course levels than the rate of other languages such as C#, Perl, PHP, VB and even Python. The results are surprising for the latter since it benefits from obvious pedagogical benefits and a rather dedicated and enthusiastic community.

Since its early adoption, the C language has never been considered as an appropriate learning tool for beginning programmers **[8, 9]**. However, this survey revealed that the reasons for this pedagogical position have evolved over time. Specifically, technical issues with the language itself are no longer the top concerns, but the lack of object oriented features is. How can this explained? The comments gathered by the survey indicate that the object-first approach is the most popular approach among respondents. Some also pointed out that typical data structure courses (CS-2) are increasingly making usage of object libraries such as the Standard Template Library (STL). This clearly

makes the C language, regardless of its intrinsic difficulties, not suitable for introduction before the progressive completion of CS-1 + CS-2 by students.

As previously noted and underscored by others **[10]**, it is believed that many of C features are not intrinsically wrong. Because of the popularity of Java, C++, and C# in academic and industrial settings, very few still consider them harmful to student programming education **[7]**. Concurrent to this situation, the sophistication of Integrated Development Environments (IDEs) is currently providing an unprecedented level of assistance to developers and students with features such as syntax highlighting and structures collapsing **[12].** The wide availability of these IDEs and extensive online documentation for object libraries are all convincing factors that overshadow the specifics of a given language and at the same time improve the learning experience of the students with the language.

But what about the *inherent* difficulties of the C language? When saying that C is "too complex for students", most educators refer to aspects of the language related to pointers arithmetic, explicit memory manipulation and so on. What can be done to address these issues in C without having to change the language itself? In agreement with the findings in **[10],** the authors of this survey strongly believe that most problems are located at the runtime and compile-time levels, and can be addressed by appropriate features of the development environment. In addition to syntax highlighting and debuggers, meaningful compilation-time error messages can significantly help students on these intricate language aspects. In fact, the findings of the survey confirmed that the lack of appropriate help at both runtime and compile-time in tracking bugs efficiently are perceived as a hindrance to the learning process. While this situation is unacceptable when introducing students to programming, it can be a simple difficulty to overcome if C is used later in the curriculum to strengthen programming skills and prepare students for the upper level courses. In this perspective, the objectives are different and more aligned with what C is identified as suitable for. Assuming that the intent is not to shun C, the question remains as to the best time in a computing curriculum to introduce this system language, what to cover in such a course and how to do it efficiently

3. Appropriate Time to Introduce C

The expressed need for C in core courses of most computing programs such as operating system, computer architecture, and networking, and its absence from core programming courses such as CS-1 and CS-2, constitutes a peculiar curricular paradox. This paradox bears the question of how instructors manage to cope with the missing pre-requisite knowledge. Following the announcement of this survey's results on SIGCSE, feedback was solicited from instructors teaching operating systems in departments in which the C language was not taught as a mandatory programming course. Received email responses indicated that a "crash course" in C, offered during the first weeks of the course, was the most common approach to let students discover the language mostly on their own. On the other hand, many CS departments offer a Unix/Linux programming course which aims at introducing students to system development on these platforms. This often includes an introduction to C, shell scripting and various GNU utilities. Such a course is clearly not intended for beginners and represents an efficient way to prepare students, after a CS-1 + CS-2 progression, to more advanced system-oriented topics. It is

important to note that such courses are focused on programming techniques and concepts without any emphasis on software engineering practices.

At the authors' institution, many students come to a gate course, the COP 3515 Program Design, after attending multiple introductions to programming using Java, C++, VB, or other high level languages. The use of C in this course allows students to revisit programming concepts with a more technical spin. With this in mind, the stack is used to explain variables duration and parameter passing while the heap is introduced along with the nuts and bolts of dynamic memory allocation. This approach strengthens already acquired knowledge and, oftentimes, justifies programming precepts that students have been following rather blindly before. The difficulties of C are dissected and used to reinforce the students' programming discipline. Upon successful completion of this course, students accepted into the CS or IT programs are found to have a deep understanding of the programming activity.

4. Appropriate Material for a C course

Regardless of the approach taken, paced or crash course, it is clear that C is best introduced at a faster pace to students who have already acquired basics of programming. The authors of this survey believe that three distinct aspects should be covered in an intermediate C course.

At first, C can be used to revisit programming concepts and strengthen students' skills. It is easy to cover C syntax with experienced students in one or two sessions with a strong focus on the language notorious pitfalls **[13, 14]**. On the data structure side, these sessions can also be used to show to students how far a language like C can go to support modularity and object orientation. The material presented by Stroustrup **[15]** provides a good example of such a demonstration. Once the basics have been covered, programming tools such as debuggers and *makefile* scripts can be introduced along with extensive hands-on practice to help students scrutinize their code through cursory debugging and code-reading skills. Students tend often to look at their code from the perspective of an executive summary; "it does this". However, when the code does not execute as they expect it to, they are unable to analyze it line per line. These first sessions are also a good opportunity to guide students through syntactical, linking and runtime errors. Exposure to typical outputs form such errors is the best way to overcome the lack of clarity familiar in many traditional C development tools.

The main part of such a course can be devoted to have students experiment with explicit memory allocation and pointers. These activities generally lead to a discussion of the executable memory image, which can be used to revisit and justify programming languages concepts from a more technical perspective. For instance, the heap can be used to explain memory allocation and the storage of literals (e.g. meaning of char* p = "hi""). In addition, the stack can be used to explain recursion and parameter passing. Furthermore, pointers can be used to contrast parameter passing by reference and value. Other topics unfamiliar to most students such as libraries, linkers, and loaders can be included also in this coverage.

Lastly, a third of the class sessions can be devoted to explicit preparation of students for upper level courses. To use the operating systems course as example; explicit memory allocation can be revisited to introduce students to some allocation algorithms and deeper explanations on the operation of the *malloc* system call. In addition, system calls

regarding other aspects of operating system can be introduced to students through simple exercises involving processes creation (e.g., *fork, join*), inter-process communication (e.g., *pipes, semaphores*), or even network communication (e.g., *sockets*) in preparation for a networking course. Furthermore, garbage collection algorithms can also be introduced to have students write their own GC library in C **[16]**.

Although this type of content is not entirely novel, its progression and rationale significantly departs from current offerings. For better appreciation of this argument, the reader is invited to consider the most popular programming textbooks employing the C language. Too many are introductory in nature and approach the C language as if it was meant for absolute beginners **[18]**. The focus of such texts is typically on program design issues and elementary control flow structures. At the other end of the spectrum, system-programming texts focus on system calls, often in a Linux/Unix environment, and fail to build an explicit link with the previous programming knowledge of students. While the latter is much closer to the model suggested in this paper, some improvements can be adopted in order to better integrate the material to support the rest of the system-oriented components of the curriculum.

5. Pedagogical aspects: do you have a CLUE?

The findings of the survey indicate that C has a niche and would most likely integrate best with current curricular practice if introduced in a course after the CS-1/CS-2 pair. At this point, the pressing issue to be addressed is the pedagogical approach that would best serve such a course and how it can be justified as an elective among the already crowded list of elective courses.

Many responses in the survey echoed concerns exposed sometime ago about the need for an appropriate development environment to learn C **[10]**. Since then, many projects have targeted other languages by developing excellent educational IDEs **[2, 12, 19]**. Some even applied these design principles to C **[20]**. Beyond syntax highlighting and debuggers, C could also benefit from visualization tools to help students understand concepts such as memory allocation and stack management. These would definitively enhance the pedagogy of a course by providing students with different learning channels (i.e., visual or otherwise) that might be more suited to their learner type **[21, 22]**. For instance, engineering students are mostly visual in their learning type. It is reasonable to think that complementing a debugger with such visualization tools would improve teaching effectiveness. In order for such a tool to address the concerns about the difficulty the students encounter in interpreting error messages, the authors of this survey propose to use a C language interpreter, which would perform elementary code analysis to produce more intuitive error messages and warn students about common syntactical pitfalls. Although this approach is similar to others **[10],** it is also meant to complete such an interpreter with an automatic bug detection tool inspired by modern code inspection tools **[23]**. These ideas are being investigated in the context of a C Language Undergraduate Environment (CLUE) toolkit that will be composed of a C interpreter, a memory visualization component, and a rudimentary code analysis tool **[24]**.

If system-related topics are not to be covered at all in a given curriculum, such a course would most likely be useless. However, if the intent is to prepare students to the many aspects of the computing disciplines, including system-level topics, then such a course can help balance the preparation of students to a wider variety of upper level

undergraduate and ultimately graduate courses. Let us compare the operating systems and software engineering undergraduate courses in order to illustrate this argument. Both are upper level undergraduate courses, yet they benefit from different pedagogy at the curricular level. In the case of software engineering, early programming courses carefully lay down the basic concepts to help students progressively build their design skills in addition to programming ones. This approach is pedagogically sound and follows the principle of introducing early and repeating often what is important **[2]**. On the other hand, concepts such as concurrency and system calls are often absent from the CS-1+CS-2 progression and are left to the very end of the course tracks. As a result, too many institutions throw students, who have only experience in high level languages and software engineering, directly into a three-credit operating system course in which they are to be initiated, at lightning speed, to another entirely different aspect of the computing disciplines. While the operating systems course is meant to discuss operating systems internals, it is disturbing to do so in front of an audience who never experienced making a simple system call. The classic use-modify-create progression, known to object-first proponents, is not implemented in such courses. This situation forces instructors to cram together what should be two distinct courses or water down the material to a level below of what it should be. A few members of the SIGCSE community have already underlined the tremendous importance of concepts such as concurrency and the benefits that could emerge from introducing them early, thus leveraging the same pedagogical arguments developed by the objects-first proponents.

Beyond these pedagogical arguments, it is ultimately the targeted learning outcomes at the department level which will dictate which area of the computing discipline will receive particular focus. This is an issue much more difficult to address and beyond the scope of this paper. The above recommendations are only meant to address the curricular paradox caused by teaching the C language late in system-level inclined curricula.

6. Summary

This paper reviewed results of a survey aimed at assessing the role of the C language in current curricula. These results clearly indicate that (i) C is not suitable for introductory and intermediate programming courses, while (ii) it is suitable for system-level programming in upper level courses. In addition, this paper discussed a wide spread “curricular paradox" by which C is required in upper-level courses but not taught in lower-level programming core courses (e.g. CS-1, CS-2, CS-3). This often results in C being “left as an exercise” for students to learn on their own when reaching these upper level courses. It makes a difficult language even more difficult, potentially pushing students away from system-oriented computing courses and wasting an opportunity to leverage the particularities of this language to strengthen the students’ programming skills. To remedy this paradox, it was suggested that C’s intrinsic difficulties ought to be addressed by the use of a proper set of development and visualization tools to effectively reinforce student programming discipline and technical knowledge of programming languages. The paper goes further to describe a targeted course, a proper timeline for introduction, and the progression of this particular content in the computing curriculum. The outcomes might be the development of a qualitatively different set of skills and knowledge in students through a motivated and significant exposure to concepts such as pointers, memory management, parameter passing, and variables’ scope. These concepts

are critical for both professional and graduates focusing on system aspects of our discipline such as operating systems but also compilers and advanced languages paradigms.

7. References

[1] Kolling, M., Quig, M., A. Patterson, J. Rosenberg, “The BlueJ system and its pedagogy”, Journal of Computer Science Education, special issue on learning and teaching object technology, vol 13, no 4, 12/2003

[2] Canning, J., Moloney, W., Rafyemehr, A., Rey, D., Reading types in C using the right left walk method, June 2004, ACM SIGCSE Bulletin , Working group reports from ITiCSE on Innovation and technology in computer science education ITiCSE-WGR '04, Volume 36 Issue 4

[3] Eckerdal, A. ; Berglund, A.What does it take to learn "programming thinking"?, ICER 2005, 1st International Computing Education Research Workshop; Seattle, Washington, USA. ACM press; 2005. pp. 135-43.

[4] Fitzgerald, S.; Simon, G.; Thomas, L., Strategies that Students Use to Trace Code: An Analysis Based in Grounded Theory, ICER 2005, 1st International Computing Education Research Workshop; Seattle, Washington, USA. ACM press; 2005.

[5] Stephen, N., Freund, Roberts, E,S., Thetis: an ANSI C programming environment designed for introductory use, March 1996, ACM SIGCSE Bulletin , Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education SIGCSE '96, Volume 28 Issue 1

[6] Cross, J.H., jGRASP: an integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond, April 2006, Journal of Computing Sciences in Colleges, Volume 21 Issue 4

[7] Koenig, A., C Traps and Pitfalls, 1 ed. Addison-Wesley Professional; 1989 Jan.

[8] Van Der Linder, P., Expert C Programming: Deep C Secrets. SunSoft Press; 1994

[9] Stroustrup, B., The C++ Programming Language (Special 3rd Edition), Addison-Wesley Professional; 3 edition (February 15, 2000)

[10] Blunden, B., Memory Management: Algorithms and Implementations In C/C++ , 2002, wordware publishing

[11] C How to program, 5/e, Deitel

[12] Allen, E., Cartwright, R., Stoler, B., DrJava: a lightweight pedagogic environment for Java, February 2002, ACM SIGCSE Bulletin , Proceedings of the 33rd SIGCSE technical symposium on Computer science education SIGCSE '02, Volume 34 Issue 1

[13] Demetrescu, C.; Finochi, I., Leonardo: A C programming environment for reversible execution and software visualization. [Web Page] 1999; http://www.dis.uniroma1.it/~demetres/Leonardo/. [Accessed Apr 2006].

[14] Felder, R.M., Silverman, L.K., "Learning and Teaching Styles in Engineering Education," Engr. Education, 78(7), 674-681 (1988)

[15] Larsen, J., McCright, P.R., Weisenborn, G., Coordinating Sensory Modality in Learning Styles and Teaching Styles in Undergraduate Engineering Education

[16] Engler, D., Coverity, available at http://coverity.com/

[17] Gaspar, A.; Ejnioui, A.; Boyer, N., CLUE: C Learning Undergraduate Environment, University of South Florida at Lakeland Scholarship Day, May 2006, Lakeland, FL, Animated slides available at: http://softice.lklnd.usf.edu/~alessio/2006a-clue.pps

[18] Gaspar, A., Ejnioui, A., Boyer, N., Role of the C language in modern computing curricula, Survey monkey results available at http://surveymonkey.com/DisplaySummary.asp?SID=3039744&Rnd=0.9342091