An Approach to Teaching to Write Safe and Correct Imperative Programs — even in C[*]

Iliano Cervesato
Carnegie Mellon University
iliano@cmu.edu

Thomas J. Cortina
Carnegie Mellon University
tcortina@cs.cmu.edu

Frank Pfenning
Carnegie Mellon University
fp@cs.cmu.edu

Saquib Razak
Carnegie Mellon University
srazak@cmu.edu

ABSTRACT

C is widely used for low-level system programming. As such, it is a staple of systems courses. Manual memory management, undefined behaviors and the many other vagaries of C make it challenging to learn (and teach). In this paper, we describe our approach to teaching safe and correct imperative programming in C. It is based on first exposing students to a custom programming language that is a small safe subset of C, augmented with a layer of executable contracts. These contracts allow students to express the pre- and post-conditions of functions, to write loop invariants, and to capture data structure invariants. Contracts help students gain a deeper understanding of their code, thereby preempting many unsafe or incorrect statements. Being executable, contracts allow them to quickly identify remaining bugs. Once students have mastered writing safe and correct code in this language, we transition them to C. We have adapted our introductory data structures course (CS2) to use this custom language. This experience report is based on teaching this course for the last eight years to many thousand CS majors and non-majors.

CCS CONCEPTS

• **Social and professional topics → Computing education**; • **Theory of computation → Program reasoning**; • **Software and its engineering → Imperative languages**;

KEYWORDS

Teaching C; Imperative Programming; Safety; Correctness; Executable contract

1 INTRODUCTION

The C programming language [10] remains as popular as ever for writing low-level system code.[1] Therefore, many curricula that teach hands-on courses on networking, embedded system, operating systems and the like need to acquaint their students with the C programming language. Teaching students to become proficient in C is a challenge as they encounter, often for the first time, the well-known complexities of this language, such as manual memory management, undefined behavior on out-of-bounds array accesses, arithmetic overflow, and implicit casts, which confound even experienced programmers [18].

The purpose of this paper is to describe the approach that our university has taken to address this challenge at the undergraduate level. This approach revolves around three main ideas:

- We separate the task of teaching students low-level imperative programming from exposing them to the idiosyncrasies of the C language. We achieve the former by starting them off in a safe subset of C called C0 (pronounced "C-naught"). 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. Once students have mastered writing substantial programs in C0, we transition them to full-fledged C: it is at this point that they learn to manage memory explicitly, become acquainted with some of the more common undefined and implementation-defined behaviors of C, and get exposed to popular idioms and pitfalls of this language. Students hone their C programming skills on several more projects.
- One of the difficulties of writing imperative code is keeping track of the many moving parts of a program or even a mid-size function. Failure to do so results in programs that crash unexpectedly or compute incorrect answers. Beginners are particularly affected, as they have not yet gained the experience and developed techniques to mitigate these problems. This results in long and frustrating debugging seances, often rooted in trial and error. We addressed this issue by engineering a layer of executable contracts [11] into our language. The contract layer supports writing pre- and post-conditions, loop invariants as well as generic assertions. Contracts achieve two purposes: first, by having them write contracts in their code, students are trained to think about what they want this code to do, which gives them the deep understanding needed to write safe and correct programs; second, because contracts in C0 are executable, they help students quickly identify and fix many lingering bugs. We call this approach *deliberate programming*. We have written a small contract library for C, which allows students to continue using these tools once they transition to C.
- Given that several of our advanced undergraduate courses rely on C, a question is when to teach this language within our curriculum. We postulated that a good place to do so is in our introduction to data structures course — essentially our CS2. Therefore, students at our university learn to use and implement stacks, hash tables, trees and other data structures at the same time as they learn C. More precisely, they learn the majority of these data structures using C0, which is safe and well behaved, with

[*]Draft manuscript, January 2019
[1]According to the August 2018 TIOBE index, C is the second most popular programming language overall, just after Java and well ahead of C++ and Python.

just a few additional data structures (mainly graphs) taught after the transition to C. This course is taken by CS majors and non-majors alike. In fact, nowadays, approximately two thirds of all students who start at our university each year take this course, most during their freshman year.

Our university has relied on this approach to teaching C, as well as basic data structures, for the past eight years. As we rolled it into our curriculum (initially just for CS majors), student performance in our systems courses did not decline and in fact has improved as more non-majors started taking them. This experiment is now well beyond its pilot phase at our university and we postulate that it can be adapted or even replicated at other institutions. In fact, we are in the process of producing an online version of our introduction to data structure course, which will facilitate adoption.

The rest of this paper is structured as follows: after reviewing related work in Section 2, we elaborate on the three main ideas of our experiment, the design of C0 in Section 3, deliberate programming in Section 4, and highlights of our data structures course in Section 5. We then proceed to an evaluation of our approach in section 6. We discuss directions of future work in Section 7.

2 RELATED WORK

Contracts have been used in programming languages for quite some time. Design-by-contract is a technique that describes the interface between a client and a library in terms of what the library expects and what it delivers [13–15]. These specifications are not executed in most programming languages and it is the responsibility of the programmer to check for these contracts using assertions. Our work is most closely related to design-by-contract techniques implemented as part of the Eiffel programming language [11, 12]. Eiffel presents design-by-contract in the context of object-oriented design where subtasks use pre-conditions that the clients of the code need to fulfil and specify post-conditions as the deliverables for these subtasks. The class invariants aim to keep the state of each object consistent with its purpose. In the course presented in this paper, we focus on using contracts to prove the correctness of code at the algorithmic level using loop-invariants to prove post-conditions and data-structure invariants to specify consistency of data structures. The focus is to help students develop good programming habits in the imperative domain. Although the importance of teaching introductory programming through contracts was emphasised more than two decades back in [20], to the best of our knowledge, the course design presented in this paper is the first time contracts are incorporate as an intrinsic part of a whole course on imperative programming.

Work has been done to add functionality to existing languages in order to provide executable contract support. *Handshake* is a library that adds contracts to Java without any modification to the Java Virtual Machine or library classes [6]. The Java Modeling Language (JML) provides specification constructs like pre- and post-conditions and invariants [5]. C0 was heavily inspired by JML.

Language subsetting is another area of active work in teaching introductory programming courses. In language subsetting, a subset of a standard programming language is used to introduce students to programming concepts. One of the most common examples of language subsetting is Dr. Scheme used to teaching functional paradigm to students [8]. MiniJava is a subset of Java that allows students to focus on the fundamental concepts of object-oriented programming rather than on mundane details [17]. Cyclone is a safe dialect of C that performs static analysis of code in order to prevent safety violations [9]. Cyclone adds run-time checks in compiled code in places where static analysis cannot determine the safety of a program. Where Cyclone takes on the responsibility of checking for code safety, design of C0 forces the students/programmers to think about code safety, while making sure that language specification itself does not allow for undefined behavior. Nina et al. designed a CS1 course where "programming by contract" was a fundamental principle of the course [16]. This course focused on the "object first" approach to teaching object-oriented programming to students with no programming background. The course used an external library to check for the validity of pre-conditions. In the design of our course, we focus on safety and correctness of programs and algorithms using loop invariants to prove post-condition for correctness in functions and imposing invariants preserve data struture consistency in the imperative domain.

3 LANGUAGE DESIGN

Central to our approach to teaching C is our reliance on C0. This language is largely a subset of C, so that students program in C from day one[2]. It protects students from the vagaries of C by being free of undefined behaviors: executing an instruction in C0 either works like in C (or is compatible with the C99 standard) or it aborts execution.

In this section, we review the structure of C0 and justify various design decisions. The contract sublanguage of C0 is the subject of Section 4. 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 current 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 of C0 is as simple as we could reasonably make it and still write natural programs to implement many common 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 manipulations 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 (the C library `<stdbool>` provides the type **`bool`**), booleans are not conflated with **`int`**s, avoiding common mistakes and providing a

[2]A handful of the most perilous primitives of C have been given new names in C0 to avoid confusion when later transitioning to C.

clear foundation for contracts which are expressed as boolean expressions (see Section 4).

- `t[]`, 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*`, and does not provide the opportunity for bounds checks.
- `t*`, pointers to cells holding values of type *t*. Pointers may be `NULL`. Unlike C, we cannot perform any pointer arithmetic on values of type `t*`. C0 supports generic pointers, whose type is **`void`**`*`, that need to be converted to a concrete pointer type (for example **`int`**`*`) before they can be accessed. Non-null pointers of type **`void`**`*` are internally tagged with the actual type of the cell they point to (for example **`int`**`*`) and attempting to access them as pointers of a different type aborts execution. Tags can be checked (only) in contracts to ensure safety.
- **`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 cannot be cast to or from **`int`**s.
- **`string`**. String are an immutable abstract type, but the runtime system provides library functions to convert between strings and character arrays (**`char`**`[]`). This design spares novices from the pitfalls of the representation of strings in C as `NUL`-terminated character sequences.
- Function types. Just like in C, the functions declared in a program can be passed as argument to other functions and stored in data structures. This supports forms of abstractions like defining generic hash tables, where the hashing function depends on the type of the value to be hashed.

Absent are different-size integers, which are best introduced when transitioning to C, floating point numbers which make reasoning about code difficult due to rounding errors, and advanced type constructors such as unions.

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 [4]. 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 of C0 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 — thus 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 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 Transition to C

As we said, C0 programs are for the most part valid C programs. The effort to turn a valid C0 program into a syntactically correct C program is minimal: replace `[]` by `*`, replace the primitives for allocating arrays and pointers with `malloc`, and change **`string`** to **`char`**`*`. In other words, students are secretly using and learning (almost) C all along! In this way, we follow Felleisen's advice [7] 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 some concrete C compilers. 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 is then centered on pointing out a few major differences.

Undefined behaviors of C. We highlight the most common 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 (see Section 4) through assertions.

Manual memory management. We introduce the students to manual memory management with `malloc` and `free`, heavily relying on the Valgrind tool [19] to detect memory bugs and leaks. We also discuss stack allocation and the address-of operator (&) which has important safety consequences.

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 [10] to discover the rest of the language for themselves.

We have found that students learn to navigate the differences between the two languages by converting one of their C0 programs to C and only then completing projects just in C.

4 DELIBERATE PROGRAMMING

An important aspect of C0 is that it embeds a language of executable contracts that promote a deliberate approach to writing code. We use "delibrate programming" as the term to describe our process of using design-by-contract and the implementation considerations of this design. The user can decide at compilation time whether contracts are to be checked (in which case contract failure aborts execution) or ignored (in which case they are treated as comments). The contract language is loosely based on a tiny subset of JML [5] 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 *unsafe* 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 *incorrect* if it returns a value that does not satisfy the postcondition when given arguments that satisfy the precondition. The contract language also supports loop invariants, written as `//@loop_invariant e;` just before a loop body, and assertions `//@assert e;` which can appear anywhere a statement is allowed.

Here is a simple example, the fast (logarithmic) power function, which makes use of all four types of contracts available in C0.

```
1  int fast_power(int x, int y)
2  //@requires y >= 0;
3  //@ensures POW(x, y) == \result;
4  {
5    int b = x;
6    int e = y;
7    int r = 1;
8    while (e > 0)
9    //@loop_invariant e >= 0;
10   //@loop_invariant POW(b, e) * r == POW(x, y);
11   {
12     if (e % 2 == 1) {
13       r = b * r;
14     }
15     b = b * b;
16     e = e / 2;
17   }
18   //@assert e == 0;
19   return r;
20 }
```

The precondition simply expects the exponent to be non-negative. The postcondition states that the returned value of `fast_power(x,y)` should be equal to x^y, which we express as `POW(x,y)`: this user-defined function (not shown) implements the standard recursive (linear-time) definition of integer exponentiation. Here, `POW` plays the role of a specification function — it too requires that the exponent be non-negative. Pre- and post-conditions summarize to a caller what the function expects and promises.

The second loop invariant, on line 10, captures the workings of the loop: it notes that, as the values of b, e and r are modified by the loop, the expression $b^e r$ remains constant and equal to x^y. The first loop invariant illustrates a use of contract to reason about code: how do we know that the two calls to `POW` on line 10 are safe, i.e., that their second argument (the exponent) is non-negative? For `POW(x,y)`, we can simply point to line 2 since y is not modified anywhere in the function. For `POW(b,e)`, one possible argument would go as follows: "*since e starts non-negative (by lines 2 and 6) and is always divided by 2, it will stay non-negative*". This type of arguments, which we label "operational", are error-prone as they expect the students to build a mental model of all possible executions. A simpler approach is to spin off this reasoning into a separate loop invariant (line 9). The safety of `POW(b,e)` is then supported by pointing to line 9. We teach students simple techniques for showing that loop invariants, like the ones on line 9 and 10, are valid.

A loop invariant is checked just before the loop guard. Thus, (valid) loop invariants must hold once we exit the loop. As such they play a critical role in showing that a function is correct. Here, the first loop invariant tells us that at this point in the execution $e \geq 0$. Since we have exited the loop, the loop guard is false and therefore $e \leq 0$. Combining these two pieces of information, we deduce that $e = 0$, which we (somehow pedantically) record as the assertion on line 18. By plugging this information in the second loop invariant (line 10), we get that $x^y = b^e r = b^0 r = r$, and r (which must be equal to x^y) is exactly what the function returns on line 19. As this example shows, contracts in C0 and the "point-to" approach to reasoning about code are central to infusing students with a deliberate programming mindset.

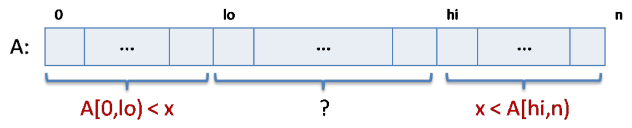
When designing C0, we limited the language of contracts to the standard boolean expressions of a typical imperative programming language (and of C0 in particular). The advantage is that students do not have to learn a dedicated specification language, and that contracts remain effectively checkable. Alternatively, we could have embedded a more complex specification logic in C0 — for example JML [5] has bounded quantifiers — thereby supporting more powerful annotations. We feel that the benefits of staying with a simple, uniform, executable language outweigh the loss of succinctness.

As a second example, we consider binary search for an integer x through a sorted array A of length n. Binary search, although intuitively simple, has tripped generations of programmers [2]. The function `search` starts as follows:

```
1  int search(int x, int[] A, int n)
2  //@requires n == \length(A);
3  //@requires is_sorted(A, 0, n);
4  /*@ensures (\result == -1 && !is_in(x, A, 0, n))
5  || (0 <= \result && \result < n && A[\result] == x);
6  @*/
7  {
8    int lo = 0;
9    int hi = n;
10
11   while (lo < hi)
```

The preconditions state that n is the length of A (the C0 primitive `\length` can only be used in contracts) and that this array is sorted in increasing order. We provide students with a small library of specifications functions about array segments; for example `is_sorted(A,0,n)` checks that the segment $A[0, n)$ of array A between indices 0 inclusive and n exclusive is sorted. The postcondition (typically developed in an earlier lecture about linear search) expresses that either $x \notin A[0, n)$, in which case the function returns -1, or it returns an index in the range $[0, n)$ where x occurs in the array.

At this point in the discussion of binary search, we turn to the loop. After exploring some concrete examples, it is common to draw a picture such as the following:



We are in some arbitrary iteration of the loop and we have determined that x is larger than any element in the array segment $A[0, lo)$ — abbreviated $A[0, lo) < x$ — and smaller than any element in $A[hi, n)$ — or $x < A[hi, n)$. The segment $A[lo, hi)$ is where we are still hoping to find x, if it occurs at all in A. This pictorial knowledge can be immediately converted into loop invariants that will help us write a correct body for this loop:

```
12   //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
13   //@loop_invariant gt_seg(x, A, 0, lo);
14   //@loop_invariant lt_seg(x, A, hi, n);
```

The first invariant captures the expected ordering of the indices. The last two use specification functions from our library to express that $A[0, lo) < x$ and $x < A[hi, n)$.

Our understanding of binary search tells us that we need to pick the midpoint of the interval $[lo, hi)$ and our loop invariants help us correctly update the indices `lo` and `hi`. For emphasis, we record everything we know about the code in assertions.

```
15   {
16     int mid = (lo + hi)/2;           // Tentative
17     //@assert lo <= mid && mid < hi;
18
19     if (A[mid] == x) return mid;
20     if (A[mid] < x) {
21       //@assert mid + 1 <= hi;
22       //@assert gt_seg(x, A, 0, mid+1);
23       lo = mid+1;
24     } else { //@assert A[mid] > x;
25       //@assert lt_seg(x, A, mid, n);
26       hi = mid;
27     }
28   }
```

The assertion on line 17 describes what we expect to hold of the midpoint of the interval $[lo, hi)$. But is it true? It turns out that this assertion does *not* hold for very large arrays, when `lo+hi` overflows [3]. A way to calculate the midpoint that is immune from this problem is as `mid = lo + (hi - lo)/2`: this is shown using a simple argument rooted in properties of modular arithmetic.

The correctness of this function is easy to show when it returns on line 19. But what happens when we exit the loop? The negation of the loop guard and the first loop invariant (on line 12) force us to conclude that $lo = hi$. This and the other two loop invariants entail that $x \notin A[0, lo)$ and $x \notin A[hi, n)$. There is nowhere else for x to hide in the array $A[0, n)$, and therefore the function shall return -1, thereby satisfying the postcondition clause on line 4. The annotated code is as follows:

```
29   //@assert lo == hi;
30   //@assert !is_in(x,A,0,n);
31   return -1;
32 }
```

This example shows how the contract language of C0 allows students to convert a pictorial intuition into correct code by leveraging contracts. We call this approach contract-based programming. The overall development of this example takes about an hour of lecture time, and enables students to tackle similarly tricky code on their own.

As our last example, we briefly show how contracts can be leveraged to express, and check, data structure invariants. We consider an implementation of queues based on linked lists terminated by a dummy node. In this implementation, a queue is a struct with two fields: `front` pointing to the front of the queue, and `back` pointing to the dummy node which is just one past the last element of the queue. Here is the code for dequeuing an element:

```
elem deq(queue* Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  elem s = Q->front->data;
  Q->front = Q->front->next;
  return s;
}
```

The data structure invariant for this implementation of queues is captured by the specification function `is_queue`. It checks that `front` and `back` are non-null, that the linked list starting at `front` is acyclic, and that it is connected to `back`.

5 COURSE DESIGN AND IMPLEMENTATION

Because our university offers several undergraduate systems courses that rely on C, it made sense to incorporate the gradual approach to teaching C seen in Section 3 and the emphasis on deliberate programming of Section 4 as part of our introduction to data structures course — essentially our version of CS2. It should be clear, however, that other choices are possible. For example, C0 could also be used at the core of a standalone introduction to C programming, or our progression could be embedded in a course devoted to systems programming.

Aside from using C0 and C and embracing deliberate programming, our introduction to data structure course is quite conventional: students learn about computational thinking, abstraction, asymptotic complexity analysis, traditional problem solving strategies, basic algorithms, and of course many data structures. Our emphasis on deliberate programming taught through C0's executable contracts opens new opportunities however. In particular, it guides students towards programming with intent rather than trial and error. Key to this is the explicit nature of contracts and the simplicity of C0, which allow reasoning about large parts of programs independently. It also provides a solid introduction to computer science by having students practice computational thinking, programming skills, and algorithms and data structures at the same time. We believe that each of these three component in isolation is difficult to master without at least some exposure to the other two.

Our course introduces C0 and deliberate programming right away while discussing basic concepts such as searching and sorting. This part of the course emphasizes safety and correctness through point-to reasoning as exemplified in Section 4. The course then moves on to data structures, which are introduced using C0. By this point in the course, writing contracts has become second nature for most students. This helps develop correct implementations quickly, in particular when dealing with data structures that feature complex invariants and algorithms that temporarily violate them — a prime example is insertion in an AVL tree, which violates the height invariant before restoring it by means of rotations. Around two thirds through the course, we take the training wheels off and transition from C0 to C proper. It takes several lectures to pinpoint the differences and develop techniques to mitigate the added complexity and unpredictability of C. The students master their newly acquired knowledge of C by learning about additional data structures and algorithms in this language.

Students cement the concepts they learn in lecture by means of labs and recitations led by experienced teaching assistants. The bulk of the practice happens by completing weekly homework, which take two forms in our course. On the one hand, conceptual homework focuses on giving students confidence on the more abstract aspects of the material, for example in verbalizing safety and correctness arguments through point-to reasoning. On the other hand, programming projects give them the opportunity to apply those techniques to the task of developing actual code. Example projects include image manipulations, writing a rudimentary editor, implementing Huffman compression (and submitting their solution in compressed form!), and writing a virtual machine for C0 itself. We emphasize testing as a complement to deliberate programming. Projects are mostly autograded, which provides immediate feedback to students. If this appears ambitious for a second course in computer science with mild programming prerequisites, it is! We are the first to marvel at our students' ability to master the skills to write such complex programs in the span of just a few short months. We witness this for majors and non-majors alike, and regardless of the nature of their prior programming experience.

6 DISCUSSION

The innovations discussed in the last three sections were introduced in our current curriculum eight years ago as part of a larger redesign. The introductory data structure course outlined in Section 5 replaced two core courses in our previous curriculum: the first was a conventional data structures course taught in Java and the second was a course on the C programming language which also introduced students to the UNIX environment.

We are starting to collect data in a formal manner to evaluate the effectiveness of our approach and will report results in a future paper. Students who took subsequent systems courses reported benefiting (a lot) from being trained to program deliberately — in particular they continued writing contracts in their projects. The instructors of these same courses told us that students who had learned C in our course were much better prepared than (mostly master-level) students who had learned it in conventional ways [redacted]. Our approach was first tested on our CS majors. We then opened our introductory data structures to non-majors, who started taking it in larger and larger numbers — nowadays it is consistently either the largest or the second-largest course at our university. Over the years, the changes we made to the C0 language and to our course have been rather modest.

Word about this course spread and, to our knowledge, versions have been adopted by two universities — [redacted] and [redacted]. We are taking steps, outlined in Section 7, to ease further adoption.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have outlined the approach that our university has adopted for teaching safe and correct programming in C at the undergraduate level. It revolves around a gradual exposure to C through C0, a safe and well-behaved subset of C, and an emphasis on teaching a deliberate attitude to writing programs that is supported by a layer of executable contracts in C0. We concretely deployed this approach in our introductory data structure course, which is nowadays taken by all CS majors and many non-majors in our university. Students who go on to taking our systems courses are reported to be well prepared [redacted].

Programming with contracts (especially if the contracts are executed with code) forces the students to write out the constraints and thoughts that they normally track only in their heads. Getting in the practice of thinking about memory bounds each time an array element is accessed and reasoning about the validity of memory being referred to by pointers leads to safe programming. Reasoning that loop invariants are valid and, in the end, imply post-conditions are also fundamental to writing correct code. Our programming projects show students the practical value of contracts letting them appreciate the time-saving benefits of deliberately reasoning through their code.

We are in the early stages of creating an online version of our data structure course. This online version is designed to promote student-centered, measurable and adaptable learning. Lectures are replaced by modules aimed at achieving specific learning objectives and skill sets. Each module alternates expository content and low-stakes hands-on exercises, and ends with a high-stakes assessment. Low stakes assessment allow students to practice concepts they just learned and relate them to concepts they mastered in previous modules. In this, they subsume the labs and recitations the current course delivery relies on for practice. We designed them to be dynamic, so that for example a student can summon new instances of an exercise to test his/her understanding of a concept if desired. Many also encourage experimentation by allowing students to use a new browser-based interpreter — this allows students for example to discover `NULL` as the default value of pointer types. High-stakes assessments correspond to our conceptual homework and can be generated dynamically on a per-student basis if desired. We do not plan to integrate programming projects in our online platform: students will carry them out just like they do currently.

To date, about one third of the course has been ported to our online platform. As a pilot study, we replaced an early lecture in the summer 2018 edition of our course with three modules (without high-stakes assessments). After some initial confusion about this new modality, students voiced appreciation for the finer control on their learning that this offered as opposed to the traditional lecture experience. Their performance on conceptual and programming homework that relied on this material was indistinguishable from other offerings of our course.

We expect to complete the online version of our course within two years. As new modules are completed, we will make them available to students, initially for the sole purpose of getting feedback, then as supplementary material (for some of the concepts they find most challenging) and eventually as core material.

The immediate objective of this effort is to have a baseline course for students admitted to our various masters' programs: they currently take a variant of our introductory systems course the summer before joining our university, but this is more specialized than some need. A CS2 course is more appropriate, but we do not currently have a remote modality. A second objective is to simplify sharing our experience and curriculum with other institutions, worldwide. We received a number of inquiries in recent years, but setting up our existing infrastructure was more than what most of these institutions were prepared to do. A course that is largely available through an online platform will go a long way towards eliminating this difficulty. A final objective is to incorporate the online modality within our own approach to teaching this course, something we plan to do gradually and very carefully.

REFERENCES

[1] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: the Spec# Experience. *Communcations of the ACM* 6, 54 (2011), 81–91.

[2] Jon Bentley. 2000. *Programming Pearls*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[3] Joshua Bloch. 2006. Nearly All Binary Searches and Mergesorts are Broken. Google AI Blog. https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

[4] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software Practice & Experience* (Sept. 1988), 807–820.

[5] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO'05)*. Springer LNCS 4011, 342–363.

[6] Andrew Duncan and Urs Hölzle. 1998. Adding contracts to Java with Handshake. (1998).

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

[8] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1997. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 369–388.

[9] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[10] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (second ed.). Prentice Hall.

[11] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (Oct. 1992), 40–51.

[12] Bertrand Meyer. 1992. *Eiffel the language Prentice Hall object-oriented series*. Prentice hall Upper Saddle River, NJ, USA.

[13] Bertrand Meyer. 2000. Contracts for Components-Interface Definition Languages as we know them today are doomed. *Software Development* 8, 7 (2000), 51–56.

[14] Nikola Milanovic and Miroslaw Malek. 2004. Extracting functional and non-functional contracts from Java Classes and Enterprise Java Beans. In *Proceedings of the Workshop on Architecting Dependable Systems (WADS 2004)*. Citeseer.

[15] Jan Newmarch. 1998. Adding contracts to Java. In *Technology of Object-Oriented Languages, 1998. TOOLS 27. Proceedings*. IEEE, 2–7.

[16] Jaime Niño and Fred Hosch. [n. d.]. Introducing programming with objects in CS1. In *Fifth Workshop on Pedagogies and Tools for Assimilating Object Oriented Concepts*. Citeseer.

[17] Eric Roberts. 2001. An overview of MiniJava. *ACM SIGCSE Bulletin* 33, 1 (2001), 1–5.

[18] Robert C. Seacord. 2006. *Secure Coding in C and C++*. Addison-Wesley Professional.

[19] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit Precision. In *Proceedings of the USENIX'05 Annual Technical Conference*. Anaheim, California.

[20] Wing C. Tam. 1992. Teaching Loop Invariants to Beginners by Examples. In *Proceedings of the Twenty-third SIGCSE Technical Symposium on Computer Science Education (SIGCSE '92)*. ACM, New York, NY, USA, 92–96. https://doi.org/10.1145/134510.134530