# C in Education and Software Engineering

R.P.Mody

Department of Computer Science,
University of Poona,
Poona 411007,
India.

## 1. Introduction

C has come to stay in the computing world. This article reports the impact of this on students and teachers of computer science as seen by a CS teacher.

Section 2 is a short history of C in our department.

Section 3 is a small set of variations on the theme of what is 'advanced' and why C is advanced.

When C is used as a medium of instruction irrelevant issues become overwhelmingly large in number and significance. When the irrelevant becomes significant, the essentials become obscured and incomprehensible. Section 4 is an assortment of programming trivia which make reaching programming via C tortuous.

After an introduction to programming via C, there are wide-ranging consequences. Section 5 is a catalogue of such consequences. It is undoubtedly lopsided in that my concrete experience is largely negative. Some brief sorties are made into 'might have beens' but they are generally not substantiated by my own experience. They are however supported by appropriate citations from the literature. I hope that the negative content will be of value to the CS teaching, practicing and research community by helping to prevent the repetition of mistakes.

The conclusion is non-constructive, emphasizing possible errors of interpretation rather than suggesting solutions.

## 2. History

In 1985 UNIX entered our department. With it came C. The author was among the first batch of students which used C. In 1987 when I joined as a junior faculty, I taught C to a batch of first-year students who had had one course in programming. I again taught the same course in 1988. As part of the continuing attempt at teaching programming in the currently best possible way, C became the medium of instruction in the introductory programming course in 1990. This was of course only one step in the innovative process, but the rest of the picture is not directly relevant to this article and so is not discussed further.

## 3. Basic pedagogic issues

It is generally accepted that advanced concepts should follow basic ones. Although advanced connotes 'hard-to-understand', there is an important equivalent meaning : 'A concept is advanced when the tools for its assimilation and use have not been previously developed'. For example if a Neanderthal man cuts an apple cleanly into two equal halves with a sharpened stone and long nails, he performs a feat, whereas when I do it with a kitchen knife I do not. Whether it is producing one line of ununderstandable code, or thousands of lines of brittle unmodifiable code, the C programmer habitually performs feats. Even if it is deemed necessary to train computer scientists to become trapeze artists one wonders how early such

dangerous activities should be allowed.

## 3.1 C is Meta-operationally defined

Concepts in programming that can be assimilated independent of machine actions are often the easiest to understand. Such concepts are called non-operational. For example the right-hand side of assignment statements correspond to typical mathematical expressions and can be more easily understood than concepts like loops and gotos and interrupts. C can be properly understood only when we understand the insides of the C compiler. In other words, C is not just operationally defined; it is meta-operationally defined! I give a couple of proofs

1. One of the problems with teaching programming using compiled languages is that we must teach concepts like compile-time constant as against run-time variable. If that were all, we could perhaps survive, but C has a whole battery of 'constantnesses'.

   `#define N sizeof(int)`
   is allowed but
   `#if (sizeof(int) == N)`
   is not.

   The reason is that the compiler is multipass and can afford to postpone evaluation of arguments of `#define` but not `#if`. How can one explain this to a student who knows nothing of the compilation process? Does one say that although `sizeof(int)` is a constant expression, the kind of constant required by `#if` is 'more constant'?

2. Pointers
   C is famous for being pointer oriented. It starts off with mathematical elegance by postulating 2 operations '*' and '&'. which are inverses of each other. That means that for any 'l-value' `x`, `*&x` ≡ `x` and for any 'pointer-value' p, `&*p` ≡ `p`

   But consider the simple assignment:
   `x = y;`
   To make it consistent with the treatment of locations as first-class values, it should be written as

   i. `&x = y;`
   or
   ii. `x = *y;`

   depending on whether an undecorated variable means its value or address. The solution is kludged by defining the C semantics as "other than on the lhs of an assignment, a variable denotes its value." This kludge does not quite work, because, in addition to 'the lhs of an assignment' we must add 'argument to &'.

   Now the soup is getting hot because '&' is no more an operation at all but 'a directive to the compiler' to choose the address attribute of an object rather than its value.*

3. Expressions with side-effects
   Many beginning C programmers use `x = x++;` instead of `x++;`. In addition to being unnecessarily long, the first one is **wrong**; it may leave x unchanged. The expression `x = x++ - 1` has 3 possible interpretations. † It may increment x, it may decrement x and it may leave x unchanged. What actually happens depends on compilation strategy.

How does one handle such subtle semantic questions?

---

* Notice how, rather than the semantics of the language defining the compiler, the language is defined in terms of nefarious conversations between programmer and compiler. Those not prepared to believe that clean pointer semantics are possible are invited to study Bliss [Wulf] or Algol-68 [McG].

† known to the author; there may very well be more

1. The very thought of a systematic, (mathematical ?) semantics for C is hair-raising. Even if it were attempted, it would obscure rather than highlight the issues. The bigger problem however is that the typical C programmer thereby concludes that systematic semantics is only suitable for toy languages rather than concluding that C has no systematic semantics.
2. An operational semantics (language defined by compiler rather than the other way round) is the only option but what if students don't know the insides of compilers, machines, assemblers etc? What normally happens is :
3. Students don't understand at all!

The fact that C abounds with advanced features that are hard to understand leaves open a number of options for the teacher – all equally unpleasant.

1. The teacher teaches the advanced features as the need arises. Since the need arises right from day one, (you can't perform input unless you know about pointers!) the teacher is branded as being very difficult.
2. The teacher give some cooked constructs and forbids use of anything not done in class. So much for the pedagogic ideals of fostering creativity and independent study.
3. The teacher teaches some isolated, trivialized subset and lets the students figure out subtleties and difficulties on their own, whenever they see a feature they don't understand.

One of the most fundamental desires of a teacher is to increase the depth of understanding of the students and to inculcate a flexible and widely applicable body of knowledge, rather than just accumulating accretions of soulless information into the students' heads as though they were dumb data-bases. Putting it from the students' point of view, a good teacher makes difficult concepts seem easy.

Assuming that we agree with this view, one of the purposes of this paper is to indicate how C makes the teaching of programming impossible.

### 3.2 C is for the Gods

Ritchie is the inventor of C and with Thomson the co-inventor of Unix. In [Shoo] they are called 'super-programmers' and their rate of programming is described as a 'significant feat.'

The preface to [Ker,Rit] says,

*C wears well as one's experience with it grows.*

This paper should show that if we make fresh students get 'experienced' with C, they might well wear out before they get experienced.

## 4. Pedagogic Problems

This section is a small sample of the problems that make learning C at the introductory level a harrowing experience – both for student and teacher.

### 4.1 Operational Definitions

Very low-level explanations are all that's possible in many cases
eg: `return &x;  /* x is a local */`
is invalid. Yet this can only be explained in terms of the machine stack.

### 4.2 Wrong programs that are right

The number of programs that should not work but still do are legion.

eg.

```
int gcd(x, y)
int  x, y ;  /* x >= y >= 1 */
{
  if (x == y)
    return x;
  else  /* NO return */  gcd(y, x%y);
}
```

This program is invalid because in one case it returns a value and in another it does not. However it will compile and run silently on most systems and actually work! The explanation as usual is given in terms of register-allocation done by the compiler – in an introductory programming course !! Further the incorrect version is potentially more efficient and this fact highlights a frightening definition of 'An expert C programmer' – he/she is one who can hoodwink the C compiler!

## 4.3 Puns

Puns occur when similar constructs have dissimilar meanings. Puns make learning extremely difficult for beginners. A few examples are here given:

1. `*x = 0`
   assigns 0 to the object pointed to by x, whereas
   `int *x = 0`
   assigns the NULL pointer to x

2. Similarly
   `x = y = 0;`
   makes x and y equal to 0 but a declaration,
   `int x = y = 0;`
   makes x and y equal to 0 but only declares x.

3. The ',' is used in a large number of places such as macro-definition and call, function definition and call, as an operation, separator in initializations, separator in declarations. Often these uses are mutually inconsistent. Eg.

   `x=1, y=x+1;`
   indicates sequencing, whereas
   `f(x=1, y=x+1);`

   explicitly does not when `f` is a function. If `f` is a macro, it all depends on the macro.

4. Consider the famous, but horrendous C idiom from no less than the C text book [Ker,Rit], for copying one character array (string) to another:

   ```
   while (*s++ = *t++);
   ```

   Firstly, it embodies the pun 'arrays are pointers'. This pun is pernicious because it throws cold water on attempts to understand data-strucures as *values* rather than storage structures. Secondly, it embodies the pun 'expressions denote values and effects' and it requires a great deal of operational reasoning to convince oneself that the effect is unambiguous. Then there is the triple pun that :

   a. 0 is an integer.

   b. 0 is false

   c. 0 is the string-terminator.

   In fact the pun on 0 is quadruple – it is also the NULL pointer.

   Surely, even an assembly language coder cannot do worse in producing obfuscating code, though he/she may produce more efficient code. C programmers often use this example to tout the stunning combination of efficiency and cogency in C. An answer to that is to consider the APL equivalent: `s ← t`. In addition to being shorter, and much simpler, it allows the compiler to produce optimal

code without doing hairy optimization – the code is naturally optimizable.

5. Since good old C is not confusing enough, the ANSI extension adds its share in the form of a new keyword `void`. `void` is

   a. the type of a function that is not a function.

   b. the type of a pointer that is semi-valid.

   c. the type of the parameters of a function, that takes no parameters.

   And why, pray, will `int f();` not do rather than `int f(void);` ?
   Because the first one, by specifying nothing allows anything, whereas the second by specifying "nothing" (void) allows only nothing. All in the name of upward-compatibility!

6. Since C does not make life difficult enough with puns it provides meta-puns in the form of language support for defining puns – multiple name-spaces.

   C uses distinct name-spaces for its identifiers. Structure tags, for instance, live in a separate world from ordinary variables. The following declaration declares a node-type `struct list` and a pointer type `list` with a clever use of forward referencing.

```
typedef struct list *list;
struct list {
  int fld;
  list next;
};
```

   It may be confusing which `list` refers to the node and which to the pointer, but we can perhaps survive so far.

   C goes further and allows struct fields still another world to live in. Therefore the following declaration is valid.

```
typedef struct list *list;
struct list {
  int fld;
  list list;
};
```

   And finally, for good measure, C adds one more name-space, that of goto-labels. The following function is valid. It has however become confusing enough to bewilder most compilers, leave aside human beings.

```
int length(list l)
{
  int i = 0;
  list: if (l) {
        i++;
        l = l->list;
        goto list;
      }
      else return i;
}
```

   There is even more to name-spaces! See [Harb,Stl].

## 4.4 Nups

If a pun occurs when ‘Similar syntactic constructs have dissimilar meanings’, then there is the converse concept ‘dissimilar (or worse, inconsistent) syntactic constructs having similar meanings’. C abounds in

both vulgar puns and ‘nups’.

1. Numeric constants are by default decimal, a preceding 0 in a numeric constant makes it octal. Character constants however are by default octal, so `'\014'` is the same as `'\14'`.
2. The address of a variable is obtained with an ‘&’ but for arrays and functions ‘&’ is not used. The problem is compounded by implementations that patch such problems in ad-hoc ways.

## 4.5 Obfuscatory Properties

1. How about this for obfuscation: Although `char` denotes the set of characters, a ***particular*** character is not a `char`, but an `int` (Or is it `unsigned` ?) i.e.

   ```
   char x;
   ```

   declares a character variable but the character constant `'A'` does not denote a character but an int (or unsigned depending on implementation). Then how does `x='A'` work? Because of casting!
2. If this is not confusing enough, consider: `getchar()`, which means ‘get a character’, actually returns an integer but does so for a different reason from the one above. The reason is that `getchar()` which must return characters must also be able to return an invalid character to signal end-of-file, therefore the type of `getchar` must be one which includes `char` and also something else. If this is confusing, Sorry! Such is the contents of the C bible [Ker,Rit]. For those who have been nurtured at the breast of C that ask, ‘How else can input be done?’, the answer is that it is so done only to use the following extremely opaque idiom:

   ```
   while ((c=getchar()) != EOF){
         ...
   }
   ```

   rather than the more perspicuous

   ```
   While ~Eof(input)
   begin
     c := Getchar(input);
     ...
   end
   ```

   What about efficiency? – the C-ers clamour. Again the answer is that the second surprisingly is *more* efficient. Assuming a language like Pascal where characters are first class, the character returned by `getchar` would be directly assigned to `c` whereas in C, there must be cast from `char` to `int` in `getchar` and a reverse cast in the caller.
3. Consider this as sample of clarity: ‘The NULL pointer is the only pointer which is defined to be invalid. Therefore a C implementation is invalid if it gives a valid pointer a value of NULL.’
4. The C text book [Ker,Rit] is often confusing but is sometimes confused itself! A couple of examples are given.
   i. On page 49 and 191 (edition 1) it says that ‘?’ has left to right precedence whereas on page 215 it says the opposite!
   ii. Consider the following from edition 2, pg.45

   > *In the construction **(type-name)expression** the expression is converted to the named type (by the conversion rules above). The precise meaning of a cast is as if the **expression** were assigned to a variable of the specified type, which is then used in place of the whole construction*

   This evidently means that in the expression `x = e` the cast from the type of `e` to the type of `x` is built-in and therefore `x = (T)e` (T is the type of x) is unnecessary. However consider the quote [Ker,Rit] (edition 2) pg.167

*The pointer returned by* `calloc` *has the proper alignment for the object in question, but it must be cast into the appropriate type, as in*

```
int *ip;
ip=(int *)calloc(n,sizeof(int));
```

How to reconcile these 2 statements is beyond me!

## 4.6 Brittle lexical and syntactic structure

Examples are:

1. # must appear in column 1. Admittedly, a number of largely successful languages eg. BASIC and FORTRAN are line-oriented. In C however, it is more hard to debug because for the most part it is free-form and only in a few instances is it line-oriented. This makes errors harder to spot.
2. A number is in base eight if it is prefixed with a 0. Therefore `10` is 10 but `010` is 8.
3. Character constants are enclosed in single quotes whereas string constants are enclosed in double quotes. This should obviously mean that `'AB'` is invalid. Unfortunately it is not invalid but implementation-dependent.
4. Combine Nup-1 with the preceding two items and we get the following anomaly: octal 101 is the character `'A'`.* In C octal numbers are written with a preceding 0 so the following prints the character A:
   ```
   printf("%c", 0101);
   ```
   If however we use `'\0101'` which is consistent with the C convention for octal numbers, as well as the style used on pg. 35 of edition 1 and pg. 37 of edition 2 of [Ker,Rit], we get some implementation dependent 2-character constant.

## 4.7 Semantic Non-redundancy

1. `if (x = 1) ...`
   is valid, but has a meaning which is most probably not intended. This happens because both `x = 1` and `x == 1` are arithmetic **expressions**.
2. The lack of a boolean type forces expressions like `x <= N` to be of `int` type. Therefore expressions `M <= x <= N` are semantically valid although pragmatically useless.

## 4.8 Mess Courtesy ANSI

The ANSI standard adds a number of irregularities to C. The multi-meaning of 'meaningless' (`void`) has already been cited in Puns. The `enum` type is another blunder. The idea is evidently borrowed from Pascal but all its benefits are forfeited.

1. Enum types are not type-checked – they are like integers.
2. Enum types may have equivalent values. eg.
   ```
   typedef enum {
     red, blue,
     green=0, yellow
   } color;
   ```
   defines `color` to be such that `red == green` and `blue == yellow`.
3. One of the major uses of enumerated types in Pascal is to pack small fields tightly into machine words. This use is not supported by the ANSI standard.

This is a striking example of how to get the over-restriction of strong typing (`enum` types not allowed in bit-fields) along with the chaos of typelessness:

---

* assuming ASCII

```
(yellow - green == red)
```

How does a teacher justify such a language feature?

## 5. Consequences of C as a mother-tongue

The harm done by an overly early introduction to C is large although invisible. C-mother-tongue programmers are life long bound to find simple, elegant yet wonderfully rich ideas like Object Oriented Programming, functional programming, modularity, CSP etc. as difficult, unnatural and 'advanced'. If these claims seem like exaggerations, it is only because of human beings' ability to unlearn. The amount of stuff which a C programmer knows, but subsequently must be discarded as being operational, machine-specific, implementation-specific, sequential-paradigm specific, contrary to robust software engineering etc., is incalculably large.

Just as a C programmer finds it difficult to grow upwards towards more high level languages, he finds it difficult to grow downwards as well. Experience with engineers (B.Techs) who are first trained on hardware, suggests that an assembly language programmer can learn C much faster than conversely. Surprisingly, an assembly language programmer can also learn higher level languages (eg. functional and object oriented) more easily than expert C programmers. The reasons seem to be that

i. Assembly language programmers are acutely conscious about the drudgery that low level programming entails. C programmers however have a persistent illusion of working in a higher level language.

ii. The assembly language programmer has a firm foothold on computational processes even if his/her understanding is entirely operational.

This seems to indicate that the classical level hierarchy:

| |
|---|
| Very High level languages<br>eg. Lisp, Prolog, APL, Smalltalk |
| High level languages<br>eg. Modula, Ada |
| Medium level languages<br>eg. C, Pascal, Fortran, COBOL |
| Assembly |

should be replaced by this one:

| | | |
|---|---|---|
| HLL's 60 .. 70<br>eg. Fortran, COBOL | HLL's 70 .. 80<br>eg. C, Pascal, Modula | HLL's 80 .. 90's<br>eg. Eiffel, Scheme, Haskell, Prolog |
| ASSEMBLY | | |

Moving upward is easy, moving downward is a little harder, but moving right horizontally is next to impossible because it does not demand increase in just knowledge but a new world view.

Furthermore the performance profile is very different today. Along with the first figure went the dictum:

*Climb the ladder of expressivity and you shall descend into the abyss of inefficiency*

Today this is not true any more. APL is highly amenable to vectorization, object-orientation seems to be the proper way for harnessing multiprocessors and data-flow machines promise efficient realization of functional languages. Today's slogan has become:

*Think high, think clean, and efficiency shall be added unto you.*

## 5.1 Impact on subsequent Courses

### 5.1.1 Data-structures

One of the worst hit subsequent courses after an initial C course is the data-structures course. Instead of data structures being a study of mapping discrete structures into computer implementations, it usually degenerates into a study of pointer-structures wherein a successful student is one who can handle '5-*' constructions. In this respect [Meyer2] says:

> *"I hadn't appreciated the C epidemic in the US. I began to appreciate how bad it was when I taught an undergraduate 'Data Structures and Algorithms' course, where usage of C was required by department policy. How could I even try to teach systematic algorithm development when I knew that the bulk of the student's time was spent fighting tricky pointer arithmetic, chasing memory allocation bugs, trying to figure out whether an argument was a structure or a pointer, making sure the number of asterisks was right, and so on? I am afraid it will be hard to recover from the damage caused by C to an entire generation of programmers."*

Famous is the story of the C-fan cum Lisp-antagonist who averred that Lisp is unsuitable for introducing data-structures because 'it possesses no data structures.'

If this displays ignorance of Lisp then it is but a small hole in the knowledge of the speaker. However it indicates a far more pernicious flaw in the understanding of the speaker:

> *An entity is not a data structure unless it prickles like a porcupine with pointers.*

This means that an integer for example, is not a data structure (unless it is larger than `long` and hence is implemented as a linked list) – **links** make data-structures.

In the field of data-structures there are a number of beautiful ideas which are perfectly simple in their own right.
Here are only a few examples.

i. Data-structures as values [Bird,Wad] [Reade] [Abel,Suss]
ii. Functional data-structures [Reade]
iii. Data-structures with logic variables [Clock,Mell]
iv. Object-oriented classification of data-structures [Meyer1]
v. Iterators [Lisk,Gutt]
vi. Infinite data-structures [Bird,Wad]
vii. Parameterized Data-Types. The number of references (nearly all of the above and many more) is too big to cite but [Goguen1-2] are extremely interesting even though a little advanced.

For a C programmer all these ideas are terribly advanced, not because they are inherently difficult but because their presentations in C, if at all possible, are convoluted.

### 5.1.2 Algorithms

When C is the base, some concepts such as coroutines and concurrency are not taught at all, and some like back-tracking become labelled as difficult.

Although many will claim that the study of algorithms is language independent consider the following definition of quicksort in Haskell:

$$
\begin{aligned}
&qsort\ [] = [] \\
&qsort\ (a{:}l) = qsort\ [x \mid x \in l, x < a]\ ++ \\
&\qquad\qquad [a]\ ++ \\
&\qquad\qquad qsort\ [x \mid x \in l, x \geq a]
\end{aligned}
$$

In order to understand this, all we need to know is that '++' is list-append and ':' is `CONS` as in Lisp. Compare this with the laborious presentation given in most data-structure and algorithm texts.

Further the study of algorithms bifurcates into a basic study of sequential algorithms and an advanced study

of parallel algorithms. Those who believe this distinction to be fundamental are invited to study the Occam introduction [Jones,Gold], and see if within the context of Occam, a distinction between sequential programs and parallel programs is necessary or natural. Another impressive tour-de-force is [Chan,Mish]. It demonstrates that algorithms can be developed at such a high level of abstraction that they are independent of programming paradigm and implementation architecture. They may be subsequently refined to various machines and languages **of which the sequential is just one possibility.**

### 5.1.3 Assembly

If C followed assembly (or perhaps even better, if they were introduced simultaneously), the students would appreciate both and see C as a quick and safe way of producing assembly. However when C is the initial foundation, then assembly seems like an unnecessary labour.

### 5.1.4 Other Courses

Courses like Compiler Construction, Operating systems, DBMS and Networks are not much affected in their course work, but suffer indirectly in their programming assignments which are typically done in C. This is so because the learning curve for C is slow and much of the students' time is spent on mastering programming issues even long after the introductory programming course.

## 5.2 Sociological Consequences

The following conversations are a sample of what the author often hears. They represent what the budding C-experts find ludicrous about the ignoramuses.

— "He put strings as case labels! Ha Ha ..."

— "She used an assignment statement: `s = malloc();` where `s` was a `struct` rather a `struct` pointer! When I drew her attention to this she said, 'Oh, shall I use `&s` rather than `s`?' Ha,Ha ..."

These C 'experts' often regale me with such tit-bits. I try to tell them that these details are irrelevant to computer science, to computer programming or even good old imperative programming a la BASIC, FORTRAN and COBOL. My discomfort becomes very acute when I see that it is impossible to make them see what I am saying. The C-expert's mind is so congested with bit-nibbling that deeper concepts find no place.

In an environment where C is the mother-tongue, the C-expert is the hero – and one who cannot think crooked is labelled as one who cannot think.

## 5.3 Software Engineering Consequences

I. C supports a curious illusion of being high level. The reason seems to be that typical Unix environments have good supporting software. This however does not make C high level.

II. C is versatile. For example, C functions can be used for using a functional style, separate source files can be used for modularity, C `struct`s containing function pointers can be used to simulate an object-oriented style, laxity of function types along with interconvertability of pointers can be used to make generic functions and much more.

    This may be very good in a work-environment where a single powerful language should be emphasized. In an educational environment however, this has a negative influence because the essential ideas become twisted when represented in C. The fresh students see twistedness very early and believe it to be the law of life rather than a quirk of current technology. Among the examples mentioned above I consider a couple:

    - Functional style:
      A devoted C programmer may very well say that C is functional or alternatively, that the functionality of C is enough because C has functions. It is hard to convince such, that when heap allocated pointer types are returned by functions, the global connectivity of the program

drastically increases (the module which allocates space may not be in a position to free it when the need arises). Therefore functional decomposition as supported in functional languages is not truly supported in C.

- Modularity
  Because C offers no notion of modularity other than the use of separate files, C programmers easily believe that modularity means putting various pieces of code into separate files. The fallacies in this approach are:

  — Files are an operating system concept. Modules are a programming language concept. Simulation of one by the other is more cumbersome and error-prone than direct support.

  — The C mechanism for connecting different modules is 'header-files'. A header-file invariably contains all sorts of information which should ideally be hidden and is therefore against the principle of information hiding. The inclusion of header files is therefore an invitation to anarchy.

In short, although the fact that C can do everything is readily recognized, the fact that it *does most things poorly* is rarely noticed.

III. C programmers have a very low capacity for abstract thinking because the abstraction mechanisms provided by C are weak. The effect is seen when students present extremely C-specific designs:
The examples range from small things like saying, '`name` is of type `char *`' rather than '`name` is a string', to believing that the C/UNIX paradigm is an eternal verity, rather than seeing it as just one possible implementation environment.

IV. The type structure of C inculcates habits that inhibit clear understanding. The equivalence of `char` and `int` has already been mentioned. Far more detrimental to the ability to think clean is the absence of a boolean type.

The importance of the boolean type is obvious whether we consider it as central to logic or whether we consider it to be the heart of digital systems. The inability of students to think with boolean values as values in their own right shows up when students almost always write

```
odd(int x)
{
  if (x%2 == 1)
    return 1;
  else return 0;
}
```

rather than the simpler, shorter, more efficient and more understandable

```
boolean odd(int x)
{
    return x%2 == 1;
}
```

The invisible but greater problem with no boolean type is that not understanding boolean operations as operations in their own right makes students that much further intellectually, from program verification which is usually based on predicate calculus [Gries].

The bad effects of pointers has already been mentioned but bears repetition. [Wirth] is a thoughtful evaluation on the problems of pointers, [Hoare] is more blunt:

> *Their introduction (pointers) into high-level languages has been a step backward from which we may never recover.*

The typical C programmer equates pointers to data-structures rather than seeing that C **does not**

support any data-structures other than scalars and that pointers are a poor man's emulation for real data-structures.

V. C perpetuates many a software-engineering myth. Some examples:

1. Programming 'in-the-large' is something different from programming 'in-the-small' and is terribly difficult. One can find a variety of answers to this problem:

   — Ada which is geared towards large scale systems

   — Eiffel which has Ada-like constructs for large scale software but allows highly generic software to be developed so that size may be reduced.

   — APL wherein large-scale programming usually becomes unnecessary because of the tremendous expressivity of the language. The greatest tribute to the language is the abuse: 'It is not a language for serious software engineering.' Most APL users are not computer scientists but general users ranging from physicists to economists to school teachers. They survive without the expensive services of software-engineers mainly because of APL.

   — Lisp which encourages a meta-level style. For any non-trivial problem a suitable base language is designed and embedded within Lisp. The expert Lisp programmer is one who formulates a coherent body of new ideas which he uses to build his own little world. Although this may be contrary to the usual software-engineering dogma which says that good code is stereo-typed code, it works very well because it usually obviates the very need for software-engineering. A Lisp programmer can produce what a battery of C programmers cannot [Stall].

   The fact that C is unsuitable for large-scale software development does not imply that large-scale software development is very difficult.

2. Systematic software verification is a mathematician's dream.

   Again this is only true because we are considering verification of software written in C-like languages. Some examples of success stories are:

   — Functional and logic languages which, compared to C, are self-documenting and self-verifying

   — Eiffel which incorporates a small assertion language. These assertions can be turned on with a compilation switch. This is an important step in combining systematic proof techniques with traditional debugging.

VI. Large bodies of Pragmatics
Software engineers are often inculcated with a number of 'guidelines' such as
(i) indent your programs (ii) comment your programs
and many more.

Most of these rules can easily be incorporated into programming languages. An example is Occam wherein indentation and minimal commenting is required *by the system*. This means that many such rules of thumb (which programmers proverbially break) can be completely dispensed with by making them part of the language. For examples of wise pragmatics that become obsolete with improved programming languages, consult [Ker,Plau]. It contains many suggestions relevant to Fortran but irrelevant in C.

An example of how pragmatics increase because of poor language design is C++[Strou]. The C++ manifesto is twofold:

i. to be a proper superset of C

ii. To support object-orientation

Since C is low level and object orientation is an inherently high level concept the result is conceptually incoherent. [Strou] is full of weak directives such as

> *Macros (*`#define`*) are almost never necessary in C++; use* `const` *or* `enum` *to define manifest constants, and* `inline` *to avoid function calling overhead.*

Why have Macros then? Because of item (i) of the manifesto.

VII. The greatest impact is on the students as future computer scientists.

C is the relic of an era when the average machine was 10 times slower and most memories were 100 times smaller than what is generally found today. The typical C programmer objects to modern languages supporting storage management, first-class values, persistent objects, transparent concurrency and other such forward-looking concepts usually on grounds of efficiency. This is very similar to a FORTRAN-1 programmer objecting to structured constructs because of the same reasons. If computing depends on such retrogressive forces we will all get stuck.

The property of C being weakly-typed and unverifiable along with the existence of good debuggers, is creating a generation of terminal-hooked hackers who can debug for 3 days but cannot use pen and paper for an hour.

## 6. Conclusion

In keeping with the negative spirit of this paper, no constructive solution is being proposed. This attitude may be justified by saying that it is important to diagnose a disease before we treat it.

A possible suggestion which many computer scientists might read into the article – to teach a respectable language like Pascal or Modula – is ***not*** being suggested. I admit that Pascal or Modula as a first language would certainly make it easier to learn programming than C. The problem however is that the most fertile areas of today would remain in the realm of the advanced, whereas the obsolete paradigms would be consolidated.

The Occam programmer finds parallelly running sequential programs natural, ML and APL programmers use a highly mathematical style of programming as though it were their right, the Eiffel programmer does not only talk about (and write PhD theses on) abstract data types, but uses them in his/her day to day work, the Scheme programmer swims easily in a beautifully homogeneous world of data-structures ranging from integers to lists to functions to exotica such as environments and continuations.

I quote from the Occam introduction: [Jones,Gold]

> *It used to be that writing a program meant finding a strict sequence of steps to achieve the desired end. ... the important thing that we show in this book is ... that parallel programs can be simpler to write and understand than sequential programs that achieve the same effect.*

It may be surprising to most that parallel programming is **more natural** than sequential programming, but there it is. The small price we are required to pay is to 'think in Occam'.

A number of other books make similar claims for functional, object-oriented and other promising paradigms. eg. [Meyer1], [Abel,Suss], [Spr,Fried], [Wiks]. All these claims can be summarized as: 'The appropriate notation helps one to think clearly.' Two forceful demonstrations of this claim are [Iver] and [Gast]. That these authors vindicate this claim can be verified by anyone who would care to read this material. I am not sure whether there exists a consistent simple combination of all these rich possibilities. ([Goguen2] might be an answer in the positive direction). I do feel however that to say, 'Since I cannot get the best caviar, chinese food and steak at the same restaurant, I shall starve!' seems to be unreasonable.

There are those who would object saying, 'We teach language independent problem-solving.' An answer to

this is to be found in the excellent programming text-book [Myers]:

> *This normally means that programs are first outlined in a pseudo-code which strongly resembles* ***Pascal, Algol, Ada, PL/1 or Modula-2*** *but not* ***Lisp, APL, Prolog or ML.*** *Pidgin Pascal is no more language independent than Pidgin APL would be; it may well be more useful, but neither is a specification language. The crucial omission is specification ...*

He admits however, that his text is best supplemented with ML, KRC or Scheme.

Having said this, I should add that incorporating a course on *Programming Paradigms* is pernicious because by its very nature the course becomes an advanced course whereas functional, object-oriented and other such programming paradigms have been introduced to *reduce* the complexity associated with von Neumann programming. This course would perpetuate the situation in which von Neumann languages are considered proper and the others are considered avant-garde.

I end with a disclaimer.

This article might easily be misinterpreted as saying that the author does not like C. This is not true. I have written non-trivial applications in C and enjoyed doing it. If complaint I must make, there are the following:

I. I deprecate the Computer Science educators who do not distinguish between the lasting and the ephemeral. Here I refer to the common confusion between programming, and coding into language X, whatever that X may be.

II. I deplore the computer scientists who cannot discriminate between the real causes behind a success and incidental details. Here I refer to the C/Unix success story.

    By elegant example, the inventors of Unix demonstrated a number of things. The scenario seen today in the computer world indicates that many computer scientists have completely missed these lessons. Rather, wrong ideas are deified into eternal truths. For example:

    1. They showed that operating system development which was traditionally done at no higher than assembly level, could be done at a much higher level.

       Yet today applications are being reduced to C, that were previously developed at a higher level.

    2. They showed that a workable, 'real' high level language need not be unduly complex – C used for Unix is simpler than PL/1 used for Multics.

       However new languages continue to get more and more complex.

    3. They demonstrated a 'tools approach' to software development. They showed that it is often profitable to develop an appropriate tool for sizable applications – C for Unix – than to make do with available resources, even if the total work seems to increase.

       Ironically, C has become the name of the Turing-Tarpit. In the name of efficiency, portability or some other such software-engineering buzz-word, C is used for everything.

    4. Unix sports an audacious and in fact brazen disregard for efficiency. For example the directory list command, `ls`, starts up an entire process. If this were integrated into the shell, it would easily be an order of magnitude faster. However it is precisely this unconcern for efficiency that makes Unix so soft on the software engineer.

       Yet, in spite of the tremendous increase in machine power and in spite of overwhelming evidence that software quality and programmer productivity are severely impaired by a misguided concern for efficiency, programmers compulsively continue to count clock-cycles.

III. I am appalled at the monstrous messes that computer scientists can produce under the name of 'improvements'. It is to efforts such as C++ that I here refer. These artifacts are filled with frills and features but lack coherence, simplicity, understandability and implementability. If computer scientists could see that art is at the root of the best science, such ugly creatures could never take

birth.

# 7. References

The references are partitioned into 2 sections. The first contains text-books that are excellent in quality and highly readable even for junior students. They are nevertheless unknown to most of today's CS students because they don't fit into the obsolete model of CS education that is generally used – and they are therefore unknown to tomorrow's computer scientists. To claim that these excellent texts are unknown to today's computer scientists because of C, may seem absurd. However it is a fact that the finiteness of CS course lenghts, along with pressures of learning C, preclude sufficient exposure to this rich body of literature.

## 7.1 Introductory Texts


[Abel,Suss] Abelson, H. and Sussman, G.J.; Structure And Interpretation Of Computer Programs. MIT Press 1985

[Bird,Wad] Bird, R. and Wadler, P.; Introduction To Functional Programming. Prentice Hall 1988

[Chan,Mish] Chandy, K.M. and Mishra, J.; Parallel Programming: A Foundation. Addison Wesley 1988

[Clock,Mell] Clocksin, W.F. and Mellish, C.S.; Programming In Prolog. Springer Verlag 1984

[Gries] Gries, D.; The Science of Programming. Springer Verlag 1981

[Jones,Gold] Jones, G. and Goldsmith, M.; Programming in Occam2. Prentice Hall 1988

[Lisk,Gutt] Liskov, B. and Guttag, J.; Abstraction And Specification In Program Development. MIT Press 1986

[Meyer1] Meyer, B.; Object-Oriented Software Construction. Prentice Hall International 1988

[Myers] Myers, T.J.; Equations Models and Programs. Prentice Hall 1988

[Reade] Reade; Elements of Functional Programming. Addison Wesley 1989

[Spr,Fried] Springer Friedman; Scheme & the Art of Programming. MIT Press 1989

[Wiks] Wikstrom, A.; Functional Programming using Standard ML. Prentice Hall 1988


## 7.2 Others


[Gast] van Gasteren, A; On the shape of mathematical Arguments. Ph.D. Thesis, Technical University Eindhoven 1988.

[Goguen1] Goguen, J.A.; Principles of Parameterized Programming. Software Reusability: Vol. 1 Concepts and Models. Eds. Biggerstaff, T.J. and Perlis, A.J. Addison Wesley 1989

[Goguen2] Goguen, J.A.; Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics. Research Directions in Object Oriented Programming. Eds. Shriver, B. and Wegner, P. MIT Press 1987

[Harb,Stl] Harbison, S.P. and Steele G.L.; C : A Reference Manual. Prentice Hall International 1987

[Hoare] Hoare, C.A.R.; Hints on Programming Language Design. Sigact/Sigplan Symposium on Principles of Programming Language 1973

[Iver] Iverson, K.; Notation as tool for thought. 1979 ACM Turing Award Lecture. CACM August 1980

[Ker,Rit] Kernighan, B. and Ritchie, D.; The C Programming Language. Prentice Hall, First Edition 1978, Second Edition 1989

[Ker,Plau] Kernighan, B. and Plauger, P.J.; The Elements of Programming Style. McGraw Hill 1978

[McG] McGettrick, A. D.; ALGOL 68 a first and second course. Cambridge University Press 1978

[Meyer2] Meyer, B.; Conversation with Bertrand Meyer. JOOP May/June 1989

[Shoo] Shooman, M.L.; Software Engineering. Mcgraw Hill 1983

[Stall] Stallman, R.M.; EMACS: The Extensible, Customizable, Self-Documenting Display Editor. Interactive Programming Environments. Ed Barstow et. al. McGraw Hill 1986

[Strou] Stroustrup; The C++ programming Language. Addison Wesley 1986

[Wirth] Wirth, N.; On the design of Programming Languages. Proc. IFIP Congress 74 North Holland

[Wulf] Wulf, W.A. et. al.; Bliss a language for systems programming. CACM Dec. 1971