Teaching Imperative Programming With Contracts at the Freshmen Level

[Experience Report]

Frank Pfenning Thomas J. Cortina William Lovas
Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu tcortina@cs.cmu.edu wlovas@cs.cmu.edu

ABSTRACT

We describe the experience with a freshmen-level course that teaches imperative programming and methods for ensuring the correctness of programs. Students learn the process and concepts needed to go from high-level descriptions of algorithms to correct imperative implementations, with specific applications to basic data structures and algorithms. A novel aspect of the course is that much of it is conducted in C0, a small safe subset of the C programming language, augmented with a layer of annotations to express *contracts* that are amenable to verification. The present version of the course assumes a basic understanding of programming (variables, expressions, loops, functions) and prepares students for subsequent computer systems courses taught in C as well as more advanced courses on algorithms and data structures.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants, pre- and post-conditions*; D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, correctness proofs, programming by contract*

General Terms

Algorithms, languages, theory, verification

1. INTRODUCTION

Recently, the School of Computer Science at Carnegie Mellon University has engaged in a major introductory curriculum revision[1], affecting all freshmen-level and some sophomore-level courses. This curriculum revision exposed the need for a course to prepare students for later courses such as *Computer Systems*, *Networks*, and *Operating Systems*, which require imperative programming skills and knowledge of the C programming language [7]. In course design we assumed some programming background, either from high school or from an introductory course taught at our university, but not any specific language or paradigm. Having taught our course now three times to a total of 220 students, both computer science majors and non-majors, we have indeed observed a diversity of backgrounds, including most students knowing only Java or only Python, and only a few students who were already familiar with C.

In our prior experience in teaching a sophomore-level systems course over several years, we found that many students were poorly prepared to deal with imperative programming and the vagaries of C, especially for assignments involving nontrivial data structures. We conjectured two root causes. One is a lack of understanding of how to reason logically about programs, which in turn can be traced to a lack of precision regarding the desired properties of code. Another is the well-known complexities of C, such as manual memory management, undefined behavior on out-of-bounds array accesses, arithmetic overflow, and some casts, which confound even experienced programmers [9].

We set out to develop a course curriculum to eliminate these deficiencies. The cornerstone of our approach was the design of a small safe subset of C called C0 (pronounced "C-naught") that would be appropriate for teaching basic algorithms and data structures. C0 is garbage-collected, checks for out-of-bounds array accesses and null-pointer dereferences, permits no casts, and has an unambiguous semantics for arithmetic expressions based on modular arithmetic. We augmented this base with a layer of annotations to express *contracts* [8]. These take the form of pre- and post-conditions for functions, as well as loop and data structure invariants. Currently, contracts are checked dynamically (during program execution), when the C0 compiler is invoked with an appropriate flag; in the future, we hope to add some tools for verifying contracts statically (during compilation).

We refined our ideas into specific integrated learning objectives along three dimensions: *computational thinking* [11], *programming skills*, and *algorithms and data structures*. We then developed a series of lectures to achieve these goals. Our experience with the resulting course, which has by now been delivered three times to a total of 220 students, has been quite positive. In this paper we provide a sketch of

[1]`http://link.cs.cmu.edu/files/ugrad-report.pdf`

the most significant aspects of this course and report on our experience in delivering it, both to majors and nonmajors. We also draw some preliminary conclusions and speculate on future work.

2. LEARNING OBJECTIVES

The learning objectives for this 15-week course are complex and interrelated, and yet we found it useful to organize them along three dimensions: *computational thinking*, *programming skills*, and *algorithms and data structures*. Before most lectures, we would explicitly state the goals or topics for that particular lecture, organized along these dimension and believed it has been helpful for students to put material of the lecture into a broader context.

If the goals below appear ambitious for a second course in computer science with a mild programming prerequisite, they are! Anticipating one of our conclusions, we believe that it is precisely the simplicity of C0 combined with the explicit nature of contracts that have allowed us to achieve these goals for most students within a semester, for majors and nonmajors alike, and regardless of the nature of prior programming experience.

2.1 Computational Thinking

In the area of computational thinking, successful students should be able to

- **understand abstractions and interfaces**. They are essential for the modular organization of programs, and for the ability to reason about large parts of a program independently.
- **relate specifications to implementation**. In order to write correct programs, it is important to articulate what "correctness" means, and in which ways program may have or may fail to have this elusive property.
- **express pre- and post-conditions for functions and loop invariants**. Pre- and post-conditions achieve at the level of individual functions what abstractions and interfaces achieve at the level of whole data structures: they allow us to decompose the correctness of a complex program into the correctness of individual functions. And loop invariants achieve the same at an even finer grain, at the level of individual loops: they allow us to establish the correctness of multiple iterations by proving properties of one single iteration.
- **use data structure invariants**. They allow us to implement and use correct and efficient operations on the elements of the data structure.
- **reason rigorously about code, both logically and operationally**. Reasoning is required to establish correctness and find bugs. Logical reasoning abstracts from the detail of an implementation (across a loop, function, or abstract interface), exploiting only the essential properties. Operational reasoning traverses small pieces of code to analyze their effect or value.
- **analyze asymptotic complexity and practical efficiency**. Beyond functional correctness, we also need to understand the characteristics of the running times of programs, in relation to properties of the input. Only then can we chose appropriate data structure implementations when solving problems.

2.2 Programming Skills

In the area of programming skills, successful students should be able to

- **understand the static and dynamic semantics of their programs.** In other words, students should be able to determine if their program will compile and how it will execute. This, of course, is fundamental to reasoning.
- **develop, test, rewrite, and refine their code**.
- **work with specifications and invariants**. Once we have decided that contracts should be explicit, they become and integral part of the programming process.
- **use and design small interfaces**. Competent programmers cannot merely use libraries, but they must be able to write libraries themselves, and that means weighing the consequences of various choices regarding how much to hide and how much to expose.
- **use and implement mutable data structures**. Because this course is on imperative programming, our emphasis is on traditional mutable data structures.
- **render high-level algorithms into correct imperative code**. The bridge here is, of course, the use of explicit contracts and interfaces to rigorously capture the high-level ideas.
- **write C programs in a Unix-based environment**. This skill is at present needed for many internships and for follow-on systems courses.

2.3 Data Structures and Algorithms

In the area of *data structures and algorithms*, successful students will be able to

- **perform asymptotic analysis on sequential computation**, including simple amortized analysis and recognition of common important complexity classes ($O(1)$, $O(\log(n))$, $O(n)$, $O(n * \log(n))$, $O(n^2)$, $O(2^n)$).
- **apply the divide-and-conquer strategy in elementary algorithm design**, including binary search and subquadratic sorting.
- **understand properties of simple self-adjusting data structures**, such as heaps or self-balancing binary search trees.
- **effectively employ a number of basic algorithms and data structures**, including stacks and queues, hash tables, heaps, balanced binary search trees, tries, binary decision diagrams, and simple graph algorithms.

3. PROGRAMMING LANGUAGE

We believe that computational thinking, programming skills, and algorithms and data structures must go hand-in-hand when introducing students to computer science. Each component in isolation is difficult or impossible to master without at least some exposure to the other two. In particular, programming itself is central, and the choice of programming language has a dramatic impact on how effectively concepts can be taught. Java has been a popular language [5] for

CS1/CS2 and has some clear advantages, for example, when discussing interfaces. However, the complexity of the underlying object-oriented model, including dynamic dispatch and inheritance, can obscure the simplicity of pure algorithmic ideas, especially when the primary course goals relate to imperative programming and mutable data structures. The course objectives include some skills in C, but pitfalls and idiosyncrasies of C [9] make it unsuitable for a course with only mild programming prerequisites. So we chose to work in a small safe subset of C augmented with annotations to express explicit contracts for the first 11 weeks, and then switch to C in week 12–15, carrying forward a variant of the contract language implemented as C macros. Revealing a language in layers, using well-defined and enforced subsets, has recently also been proposed by Felleisen [6] for pedagogical reasons.

We now review the structure of C0 and justify various design decisions. We provide the students with a compiler and an interpreter for C0 as well as several small libraries, mostly dealing with input/output and strings. The compiler performs parsing, type checking, and verifies some properties of the static semantics (for example, that variables are initialized before they are used), and then produces C code as output, which is in turn compiled to an executable. The interpreter performs the same checks but then executes the program directly.

3.1 Type Structure

The type structure is as simple as we could reasonably make it and still write natural programs to implement the various algorithms and data structures.

- `int`. Integers are interpreted in modular arithmetic with a 32-bit two's complement representation. They also support bit-wise operations so one can implement, for example, image manipulation using integer arrays in the ARGB color model. A salient difference to C is that the results of all operations are defined. The behavior of integer operations is therefore consistent with the C specification.

- `bool`. Booleans have just two values, `true` and `false`, and can be tested with conditionals as well as the usual short-circuiting conjunction `&&` and disjunction `||`. While consistent with C, booleans are not conflated with ints, avoiding common mistakes and providing a clear foundation for contracts which are expressed as boolean conditions (see Section 3.3).

- `t[]`, the type of arrays of values of type t. C0 distinguishes arrays from pointers. Arrays have a fixed size determined at allocation time, and are compiled so that array accesses can be dynamically checked to be in array bounds. In C, the type `t[]` would be written as `t*`, which does not provide the opportunity for bounds checks. Fixed size arrays from C are not supported in C0.

- `t*`, the type of pointers to cells holding values of type t. Pointers may be `NULL`. Unlike C, we cannot perform any pointer arithmetic of values of type `t*`.

- `struct s`, the type of structs (also called records) with name s. Structs must be explicitly declared.

- `char`. These are ASCII characters as in C, restricted on the range from 0 to 127. They can not be implicitly converted to or from ints.

- `string`. String are an immutable abstract type, but the runtime system provides library functions to convert between strings and character arrays (`char[]`).

Because we distinguish arrays and pointers, and consequently array access and pointer dereference, and further disallow pointer arithmetic, C0 permits a simple type-safe and memory-safe implementation. In particular, it is amenable to garbage collection, avoiding the problems of `malloc` and `free`. We currently use the conservative Boehm-Weiser collector [2]. This is of significant benefit to the students, who can write complex data structures without having to worry about obscure segmentation faults or bus errors.

3.2 Control Structure

The control structure is quite conventional. C0 separates expressions from statements, where assignments are considered statements, eliminating yet another class of nefarious bugs. Unlike C, expressions are guaranteed to be evaluated from left to right, eliminating another source of implementation-dependent unpredictable behavior. Variables must be declared and initialized before use, which is checked with a simple dataflow analysis. Arrays, when allocated, are initialized with default values which are specified for each type. We have conditionals (`if` and `if/else`) and loops (`while` and `for`). Functions take a fixed number of arguments of fixed types, and either return a value of fixed type or no value (`void`).

3.3 Contracts

The contract language is loosely based on a tiny subset of JML [3] and Spec# [1]. Preconditions for functions are expressed in clauses of the form `//@requires e;`, where e is a boolean condition. A call to the function is considered to be in error if e evaluates to false. Postconditions for functions are expressed as `//@ensures e;`, where e may mention the special variable `\result`. A function is considered to be in error if it returns a value that does not satisfy the postcondition when given arguments that satisfy the precondition.

Here is a simple example of an integer logarithm function, with a minimal contract specifying only some properties on the legal range of argument and result.

```
int log(int x)
//@requires x >= 1;
//@ensures \result >= 0;
{ int r = 0;
  while (x > 1) {
    x = x / 2;
    r = r + 1;
  }
  return r;
}
```

The use of boolean expressions instead of logical propositions has some advantages and drawbacks. The big advantage is that students do not have to learn an additional language for specifications, and that contracts remain effectively checkable. The disadvantage is that quantifiers are not available, and specific uses of (bounded) quantifiers have to be coded as ad hoc functions. We feel that the benefits of staying with a simple, uniform, executable language outweigh the loss of expressive power.

As a second example, consider a simple linear search through an unsorted array. We pass in the element x we are looking for, and array A, and a bound n which must be less or equal to the length of the array. We either return -1, or an index i such that $A[i] = x$. What we do *not* express in the contract is that we return -1 only if x is not in the array. In the absence of quantifiers, we could express this only by writing another, almost identical function which returns true if x is in the array and false otherwise.

```
int linsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures -1 <= \result && \result < n;
//@ensures \result == -1 || A[\result] == x;
{
  for (int i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    if (A[i] == x) return i;
  return -1;
}
```

In this example we also use a loop invariant. A loop invariant is checked *just before the exit condition*, so it must include the extremal value ($i \leq n$, not $i < n$).

3.4 Transition to C

Note that the code above compiles and behaves correctly in C if we just replace `int[] A` with `int* A`. However, it suffers from the possibility of out-of-bounds array access if n should be incorrect, while the version above is safe and would abort. It turns out that replacing `[]` by `*` is the only change we have to make so that C0 programs are syntactically correct in C and compile. In other words, students have been secretly using and learning (almost) C all along! In that way, we follow Felleisen's advice [6] to present a language in enforced layers. On the other hand, wanting to reason soundly about programs means that C0 cannot always be semantically compatible with C. For example, an overflowing addition must be handled according to modular arithmetic laws in C0, but the result is undefined for C.

The transition to C around week 11 of the course is then centered on four different topics.

1. **Undefined behaviors of C**. We enumerate the undefined behaviors of C and teach safe programming practices. Many of these are consistent with good C0 practice. An important tool, especially on out-of-bounds memory accesses and null pointer dereferences, is a set of C macros we provide for the students that emulate contracts through assertions.

2. **Manual memory management**. We introduce the students to manual memory management with `malloc` and `free`, heavily relying on the Valgrind tool [10] to detect memory bugs and leaks. We also discuss stack allocation and the address-of operator (`&`) which has important safety consequences.

3. **Genericity**. We discuss implementation techniques for generic data structures using `void*`, function pointers, and casts.

4. **Additional language constructs**. We highlight a few important additional features of C not present in C0, such as integer types of different ranges and `switch` statements. However, our coverage is not complete, relying on students to be able to read sample code, tutorials, or the standard reference [7] to discover the rest of the language for themselves.

The course culminates in a final assignment where students implement a virtual machine for C0 in C, nicknamed C0VM. Since our C0 compiler can produce byte code as output, the students' byte code interpreters can execute almost all the C0 code they wrote throughout the semester, excepting only those using the image library from the first project. The students found this to be a challenging but very satisfying project. Not only is it an appropriate use of C, but it allows them to reflect on the details of the operational meaning of programming constructs they have been writing all along. Finally, it exercises the computational thinking idea that we often have to view programs as data.

4. COURSE IMPLEMENTATION

The first course instance was offered to incoming computer science freshmen in their first semester at our university, if they have had some programming experience from high school. It turned out all 101 students starting this course (93 of which completed it) had prior programming experience with Java, which is syntactically and semantically close to C0. Java objects are decomposed into the more primitive structs and pointers, but these are not discussed immediately.

The second instance of the course was offered to both majors and nonmajors. This course started with 127 students of which 114 remained at the end. The final breakdown by college was 48 computer science majors, 29 engineering majors, 22 science majors, 9 humanities majors, 5 interdisciplinary majors, and 1 business major. Their programming experience was almost exclusively from the CS1 course in our new curriculum, delivered in Python. This provided a much more difficult transition. Syntactic differences, as well as the differences between static typing in C0 and dynamic typing in Python required some additional lectures on the C0 language itself at the beginning of the course, which we then integrated into the curriculum, trading them for two advanced lectures.

The third instance of the course was compressed into six weeks in the summer, with daily lectures, and followed the same curriculum as a second instance. It was a small class of 14 students, 11 of which completed it.

4.1 Lecture Topics and Projects

Here is a very brief outline of the lecture topics, by week.

- Week 1: Course overview, contracts, introduction to C0 (functions, statements, expressions, types)
- Week 2: Modular arithmetic, arithmetic and bitwise operations, arrays, loop invariants.
- Week 3: Linear and binary search, divide and conquer, asymptotic complexity.
- Week 4: Sorting algorithms, mergesort, quicksort.
- Week 5: Queues, stacks, linked lists, pointers, recursive types, data structure invariants.

- Week 6: Memory layout, recursion, *Midterm Exam I.*
- Week 7: Unbounded arrays, amortized analysis, hash tables.
- Week 8: Interfaces, priority queues, heaps, ordering and shape invariants.
- Week 9: Restoring invariants, heapsort, binary search trees.
- Week 10: AVL trees, rotations, program testing.
- Week 11: Polymorphism, introduction to C, *Midterm Exam II.*
- Week 12: Memory management (malloc/free), `valgrind`, generic data structures.
- Week 13: Virtual machines.
- Week 14: Tries, decision trees, binary decision diagrams, sharing, canonicity.
- Week 15: Graph algorithms (spanning trees, union-find).

In addition to the two midterms and the final, we gave a total of 8 quizzes online, and 8 homework assignments, which accounted for 45% of the final grade. Each assignment except the last one had both a written and a programming component, practicing different aspects of the material. The topics of the programming portions of the assignments were:

1. Project 1: Image manipulation (arrays, bit-level operations, loops)
2. Project 2: Text processing (searching and sorting)
3. Project 3: Word ladders and parsing (more searching)
4. Project 4: Gap buffers for editors (doubly linked lists, arrays revisited)
5. Project 5: Lights out game (hash tables, backtracking)
6. Project 6: Huffman coding (binary trees and heaps)
7. Project 7: Ropes (strings with fast concatenation, in C)
8. Project 8: A virtual machine for C0 byte code (in C)

4.2 Sample Lecture Material

Here is a sample abstract for the lecture on heaps, as used to implement priority queues.

> *In this lecture we will implement operations on heaps. The theme of this lecture is reasoning with invariants that are partially violated, and making sure they are restored before the completion of an operation. We will only briefly review the algorithms for inserting and deleting the minimal node of the heap; you should read the notes [on priority queues] and keep them close at hand.*
>
> *Temporarily violating and restoring invariants is a common theme in algorithms. It is a technique you need to master.*

Here is the code for `is_heap_except_up(H, n)`, which checks that H is almost a valid heap, allowing a violation only between node n and its parent. It is used only in contracts.

```
/* is_heap_except_up(H, 1) == is_heap(H); */
bool is_heap_except_up(heap H, int n) {
  if (H == NULL) return false;
  //@assert \length(H->heap) == H->limit;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  /* check parent <= node for all nodes
   * except root (i = 1) and n */
  for (int i = 1; i < H->next; i++)
    //@loop_invariant 1 <= i && i <= H->next;
    if (!(i == 1 || i == n || H->heap[i/2] <= H->heap[i]))
       return false;
  return true;
```

This function is used as a loop invariant in the *sift up* operation which is employed after the new element is inserted at the end of the array.

```
void sift_up(heap H, int n)
//@requires 1 <= n && n < H->limit;
//@requires is_heap_except_up(H, n);
//@ensures is_heap(H);
{ int i = n;
  while (i > 1)
    //@loop_invariant is_heap_except_up(H, i);
    {
      if (H->heap[i/2] <= H->heap[i]) return;
      swap(H->heap, i/2, i); /* swap i with parent */
      i = i/2; /* consider parent next */
    }
  //@assert i == 1;
  //@assert is_heap_except_up(H, 1);
  return;
}
```

4.3 Student Performance

Generally, students' performance exceeded our initial expectations. The attrition rate was low: 8/101 for first instance with majors, and 13/127 for second instance with mixed student population. The average of the overall course grade for the first course instance (majors only) was 83.4% with a standard deviation of 8.0%, with 2 students failing. The average for the second instance (mixed population) was 79% with a standard deviation of 9.0%, with 1 student failing. When the course was taught with a mixed population, and an ANOVA analysis revealed no statistically significant difference in performance between students from different colleges. By and large, we felt, the ambitious learning goals we set were achieved, although at present we do not yet have information from the downstream systems course regarding students' programming performance in C.

4.4 Student Comments

We collected early student feedback in all three instances—just before the first midterm in the first two and halfway through the summer instance—and then again at the end of the semester through a university-wide course evaluation system. Early feedback revealed that clear majority students found the pace of the course to be fine, with some claiming it to be a little fast, and a few way too fast. However, the middle section of the course was most comfortable for the students, with an accelaration near the end, when it transitions to C and students have to implement a conceptually difficult virtual machine in their last project. Among the course aspects most helpful to student learning were, in order, recitations, lecture notes, lectures, and assignments.

Here are some results from the online course evaluations conducted by the university. Numerical scores (except hours per week) are averages on a scale of 1 to 5, with 5 being best.

Category	Sem. 1	Sem. 2	Summer
hours per week	11.22	10.02	12.71
clear learning goals	4.29	4.31	4.71
feedback to students	3.69	3.68	4.57
importance of subject	4.41	4.56	4.86
explains subject matter	4.24	4.39	4.86
overall course	4.10	4.08	4.71

As is clear from these data, and the students comments, the most unsatisfying aspect of the course was the slow feedback to the students, which was due to a small course staff overwhelmed with developing the course materials. Despite this, the overall ratings for the course were very good. Students generally responded favorably to learning C0 first before moving on to C, once the pedagogical reasons were outlined to them in class. Here are a couple of somewhat representative comments:

> *Overwhelming at first - a gentler introduction would be preferrable (especially the first homework assignment, but all of the first couple of weeks in general). A good overview that taught a lot about many different areas. The shift to C could have been jarring, but it was handled well.*
>
> *I thought the material of the course was very good, and C0 is a very nice language to learn before transitioning into C.*
>
> *[This class] hasn't just taught me the material excellently, it's given me a better appreciation for computer science and made me a much, much better programmer.*

5. CONCLUSIONS AND FUTURE WORK

We have sketched a new freshmen-level course on imperative computation, which we have now taught three times to a total of about 220 students. It rests on the integrated teaching of computational thinking, programming skills, and data structures and algorithms. Perhaps the most novel aspects of the course are the emphasis on explicit contracts and reasoning about correctness throughout, and the use of C0, a small safe subset of C supporting contracts. All course materials are freely available on-line[2] We expect a public release of C0 some time this fall.

Our experience with the course has been very positive. We believe that removing complexity from an industrial-strength programming language and augmenting the results with means to express contracts, are techniques that have enabled students to reach our ambitious learning goals. They are not just based firmly in the theory of programming languages, but pedagogically sound.

During the first course instance, a guest lecturer from the Microsoft Windows development team talked about the pervasive use of lightweight specifications in SAL [4] in the implementation of Windows and other Microsoft products. Our course now teaches our students similar languages and programming techniques at the freshmen-level. In future work we would like to consider if we can shift some of the work of checking contracts from run-time to compile-time. Recent advances in program verification are encouraging, but are not yet at a stage where we know how to use them effectively in introductory computer science. The present course provides an excellent setting to explore how this might be accomplished.

[2] `http://www.cs.cmu.edu/~fp/courses/15122`

Acknowledgments

Many have contributed to the course curriculum, the C0 programming language, and its tool suite. In particular, we would like to acknowledge Rob Arnold, the principal developer of C0, and Laura Abbott, Jason Koenig, Karl Naden, Rob Simmons, Nathan Snyder, Anand Subramanian, and Jakob Uecker who all have contributed to various parts of the language design and implementation.

The development of C0 and the 15-122 course curriculum has been supported by the MSR-CMU Center for Computational Thinking under a project on *Specification and Verification in Introductory Computer Science* and by a gift from Google for the development of CMU's introductory undergraduate courses.

6. REFERENCES

[1] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Communcations of the ACM*, 6(54):81–91, 2011.

[2] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, pages 807–820, Sept. 1988.

[3] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO'05)*, pages 342–363. Springer LNCS 4011, 2005.

[4] M. Das. Formal specifications and industrial-strength code — from myth to reality. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, page 1. Springer LNCS 4144, 2006. Invited talk.

[5] S. Davies, J. A. Polack-Wahl, and K. Anewalt. A snapshot of the current practices in teaching the introductory programming sequence. In *Proceedings of the 42 ACM Technical Symposium on Computer Science Education (SIGCSE 2011)*, pages 625–630, Dallas, Texas, Mar. 2011.

[6] M. Felleisen. TeachScheme! In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE 2011)*, pages 1–2, Dallas, Texas, 2011. Keynote talk.

[7] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[8] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.

[9] R. C. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, 2006.

[10] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, Apr. 2005.

[11] J. Wing. Computational thinking. *Communications of the ACM*, 49(2):33–35, 2006.