A functional paradigm using the *C* language for teaching Programming for Engineers

Víctor Theoktisto
Departamento de Computación y Tecnología de la Información
http://ldc.usb.ve/~vtheok
Universidad Simón Bolívar, Caracas, Venezuela

***Abstract*—Many engineering programs place in their curricula some courses in Computer Programming, whose content and quality are generally less rigorous than their Computer Science equivalents. Most use the *C* language as development tool, but the approach applies more effort in describing the language's peculiarities than exploiting the practical applications in engineering endeavors. The contributed programming techniques are born from the experience of teaching Functional Programming, embodied in *headers*) that enhance depth and rigorousness in the development of high quality code: *(i)* Functional emphasis from the beginning; *(ii)* Formal Specifications with exception generation; *(iii)* Early focus in recursion, particularly *tail recursion*; *(iv)* Development of Abstract Data Types using C's opaque types; *(v)* Incorporation of First Order and Higher Order Functions; *(vi)* Efficient dynamic memory management using a "garbage collector"; and *(vii)* Using and Integrated Developed Environment (IDE) with embedded debugger. The aforementioned headers are attached at the end of the article.**

***Index Terms*—Functional Programming, Teaching Strategies for Computer Programming, Recursion, Higher Order Functions.**

I. Introduction

Programming courses for [non computer science] engineering majors (and other scientific disciplines too) have an undeserved reputation of being the poor sisters in the formal teaching of Algorithmics and Computer Programming.

The proposal herein described aims to change the learning focus without changing the substance of the chosen language (C), by reinforcing an imperative paradigm with several functional programming topics rooted in calculus, providing a basic substrate for formal verification of pre- and post- conditions, invariants, and other additions.

The article is organized as follows: There is a brief review of related work in section II, and of the current thematic teaching strategy in section III. The technical description of the implementation continues in section IV, and final conclusions are located in section V, along two appendices showing practical programming examples.

II. Previous Work

Whether taught by the CS faculty or non CS faculty , their contents usually fail short of all the interesting structures and code practices that make programming fun to learn. Consolidated engineering schools have their own formed staff focused in fulfilling the minimum course requirements [1] rather than instill true analytical and coding capabilities to future engineers [2].

For the teaching of computer programming concepts, the traditional approach has been one of building upward from the concept of variable, keeping very close to the language's specifications and the imperative programming paradigm, perhaps not the best route of approaching its role in algorithm design, for which other types of tools and strategies are needed. The main antecedent on teaching proper concepts of computer programming building on a constructivist approach best suited for engineers is postulated by Ben-Ari [3], detailing the order in which must be introduced all conceptual and algorithmic elements using a programming language as tool, with a later adaptation by Chesñevar *et al* [4] and [5].

The *C* programming language [6] is still one of the most used in engineering application development just below FORTRAN, with a wide and solid installed code base, incorporated standard and specialized libraries, and a wealth of available books and online courses for supervised and unsupervised learning. Along with its derivatives, it is the language used in core *kernel* construction of most used operating systems (MSWindows, MacOS, BSD, Linux) and their applications.

In general, by purposeful construction *C* follows an imperative programming paradigm and was not designed to support a functional one. No wonder there has not been an approach for teaching algorithms incorporating functional programming concepts in that language [7], and very few using formal specifications techniques [8]

On the other hand, there is no manifested interest from other engineering schools (not closely related to Computer Science) to migrate toward a purely functional language (such as Python) in the introductory programming courses. What is formulated as an objective is to incorporate as unobtrusively as possible some strengths and concepts of the functional paradigm enabled by the most recent standard of the *C* language and several key compiler extensions to enhance the width and depth of the learning experience in the Computer Programming I [Introductory] [9] and Computer Programming II [Intermediate] [10] for engineers.

III. Thematic programmig teaching strategy

Most teaching strategies in Algorithmic Programming based on the C language rely on a book or guides that [unerringly] develop the traditional approach [11] in which the following concepts are introduced in a rigidly sequential manner, although it may be helped by visual programming aids:

a. Some detail in problem solving methodology for developing simple [noncomputer based] algorithms, as processing input values into output results variables.
b. Concept of state variables and basic data types, which are not based in data domains but in how much range can be expressed in limited storage size. Usually is accompanied by input and output formatting using “functions” and presentation templates, and students are instructed to “ignore” needed functional, with the excuse that they will be taught later, and the same goes for value and reference parameter passing (the need for the & in the *scanf(%i, &x)* instruction).
c. Control structures ***if/else/switch/case/while/do/for*** are next introduced to immediately enable the implementation of common algorithms based on bifurcation and iteration.
d. *Homogeneous Structured Data Types*, commonly named *arrays* (vectors, matrices and other tensors of 1, 2 or more dimensions), to reinforce the concept of iteration. Usually character strings and standard libraries are introduced at this level too [12].
e. *Heterogeneous Structured Data Types*, commonly called *registers* or **structs**, and struct arrays, are introduced at this level, covering the input/output of data and values to/from directory files
f. Structured methods, functions and procedures, value and reference parameter passing. Abstract Data Types and its implementation as Concrete Data Types may be aboarded at this stage, but it is not common and may be time restricted.
g. Pointers, use and abuse. There is no diving into direct memory allocation administration, smart pointers or even talk of applying a *Conservative Garbage Collector* to avoid memory allocation bugs. including cumulative memory leaks, orphan pointer catastrophes and free memory depletion.
h. Haphazard use of an *Integrated Development Environments* (IDE). Although there are several *opensource* IDEs (Code::Blocks, Eclipse, NetBeans), with nice GUIs and intrinsic debugger for visualizing dynamic execution, bug detection and analysis, there is still a segment that relies on old school command-line tools for the complete coding-compile-execute cycle.

Transversal to all this is the treatment of the algorithmic deduction-development phase, with more or less depth. Recursion is a concept that is rarely added to engineering teaching programs, in the view that is somehow "inefficient" and difficult to master. This is the main difference in approach to how programming is properly taught in Computer Science and affine disciplines. In the latter a lot of emphasis is made in formal specification design and checkup *pre-* and *post-conditions*, *invariants* and algorithmic proofs, in which the chosen implementation language is of less importance. although many modern languages (*C++, Java, Python, Ruby, Go, Rust, etc.*) present multiparadigm characteristics (*Object-oriented, Functional, Imperative*).

Themes that require going beyond what is expected at lab practice are barely touched if at all, such as efficiency and programming style. Whole swathes of engineering programs would benefit starting an approach more closely related to the field.

IV. A Proposal for Adding Functional Capabilities when Teaching C Programming

To strengthen and improve the teaching and learning experience we have incorporated the following hybrid method, which borrows concepts and structures from the functional paradigm, and is selectively implemented using proper macro constructions (or coding hacks) on top of the current *C* language specification coupled with nonstandard but widely used compiler extensions. Incorporated coding solutions not in the public domain are shown proper attribution when known.

i. Functional approach from the beginning, more oriented to state changes than to variable assignment;
ii. Simplified formal specifications enabling checking pre- and post- conditions, invariants and bounds over expressions;
iii. Early introduction of *recursion* for algorithmic solutions, with particular emphasis in *tail recursion*;
iv. Abstract Data Types (ADTs) implemented as opaque struct pointer types under an Application Programming Interface (API), stored in modular code libraries;
v. Higher Order Functions abstractions to implement for [almost] generic traditional ADTs (*Collection, Set, Stack, Queue, Sequence, single and double linked Lists*);
vi. Using a *Garbage Collector* for smart managing of dynamic memory allocation;
vii. Use an IDE with embedded debugger and dynamic memory validation tools.

Next we describe in detail the aforementioned points, with examples of their application.

IV-A. Functional approach from the beginning

The Functional Programming paradigm implements the following concepts:
Pure Functions: A function only looks at the parameters it receives, and returns a value based on a calculus associated to those values. It has no secondary effects whatsoever, such as modify input parameters (pass by value) or change variables outside the scope of the function (*state immutability*).
First Order Functions: Functions are basic dats types having its own functional operators. It implies the existence of anonymous functions and assignable functors just like any variable.

$f = (lambda(x) : x^2 - x + 1) \Longrightarrow f(2)$, evals to 3.

Recursion: The main programming technique is finding self-referencing algorithms allowing a divide-and-conquer strategy. Implies the existence of function composition, including self.

Lazy Evaluation: Function and parameter evaluation is delayed until their values are really needed. This allows definitions by comprehension and functions that remain undefined while there are no calculated values.

However, the *C* language is not a functional language at all. In fact, it is actually two different languages: proper *C* and the C PreProcessor or *CPP*. Both languages are "Turing complete" (some arcane tricks must be applied for CPP), and with the appropriate combination of the former statements and careful implementation it can be elevated to a more sophisticated language, changing neither substance nor style.

For this proposal, the introductory course initiates from the algebraic concept of "function", freshly carried from the previous basic Calculus courses, and are then translated to the language, composing functions over other functions.

The key to starting programming functions from scratch in *C* is defining a generic function f with arguments in 0 or more [numeric] domains and which returns a single [numeric] value.

$$f : \mathbb{D}_1 \times \mathbb{D}_2 \times \cdots \times \mathbb{D}_n \longrightarrow \mathbb{D}_0, \quad r = f(p_1,\ p_2,\ \ldots\ p_n)$$

being $\mathbb{D}_0$ the range or codomain of f, and arguments $p_1,\ p_2,\ \ldots\ p_n$, with $p_k \in \mathbb{D}_k$, for each domain $\mathbb{D}_k$.

For example, we denote the following *real* functions

$$\begin{aligned} sum: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ mul: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ com: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \end{aligned}$$

The algebraic specification of the above functions is represented as C code using a simply derived template, as shown next in Listing 1 (*func.c*), with $\mathbb{D}_k$ being any of *char, short, int, long, float* or *double C* numeric types.

Listing 1: **func.c**

```
D0 funcname (D1 p1, D2 p2, ... Dn pn) {
    D0 result;     /* value to calculate          */
    result = {...}; /* code using p1, p2, ... pn */
    return result;
}

double sum (double x, double y) {
    return x + y;
}
double mul (double w, double z) {
    return w * z;
}
double com (double a, double b) {
    double x = sum( mul(a,b), sum(4,-3) );
    return x;
}
```

Functions in *C* may be assigned to variables and passed as arguments to other functions using *function pointers*, a very efficient manner of implementing a simple variant of parameter's "pass by name".

IV-B. *Simplified formal specifications*

The existing *assert()* directive in *C* is crafted to define: invariants [**invariant** (*predicate*)], pre-conditions [**expects** (*predicate*)], post-conditions [**ensures** (*predicate*)] and index bounds for loop control [**bounded** (*expression*)], along with other useful macros in a *C header* called "hoare.h" (see Appendix B), independently available as source code [13].

Failure in the validation of the *predicate* in any of the directives (a violation of some formal specification) triggers an immediate exception, signaling the respective sourcecode line, and some explicative message. This only happens if the DEBUG flag is active, otherwise is ignored. It is a limited implementation, there are no quantifiers for correctness of formal specification verification as in advanced languages. Listing 2 (*hoare.c*) illustrates a very short example.

Listing 2: **hoare.c**

```
#ifdef HOARE_H
    _expects( x != 0.0 );              /* precondition  */
    _invariant( Sum == (k*k+k)/2 );    /* invariant     */
    _bounded( j );                     /* bound (j>=0)  */
    _ensures( factor >= N );           /* postcondition */
#endif
```

IV-C. *Early introduction of* recursion

Most traditional teaching strategies for non-computer scientists favor *iteration* over *recursion* on the wrong assumption that the latter may be more elegant but harder to teach, even though the work of Mirolo [14] shows hard data revealing that a functional recursive approach is transformational even when students have been exposed to imperative programming in high school. Starting from the first class, emphasis is made into formulating recursive solutions to problems, specially *tail recursion* implementations that take advantage of compiler optimizations. Listing 3 (*raise.c*) shows a classic example of an efficient algorithm to raise a real number x to an integer power n in a *tail recursive* manner, and Listing 4 (*raise_tr.asm*) shows the compiler's translation into Assembler.

Listing 3: (**raise.c**) Tail recursive function to raise a real number to an integer power

```
double rsaux_tr(double prod, double x, int n) {
    if (!n)            /* is n ==0 ?    */
        return prod;   /* end recursion  */
    if (n&1)           /* is n odd ?    */
        return rsaux_tr(x*prod, x*x, n >> 1);
    else               /* n is even      */
        return rsaux_tr(  prod, x*x, n >> 1);
}

double raise(double x, int n) {
    return rsaux_tr(1,x,n);  /* tail recursion  */
}
```

The chosen compiler, *gcc v.6.4*, under the actual standard allows important code optimizations enabling compiler options `-O2` or `-O3`. Tail recursive function calls are translated into inline loops, which means that formal parameters, instead of being pushed into the stack, are stored as local variables, and the recursive invocation (*call*) is made into a conditional jump

Listing 4: (**raise_tr.asm**) Automatic optimized Assembler translation, recursive calls made into conditional jumps

```
; Compiled by GNU C version 6.4.0 20180410
         .globl  rsaux_tr
         .type   rsaux_tr, @function
rsaux_tr:
         .startproc           ; func rsaux_tr
.L6:     testl   %edi, %edi   ; is n == 0 ?      (*)
         je      .L1          ; end recursion, jump
         testb   $1, %dil     ; is n odd ?
         je      .L5          ; n is even, jump
         mulss   %xmm1, %xmm0 ; x * prod
.L5:     sarl    %edi         ; n >> 1
         mulss   %xmm1, %xmm1 ; x * x
         jmp     .L6          ; loop back, jump (**)
.L1:     rep     ret          ; return
         .endproc
```

(*jmp*) backwards, combining an elegant programming style into a fast and memory efficient equivalent iterative loop that will not grow the stack at all, at no cost to the programmer.

IV-D. Abstract Data Types (ADTs) implemented as opaque struct pointer types

Without delving deeply into the ADT concept, the mechanism of separate interface headers (.h) and executable code (.c) allows defining in the header *struct pointer types* whose complete implementation description is stored elsewhere in the corresponding source (an *opaque* type), allowing hiding the implementation details from the interface, and enforcing good programming practices of accessing the ADT implementation (the Concrete Data Type, CDT) exclusively through the specified functions in the header, without being able to know or manipulate directly the interior data structure. In the example for a Rational ADT interface, this is shown in Listing 5 (*rational.h*).

Listing 5: **rational.h**

```
typedef struct rational_cdt* Rational;
    /*  Rational is a opaque pointer to the internal      */
    /*  concrete pointer data type rational_cdt           */

Rational consR(int num, int den);         /* constructor */
int setNumerR(Rational q, int num);       /* setter      */
int setDenomR(Rational q, int den);       /* setter      */
Rational simplifyR(int num, int den);     /* setter      */
int getNumerR(Rational q);                /* getter      */
int getDenomR(Rational q);                /* getter      */
Rational sumR(Rational x, Rational y);    /* operator    */
Rational mulR(Rational x, Rational y);    /* operator    */
```

The type is made opaque by defining in the header *rational.h* the Rational ADT as a *pointer type* to a *struct* (*rational_cdt*) without any internal details, in effect creating an opaque definition of the referenceable object. In the source implementation of the Rational ADT (in Listing 6 it is called *rational_cdt*), it is then redefined with all its attributes and access functions: constructors, selectors, mutators and destructors, created as a dynamic structure referenced by the opaque pointer. Its contents are not visible and it its only accessible by the selectors and mutators exposed in the interface header.

Listing 6: **rational.c**

```
#include "hoare.h"
#include "Rational.h"

typedef struct rational_cdt {
        /* Rational Concrete Data Type */
    int num, den;
        /* numerator and denominador pair */
} rational_cdt;

Rational consR (int num, int den) {    /* constructor */
    Rational q = (Rational)GC_MALLOC(sizeof(rational_cdt));
    _expects(den != 0);
        /* preconditionn: denominator must not be zero   */
    if (den < 0) {
        /* if denominator is negative, flip signs */
        q->num = -num;
        q->den = -den;
    } else {
        q->num =  num;
        q->den =  den;
    }
    return q;
}
int setNum (Rational q, int num) {    /* setter      */
    _expects(q != NULL);
        /* precondition: q must Exist !! */
    return (q->num = num);
}       /*  replaces numerator only  */
Rational sumaR (Rational x, Rational y) { /* operator  */
    _expects(x != NULL and y != NULL);
        /* precondition: x,y exist!! */
    return consR((x->num * y->den + x->den * y->num),
              (x->den * y->den));
}   /*  returns a new rational as the sum of *
     *  two other Rationals                  */
```

If we changed the implementation of *rational_cdt* to **int** values[2], and adapted all related functions defined in *rational.c* to this representation, the API interface to Rational (*rational.h*) would not change and existing code would keep working without modification.

IV-E. Higher Order Functions

In Functional Programming *Higher Order Functions* or HOFs are used to create and manipulate generic functions that operate on other function types. A catalogue of the most useful ones follow:

map: Automorphism, a function $f : \alpha \rightarrow \beta$ is applied nondestructively transforming each item of a sequence S of elements in α, producing a new sequence of elements in β in the same relative order.

$$map : (\alpha \rightarrow \beta) \times Seq{<}\alpha{>} \longrightarrow Seq{<}\beta{>}$$

filter: Automorphism, a predicate $p : \alpha \rightarrow \mathbb{B}$ (*boolean*) extracts in nondestructive fashion all elements α from S that comply with the predicate p, obtaining a new sequence S' in the same relative order, with $\#(S') \leq \#(S)$.

$$filter : (\alpha \rightarrow bool) \times Seq{<}\alpha{>} \longrightarrow Seq{<}\alpha{>}$$

reduce or fold-right: Accumulator, where the elements of a monoidal sequence S are combined in a resulting value $r \in \beta$ when applying some right-associative binary composition operator $Op : \alpha \times \beta \rightarrow \beta$ (having neutral element or identity $\epsilon_0 \in \beta$) to the whole sequence. The left-associative variant ***fold-left*** is also implemented.

$$reduce : (\alpha \times \beta \rightarrow \beta) \times \beta \times Seq{<}\alpha{>} \longrightarrow \beta$$

compose: Algebraic Function Composition. In functional theory the composition operator ($\circ$) is a ***fold-left*** over functions operating in algebraic chainable (compatibles) types.

$$\begin{aligned} h(x) &= (f \circ g)(x) \equiv f(g(x)) \\ g : \alpha \rightarrow \gamma \;\wedge\; f &: \gamma \rightarrow \beta \implies h : \alpha \rightarrow \beta \end{aligned}$$

It is possible to create a generic *compose* function making a cumbersome macro use in the style of *C++*, but if we limit functions to the integer or real domains ($\Omega = i|l|f|d$) with up to 3 arguments (extensible to more), we can define the following generic *pointers to function* patterns:

$$\begin{aligned}
\Omega\_compose\_f\_gx(f,g)(x) &\equiv f(g(x)) \\
\Omega\_compose\_f\_gxy(f,g)(x,y) &\equiv f(g(x,y)) \\
\Omega\_compose\_f\_gx\_hx(f,g,h)(x) &\equiv f(g(x),h(x)) \\
\Omega\_compose\_f\_gx\_hy(f,g,h)(x,y) &\equiv f(g(x),h(y)) \\
\Omega\_compose\_f\_gxyz(f,g)(x,y,z) &\equiv f(g(x,y,z)) \\
\Omega\_compose\_f\_gx\_hx\_wx(f,g,h,w)(x) &\equiv f(g(x),h(x),w(x)) \\
\Omega\_compose\_f\_gx\_hy\_wz(f,g,h,w)(x,y,z) &\equiv f(g(x),h(y),w(z))
\end{aligned}$$

We can also define HOFs specific to a particular type, as shown for the List ADT in Listing 7 (*list.h*). Here, higher order functions create new lists from others without modifying or destroying them, as can be appreciated in Listing 8 (*list.c*).

Listing 7: **list.h**

```c
    /* List.h */
typedef struct ListCDT_t* List_t;
    /* List_t pointer to opaque type ListCDT_t  */
typedef int Value_t, Elem_t;

List_t consEmptyEL ();
    /* returns an empty List  */
List_t consEL(Elem_t e, List_t s);
    /* cons of element e and tailing with List s */
Elem_t firstEL(List_t s);
    /* first Elem e from List s */
List_t restEL(List_t s);
    /* rest of List s */
int    insertEL(List_t s, Elem_t e, int pos);
    /* insert Elem e at position pos in List s */
List_t mapEL(Elem_t (*fun) (Elem_t), List_t L);
    /* returns a transformed copy of List s */
List_t filterEL(bool (*pred) (Elem_t), List_t L);
    /* returns a filtered copy of List s */
Value_t reduceVEL(Value_t (*opr)(Elem_t, Value_t),
                  Value_t eps0, List_t L);
    /* returns fold applying operator opr over List s */
```

IV-E1. Anonymous Functions (lambda): Anonymous functions are a core part of the building blocks of functional programming. The *C* preprocessor allows working with argument lists of indeterminate length. A very useful macro was

Listing 8: **list.c**

```c
typedef struct ListCDT_t {
    Elem_t  info;
    List_t next;
} ListCDT_t;   /* ListCDT_t Concrete Data Type*/

List_t consEL(Elem_t e, List_t S) {
    List_t  node = GC_MALLOC(sizeof ListCDT_t);
    node->info = e;
    node->next = S;
    return nodo;
}   /*  Returns a List with just Elem e  */
List_t mapEL(Elem_t (*func)(Elem_t), List_t L) {
    if ( L ) return consEL((*func)(firstEL(L)),
                        mapEL(func, restoEL(L)));
    return NULL;
}   /*  Recursive map HOF implementation (new List) */
List_t filterEL(bool (*pred)(Elem_t), List_t L) {
    if ( !L ) return NULL;
    if ( (*pred)(firstEL(L)) )
        return consEL(firstEL(L),
                      filterEL(pred, restEL(L)));
    return filterEL(pred, restEL(L));
}   /*  Recursive filter HOF implementation (new List) */
Value_t reduceVEL(Value_t (*opr)(Elem_t, Value_t),
                  Value_t eps0, List_t L) {
    if ( !L ) return ident;
    return reduceVEL(opr, (*opr)(eps0,
            firstEL(L)), restEL(L));
}   /* Recursive fold HOF applying "opr" on List s */

bool isOdd(Elem x) { return ODD(x); }
Elem_t square(Elem x)  { return SQR(x); }
Elem_t sum(Elem_t res, Value_t val)  { return res + val; }

int main() {    /*  composition of LAMBDA functions     */
    List_t a = consEmptyEL();
    List_t b, c;
    Value_t v, w, z;
    a = consEL(1,consEL(2,consEL(3,consEL(4, a))));
     /*  a is {1 --> 2 --> 3 -->  4}                    */
    b = mapEL(sqr, a);
     /*  maps sqr() to a, = {1 --> 4 --> 9 --> 16}      */
    c = filterEL(isOdd, b);
     /*  filters b with isOdd(), = {1 --> 9}            */
    v = reduceVEL(sum, 0, c);
     /*  sum is c folding "+" returns = 10              */
    w = reduceVEL(sum, 0, filterEL(isOdd, mapEL(cuad, a)));
     /*  same as above, but composing filter and map    */
    z = mapfilterreduce(square, isOdd, sum, a);
     /*  same, but composing the three HOFs             */
}
```

devised allowing the definition and use of *lambda* functions in the code, as shown below The scheme also allows for the algebraic composition of *lambda* functions:

$$i\_compose\_f\_gx = lambda(f,g,x)\{f(g(x))\},$$
$$\text{with } f = lambda(x)\{...\},\; g = lambda(x)\{...\},\; x \in \mathbb{Z}$$

It is a sophisticated technique that is incorporated optionally in the introductory course for three main reasons

(i) There is no requested need to teach the use of *lambda* functions at this level in the course syllabus;
(ii) It is a feature outside the standard, even though modern compilers such as *gcc* and *clang* allow the *nesting* of function definitions, needed for its correct execution; and
(iii) A wrong function definition may give rise to compiling or execution errors quite difficult to trace and amend.

Listing 9: **lambda.c**

```c
#include <stdio.h>

        /* LAMBDA expression returning a value */
#define lambda(FUNTYPE, PARAMS, ...)  ({FUNTYPE lambda  PARAMS {__VA_ARGS__;}  lambda;})

       /*  Defining a type for declaring the function   */
typedef int (*IFPTR_t)(int);

int (*i_compose_f_gx) (IFPTR_t, IFPTR_t, int);  /*  compose entire functions  */

float (*f_compose_f_gx) (float (*)(float), float (*)(float), float);
    /*  Or making the definition inline to compose other float functions  */

int main()
{
    int (*r)(int)       = lambda(int, (int x), return x+1);
    int (*s)(int, int) = lambda(int, (int x, int y), return x/y);

    printf("result  = %i\n", s(r(7), r(3)));

        /*  Defining function composition  (f.g)(x)  = f(g(x)) [int domain]  */
    i_compose_f_gx = lambda(int, (IFPTR_t f, IFPTR_t g, int x ), return f(g(x)));

    printf("f(g(x)) = %i (ints)\n",
           i_compose_f_gx(lambda(int, (int x), return x+x), lambda(int, (int x), return x*x), 6));

        /*  Defining function composition  (f.g)(x)  = f(g(x)) [float domain]  */
    f_compose_f_gx = lambda(float, (float (*f)(float), float (*g)(float), float x), return f(g(x)));

    printf("f(g(x)) = %f (floats)\n",
           f_compose_f_gx(lambda(float, (float x), return x*x),
                          lambda(float, (float x), return x+x), 4.0f));

    return lambda(int, (int x), return x )(0);  /*  returns the IDENTITY function applied to 0 */
}
```

IV-F. *Using Boehm's* Garbage Collector *for smart managing of dynamic memory allocation*

A common recurrent situation when working with dynamic memory allocation is the problem of dangling pointers that no longer reference valid structures, with the consequent reduction in available memory ("leakage") and potential catastrophic effects when trying to dereference and invalid pointer.

The chosen solution is using the available opensource (*Garbage Collector*), particularly Boehm's GC [15]. The *GC* substitutes all calls in *C* that create, free and manipulate dynamic memory (*malloc(), calloc(), realloc()*) for a more efficient set (*GC_MALLOC_ATOMIC(), GC_MALLOC(), GC_REALLOC()*) which use *smart pointers* to keep track of memory use. There is no need of using the *free()* function any more, although it can be called to provoke a memory scrambling (*GC_FREE()*) because all memory is now handled automatically under demand, the way is done in other common functional languages (Python, Scheme, Haskell, ML, and also Java).

Together with the byte-level memory tracker *Valgrind* from Nethercote and Seward [16] they do the job of diagnosing memory execution state to monitor performance and avoid damaging program disruptions. This allows building relative big applications having an implicit but efficient memory management *without* the programmer being overtly aware of it, and adjusted to the latest *C* standard and CERT security coding practices [17].

IV-G. *Use an IDE with embedded debugger and dynamic memory validation tools.*

Multiplatform IDEs (Integrated Development Environments) such as *Code::Blocks* [18] or *Eclipse* [19] are essential tools for correct tracing and tracking of the interactive programming process. They host real time embedded source code debuggers that allow step-by-step code execution, examining variables values and memory state at every instant. They enable an integrated vision of program execution at levels thought unthinkable in former approaches in the teaching of Computer Programming for engineers, where active error hunting was touched briefly if at all, other than blindly using interspersed *printf* statements around suspected areas.

V. CONCLUSIONS

One key objection that can be raised for this course strategy is why going to all this work with *C*, when it would be so much easier just going along directly using a language with a truly functional paradigm. The answer is that choosing the most adequate programming tool that future engineers should learn requires joint agreement of all engineering departments, who are reluctant to change the base language in which much of their academic and professional practice rely on. Segmenting student population by engineering discipline so they could learn separate programming languages does not make efficient use of the teaching resource, since each school would choose.

V-A. Impact and Consequences

Applying this approach en the current teaching of Computer Programming for Engineers, split into an introductory course and an advanced one, has had qualitative impact and consequences.

V-A1. ***Qualitative results:*** This is the third consecutive period running with the approach, and although is evidently clear that there is not enough data in the time period for making and affirmation backed by solid statistics, the number of the topics covered in the courses have increased by 20%, and allowed to cover some of them to greater depth. The problems and lab projects have increased in complexity, and so have the proposed solutions, highlighting greater comprehension by students of the functional algorithmic process, much more appropriate for engineering disciplines that already work based on the aggregation of functional components.

Results reveal a noticeable jump in quality for homework and lab coding assignments, and incremented legibility. It allows painless transference of Calculus (differential and integral) competences, since this concepts are directly expressible using the constructions already described.

The topics covered in the Introductory course follow this learning schedule: functions, formal specifications, conditional execution, recursion, input/output, parameter passing, iteration, algorithms on data structures, ADTs,

Looking into the future, recent powerful languages such as Julia, Python, Rust and Erlang allow greater efficiency as hybrid languages, and merit further research to determine whether they are well suited as the computer language of choice for programming courses in non-computer science majors.

In the Appendix we show a simple example of a code for finding the roots of a 2nd degree polynomial (Listing 10), and how it looks when includes the following “hoare.h” header (Listing 10) to facilitate program correctness.

References

[1] I. Milne and G. Rowe, “Difficulties in Learning and Teaching Programming – Views of Students and Tutors,” *Education and Information Technologies*, vol. 7, no. 1, pp. 55–66, 2002.

[2] A. Robins, J. Rountree, and N. Rountree, “Learning and Teaching Programming: A Review and Discussion,” *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[3] M. Ben-Ari, “Constructivism in Computer Science Education ,” *Journal of Computers in Mathematics and Science Teaching*, vol. 20, no. 1, pp. 45–73, 2001.

[4] C. I. Chesñevar, M. P. González, A. G. Maguitman, and L. Cobo, “Teaching Fundamentals of Computing Theory: a Constructivist Approach,” *Journal of Computer Science & Technology*, vol. 4, no. 2, pp. 91–97, 2004.

[5] P. H. Hartel and H. L. Muller, *Functional C*. Harlow, UK: Addison Wesley Longman, 1997. [Online]. Available: http://eprints.eemcs.utwente.nl/1077/

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Upper Saddle River (N.J.): Prentice Hall, 1988.

[7] S. Peyton-Jones and D. Lester, *Implementing functional languages: a tutorial*. Prentice Hall, 1992.

[8] D. L. Chaudhari and O. Damani, “Introducing formal methods via program derivation,” in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE ’15. New York, NY, USA: ACM, 2015, pp. 266–271. [Online]. Available: http://doi.acm.org/10.1145/2729094.2742628

[9] V. S. Theoktisto. (2016) *CI-2125 Computación I. Curso introductorio. Programación para Ingenieros*. Universidad Simon Bolivar. [Online]. Available: http://ldc.usb.ve/~vtheok/cursos/ci2125/aj14

[10] ——. (2016) *CI-2126 Computación II. Curso avanzado. Programación para Ingenieros*. Universidad Simon Bolivar. [Online]. Available: http://ldc.usb.ve/~vtheok/cursos/ci2126/sd14

[11] D. Marchena Parreira, A. Petersen, and M. Craig, “Pcrs-c: Helping students learn c,” in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE ’15. New York, NY, USA: ACM, 2015, pp. 347–347. [Online]. Available: http://doi.acm.org/10.1145/2729094.2754852

[12] P. J. Plauger, *The Standard C Library*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1991.

[13] V. S. Theoktisto. (2016) *Encabezado (header) para especificaciones formales en C “hoare.h”. Curso avanzado. Programación para Ingenieros*. Universidad Simon Bolivar. [Online]. Available: http://ldc.usb.ve/~vtheok/cursos/ci2126/aj14/hoare.h

[14] C. Mirolo, “Is iteration really easier to master than recursion: An investigation in a functional-first cs1 context,” in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE ’11. New York, NY, USA: ACM, 2011, pp. 362–362. [Online]. Available: http://doi.acm.org/10.1145/1999747.1999876

[15] H.-J. Boehm, “Bounding Space Usage of Conservative Garbage Collectors,” *SIGPLAN Notices*, vol. 37, no. 1, pp. 93–100, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/565816.503282

[16] N. Nethercote and J. Seward, “How to Shadow Every Byte of Memory Used by a Program,” in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE ’07. New York, NY, USA: ACM, 2007, pp. 65–74.

[17] S. E. I. SEI, *SEI CERT C Coding Standard Rules for Developing Safe, Reliable, and Secure Systems*. Carnegie Mellon University, 2016.

[18] Code::Blocks. (2016) *Code::Blocks: the free Open Source (GPLv3) cross-platform C, C++ and Fortran IDE*. The CodeBlocks Team. [Online]. Available: http://www.codeblocks.org

[19] Eclipse::Neon. (2016) *The Eclipse Top-Level Project - an open source, robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools and rich client application*. The Eclipse Foundation. [Online]. Available: http://www.eclipse.org/ide/

Listing 10: **rootpoly2.c:** Example of a polynomial root-finding program using the “hoare.h” header

```c
/*
 * DESCRIPTION: calculate roots of a second degree polinomial   *
 * p(x) = Ax^2 + Bx + C = 0                                     */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "hoare.h"
const double eps = 0.000001

int main()  {
        /* Good practice: initializing variables to predefined      +
         * values, even though they will be substituted by others  */
        /*  Input  */
  double a = 1.0, b = 0.0, c = 1.0, d = 0.0, r = 0.0;

  double root1 = 0.0,  root2 = 0.0;  /*  real roots        */

  double preal = 0.0,  pimag = 0.0;  /*  complex roots     */

  bool isComplex = false;     /*  true if roots are complex */

  printf("\Obtain 2nd degree polynomial roots"
         "p(x) = Ax^2 + Bx + C = 0");
  printf("\n\Give value of cuadratic term 'A' (real): ");
  scanf ("%lf", &a);
  printf("\nGive value of linear term 'B' (real): ");
  scanf ("%lf", &b);
  printf("\nGiva value of independent term 'C' (real): ");
  scanf ("%lf", &c);
  printf("\nLoaded terms A = %f, B = %f, C = %f\n\n", a, b, c);

  _expects( a != 0.0);     /*  modified precondition   */
  d = b*b - 4*a*c;         /*  discriminant            */
  isComplex = (d < 0.0);
        /*  If d is negative, we get complex roots     */
  printf("\nDiscriminant D = %lf;  is it Complex ? = %d\n\n", d, isComplex);

        /*  obtain square root of the absolute value of the discriminant d  */
  r = sqrt(ABS(d));

  if (isComplex) {
     preal = -b/(2*a);
     pimag =  r/(2*a);
     printf("\nComplex roots %lf +%lfi and %lf %lfi\n\n",
            preal, pimag, preal, -pimag);
  } else {
     root1 = (-b + r)/(2*a);
     root2 = (-b - r)/(2*a);
     printf("\nreal roots %lf y  %lf\n\n", raiz1, raiz2);
  }
            /*  modified postcondition          */
  _ensures( ((ABS(root1*root2 - c/a) < eps) and
             (ABS(root1+root2 + b/a) < eps)) or
            ((ABS(preal*preal+pimag*pimag - c/a < eps)) and
             (ABS(preal+preal + b/a) < eps)) );
  return 0;
} /* end main */
```

Listing 11: **hoare.h:** A header incorporating several useful macro tips to aid in formal specifications design

```c
/* Code distributed under the GNU  LGPL 3.0 License    *
 * Version 1.07 08/11/2016, Author: Victor Theoktisto  */
#pragma once
#ifndef _HOARE_H_INCLUDED_
#define _HOARE_H_INCLUDED_
#include <stdlib.h>
#include <stdio.h>
/* CompilerAssert(exp) is designed to provide error checking at compile-time for assumptions
 * made by the programmer at design-time and yet does not produce any run-time code.
 * Example: if (CompilerAssert(sizeof(int)==4)) ...                                    */
#define CompilerAssert(Predicate) extern char _CompilerAssert[(Predicate)?1:-1]

enum bool {FALSE=0, TRUE=~0};   /* Useful definitions */
#define and &&
#define or  ||
#define not !
#define xor ^
#define GLUE(a,b)   a##b
#define XPREFIX(s)  s
#define PREFIX(a,b) XPREFIX(a)b
#define ABS(a) ({__auto_type __a = (a); __a < 0 ? -__a : __a;})
     /* Compiler warns when the types of x and y are not compatible  */
#define MAX(x,y) ({__auto_type __x=(x); __auto_type __y=(y); (void)(&__x==&__y); __x>__y? __x: __y;})
#define MIN(x,y) ({__auto_type __x=(x); __auto_type __y=(y); (void)(&__x==&__y); __x<__y? __x: __y;})
#define ODD(n)    ((n)&1)
#define EVEN(n)  (!((n)&1))
#define SWAP(__A,__B) do{ __auto_type __T=(__A); (__A)=(__B); (__B)=__T;}while(0) /*  failsafe SWAP  */
#define MEMSWAP(A,B) \
        do{ unsigned char __C[sizeof(A) == sizeof(B) ? (signed)sizeof(A) : -1]; \
           memcpy(__C,&B,sizeof(A));  memcpy(&B,&A,sizeof(A));  memcpy(&A,__C,sizeof(A)); \
        }while(0)   /*  failsafe Memory SWAP for any size  */
#define ISPOWEROF2(x)  (!((x)&((x)-1)))                     /*--- Is a number a power of two ---*/
#define NUMCELLS(_arraytype)  (sizeof(_arraytype)/sizeof(*_arraytype))     /* _NUMCELLS() macro */
#define ONESCOMPLEMENT(x)  ((x)^(~0)))                  /* One's complement negation as a macro */
#define TWOSCOMPLEMENT(x)  (((x)^(~0))+1)                /* Two's complement negation as a macro */
#define SQUARE(x)  (__auto_type __x=(x); (__x)*(__x))   /* Safe SQUARE() macro, final form */
#define CUBE(x)  (__auto_type __x=(x); (__x)*(__x)*(__x))   /* Safe CUBE() macro, final form */
#define ALLONES ~0               /* fills an integer value with ONES independent of type size */
#if UNICODE                           /* Distinguishing between ascii chars and wchar chars:  */
    #define dchar wchar_t
    #define TEXT(s) L##s              /* word chars (unicode)   */
#else
    #define dchar char
    #define TEXT(s) s                 /* byte chars (ascii)     */
#endif
    /*  Macros for memory allocation using Boehm's GC              *
     *  Redefine "malloc()" using GC_MALLOC, "free()" is optional  */
#define NEW(__PointerVar) (__PointerVar=GC_MALLOC(sizeof(*__PointerVar)))
#define NEWARRAY(__Dim,__DataType) (GC_MALLOC(__Dim*sizeof(__DataType)))
#define DELETE(__PointerVar) (NULL)
       /*  General assertion (Predicate)      */
#define _assert(Predicate)      __do_assert("Assertion does not hold for",      Predicate)
       /*  Precondition (Predicate)           */
#define _expects(Predicate)     __do_assert("Precondition does not hold for",   Predicate)
       /*  Postcondition (Predicate)          */
#define _ensures(Predicate)     __do_assert("Postcondition does not hold for",  Predicate)
       /*  Invariant (Predicate)              */
#define _invariant(Predicate)   __do_assert("Invariant does not hold for",      Predicate)
       /*  Precondition (integer expression)  */
#define _bounded(IntExpression) __do_assert("Bound does not hold for", (IntExpression)>=0)
#ifdef NDEBUG
    #define __do_assert(Message,Predicate)  ((void)0)  /* Null statement if not debugging */
#else /* We are debugging !!! */
    #define __do_assert(Message,Predicate)  \
           ((void)((Predicate)?0:__my_assert(Message,#Predicate,__FILE__,__LINE__)))
    #define __my_assert(Message,Predicate,File,Line) \
           ((void)printf (">>> At %s:%u:  %s '%s'\n<<< Assertion failed. " \
             "Execution will stop now.\n",File,Line,Message,Predicate),exit(1),0)
#endif  /* NDEBUG */
#endif  /* ifndef _HOARE_H_INCLUDED_ */
```