Why Not "C"?

This is the second draft of some personal thoughts on the "C" language. The first draft was sent to a number of serious "C" hackers. These, naturally, objected to the general tone of the note, and pointed out some errors in the first draft, but even they found it difficult to find any real way of refuting the conclusions.

Their replies, however, did answer one of my main worries: why do some people *like* writing software in "C"? The answer seems to be that it is a bit like doing a crossword puzzle. Crossword puzzles (or the non-British equivalents) contribute nothing to a countries GDP or its environment, while consuming a disproportionate part the mental efforts of some of the most influential members of society. How much better would the world be if even a small fraction of that mental effort were devoted to solving real problems? "C" not only provides the computer equivalent of the crossword puzzle, but also, in an era where most computer programming has become trivial, it turns the most mundane of programming tasks into a major intellectual exercise. For a True Believer, the problems inherent in using "C", some of which are described below, are its major attraction.

"C" Clearly

I do not think that even the most ardent supporter of "C" could contend that "C" programs are *naturally* clear, easy to read and easy to understand. Supporters of "C" contend that this inherent lack of clarity is a natural consequence of the power of the language and that "C" programs are only as difficult to understand as the problems they are representing. These supporters of "C" point out that the programs they write themselves are clear and easily understood: it is only incompetent programmers (i.e. the rest of the world) who make such a mess of writing software in "C".

I do not subscribe to this view. It seems to me that the difficulty of producing clear and easily understood programs in "C" arises from fundamental shortcomings in the design of the language. While there are many aspects that I find unsatisfactory, there are two faults in conception which, in my view, should rule out "C" from any serious consideration as a general programming language.

1. From the point of view of the functionality of a programming language, "C", being conceived for an archaic operating system on a (now obsolete) specific computer, is not well adapted to implementing either applications programs or systems software for systems using current technology.
2. From the point of view of the construction of a programming language, "C" is inherently incoherent and inconsistent.

That "C" is an incoherent language is beyond dispute, so I shall only present a few examples. The question of appropriateness is less clear. Some people are still using the archaic operating system for which "C" was developed and so may not appreciate how odd many of the underlying concepts of "C" really are. Others are not familiar with the coordination problems of large system developments or the maintenance problems peculiar to the continuous development of software used by the general public. I shall, therefore, deal with my more subjective criticism first.

Uniquely inappropriate

Each time I encounter "C", I am struck by the extent to which it is based on the assumptions and restrictions of UNIX. Whatever system "C" is implemented on, it tries to twist the real world to fit the UNIX world of multi-user mainframes. It forces the use of a memory model which is not only inappropriate for the majority of processors manufactured today, but also is inefficient and serves merely to obstruct communication between tasks (processes). The UNIX memory model is built into "C" from the lowest level upwards and many more UNIX related assumptions are made in the standard headers and libraries.

The most serious UNIX related assumption is that IO is merely a matter of sending a stream of bytes from memory to a peripheral or vice versa. This was not necessarily true even in the 1960s when "C" was born (although it was true for the multi-user mainframes on which the design of both UNIX and "C" were based). Today it is a ridiculous assumption to make. User interfaces, even for real time systems, are now dominated by graphical displays and attempts to make sensors and actuators in intelligent machinery conform to the serial IO model have not always been successful.[1]

Almost as serious is the assumption, in the "C" standard libraries and headers, that the operating system's IO is so inefficient that the bulk of the work (buffering, unbuffering, building protocols and maintaining information about the state of the IO) needs to be done within an application itself (albeit by "library routines") rather than leaving the operating system to provide the most appropriate means for the hardware in use.[2]

By using a more advanced IO model, it is possible to scrap the entire standard "C" IO system, stripping out layers of structures and routines and streamlining the whole process. Unfortunately, although the resulting programs are smaller, more portable and potentially more efficient, they are not standard "C".

Applications or systems development?

If we scrap the antiquated IO system of "C", and we find some way of living with the UNIX memory model, would "C" be a good language to use for either applications or system software development?

As a systems software development language "C" appears to have some advantages. It has some quite compact notations for performing multiple assignments (or assigning and testing) and providing auto increment and decrement of pointers. The intention appears to be to enable the software developer to guide the compiler towards producing efficient code. However, even very simple optimisation strategies are likely to provide as effective code generation from more readable, if less compact, source code. In particular, the auto increment and

1 It is possible to graft extra layers such as the "X" protocols on top of this primitive IO, but the result is even less efficient and more cumbersome than "C" on its own.

2 This "do it yourself" approach to IO protocol handling is at the heart of "C" portability problems. As the protocols for handling different peripherals have to be built into the application program, the application program needs to be changed to accommodate different protocols. Hence the current interest in standardising (i.e. downgrading to the lowest common denominator) peripheral protocols.

decrement operations do not necessarily translate very well from processor to processor[3] and it would be better to let the compiler choose an suitable method.

These "C" tricks, therefore, seldom provide any real benefit in speed or code size, while they frequently have an undesirable effect on clarity.

On the benefit side, however, "C" does have a useful, usable and clear definition of simple data structures which are adequate for elementary system software. These are, unfortunately, not just unnecessarily rigid, but also are let down by the methods used in "C" to handle arrays and pointers.

As an applications development language, where the primary aim must be for clarity (on any sensible operating system the bulk of the nitty gritty work should be done by the operating system and associated support software) and coherence (so that the user does not have to learn every facility, but, having learnt to use part of a system, using the rest is fairly natural), "C" is a non-starter. With its internal inconsistencies, "C" does not encourage either clarity in program expression or coherency in user interfaces.[4]

For applications developed by "average" users, the fundamental requirement is that the system must be easy to access and not require extensive background knowledge. My most recent foray into a "C" development system shows how "C" has developed. This system (released 1992) comes with more than 4000 pages of manual. The introduction quite frankly admits that the manuals give no information on how to write a program in "C". Using the system for more than "hello world" can be daunting: the "C" compiler itself (ignoring the linker and the debugger) has more than 120 command line options! If this development system defeated even an experienced systems consultant (not me, I don't give up that easily) what hope has the average user?

"C" how inconsistent you can be

Originally "C" was intended to be very flexible. At the time it was developed, flexibility was seen as the opposite of coherence. A primitive language was defined (I will call this "True C") and the flexibility was provided by a macro processor. It became standard practice to put the macro definitions into "header files" which could be included into the source files. This may have been intended to maintain some coherence, but, if so, it was largely unsuccessful. Different header files were defined at different times and used different conventions. Moreover, if it is to be possible to transfer a source file to another machine, then the effect of the macro definitions needs to be the same. This gives rise to a much larger definition of the "C" language ("Standard C"). In "Standard C", the meaning of the standard macros if fixed, as are the functions of the standard library. While this standardisation is necessary, the effect is that much of the original flexibility of the language is lost while the incoherence remains.

---

[3] The MC680x0, for example, does not have pre-increment or post-decrement, while the i80x86 has only post-increment and post-decrement and, for efficient i80x86 code, the two should not be mixed.

[4] As a long-time computer user as well as software developer, I would claim that systems written in particular languages tend to show the same generic traits as the languages used to develop them. I am aware that these traits can be minimised or even removed altogether but, never the less, I still believe that while Pascal programs tend to be simplistic and BASIC programs primitive, "C" programs tend to be incoherent.

Nevertheless, there is still some flexibility: additional header files can be defined to create application specific macros, and these application specific macros can be used to reduce the time taken to write the code. This has the unfortunate effect that, without a complete understanding of the operation and the side effects of these macros, the code can become incomprehensible. Even more problems arise when, as is commonplace, macros have different definitions in different header files: this is the natural result of allowing different source files to have different headers.

Once you have taken the decision that consistancy is important, that all macros of the same name must have the same meaning and that all data structures of the same name must have the same structure, the header file concept becomes just a nuisance. As all source files need the same definitions, these definitions neither need to be included, nor should be included in individual source files. These definitions should be part of the programming environment.

"C" adepts will know all about the inconsistency of the "C" language, but with practice may have come to exploit them. Trying to pick a few examples from the chaos of "C" is rather difficult, but I will start with the fundamental layout.

"True C" itself is defined in such a way that the end of line is no more (or less) significant than a space or spaces (or tab). "True C" source is treated as continuous text, regardless of how it is arranged on lines.[5] "Standard C", however, includes a macro pre-processor (even if, these days, it is likely to be included in the "C" compiler itself). For historical reasons ("C" was conceived for a patchwork quilt of an operating system, and so developed as a patchwork quilt of a program development environment) the macro pre-processor has a line based syntax, at odds with the "True C" language itself. "Standard C", therefore, ends up by treating the end of line as significant or insignificant depending on the context.

Within the body of a "C" program, the syntax rules for separators are also inconsistent. Braces can be used to group statements into a compound statement where the syntax requires a single statement:

```
if (cond) stmt;                 can be replaced by ...
if (cond) {.....}               (note the ; has disappeared).
```

The way this is done is bizarre, not only does the semicolon disappear from the end of the statement (often you can leave it there, but in some contexts it is illegal or may change the sense of the code) but, within the braces, you require an extra semicolon:

```
if (cond) {stmt; stmt; stmt}    is illegal because ...
if (cond) {stmt; stmt; stmt;}   the extra ; is required.
```

However, the same quirk does not exist when you use braces to group initialisation expressions.

```
int day = s+1;
int day[4] = {s+1,s+2,s+3,s+4};    the ; is still there and there is no extra comma.
```

5 If you have some literal strings then things get a bit more tricky. Putting a newline character in a string may (or may not) insert a newline character into the line, or it may cause a compiler error. To be sure, you should insert newline characters with "\n" or remove them with "\".

A strict syntax is acceptable if it is well defined and consistent, but not if it is bizarre, inconsistent and deviations can change the sense of the code.

There is, naturally, a contorted logic to justify this inconsistency. Used here, the semicolon is not defined as a separator, but as an expression statement terminator. The compound statement, on the other hand, does not require a terminator. This unusual syntax definition, however, seems to be no more than an attempt to justify, in retrospect, an error. In the "for" statement, the semicolon is used, beyond any reasonable doubt, as an expression[6] separator.

for (expr; expr; expr) stmt;	defines initial, test, and repeat expressions.

To be consistent with compound statements or compound initialisations, we should be able to have more than one initialisation expression.

for ({expr; expr;} expr; expr) stmt;	this would be consistent with compound statements
for ({expr; expr}; expr; expr) stmt;	this would be consistent with compound initialisation
for (expr, expr; expr; expr) stmt;	this, however, is required.

This introduces yet another construction of compound item, the comma expression where the meaning is critically dependent on the presence (;) or absence (,) of a dot. It would be too much to hope that this unnecessary addition to the "C" syntax could be used anywhere where an expression could be used, but that would be consistent. That would not be "C".

The point is to confuse

The champion of muddled thinking in "C" must be the definition and handling of pointers. In "C" you can use pointers to data stored in memory, but, despite initial appearances, you cannot declare a variable to be a pointer to something (the address of something).

Logically, if we wish to declare a variable to be a pointer to an int, (i.e. the variable holds the address of the int, not the int itself) we should use the address operator to qualify the type declaration.

&int my_ptr;	would declare my_ptr to be &int - the address of an int.

But this is "C". Instead of declaring that my_ptr is a pointer to an int, "C" requires us to declare that "dereferencing" my_ptr gives an int.

int *my_ptr	declares the object pointed to by my_ptr to be an int

This can not only mislead the inexperienced into thinking they have actually declared an int but this back-to-front definition falls apart when a pointer is to be initialised.

int xx = 1;	initialises xx to 1, but...
int *xxx = 1;	does not initialise *xxx to 1, it initialises xxx to 1.

If the declaration were made the "right way round", there is no problem.

```
&int xxx = 1;
```

6 The difference between "statement" and "expression" cannot explain the inconsistency in the use of the semicolon. Part of the flexibility of "C" is that "statement" and "expression" are almost interchangeable. In C++, there is even less difference between them.

Life is made even more complicated by the automatic referencing or dereferencing of identifiers, which seems to vary quite considerably from compiler to compiler.[7]

Furthermore, the use of arrays is needlessly complicated by treating both * and [ ] as dereferencing operators. Even more confusingly, the [ ] operator can be prefix or postfix. Thus to access the second element of the array "fred" you can use at least three different constructions.

fred[2]	logical this, but...
2[fred]	is the same!!
*(fred+2)	as both translate to this.

This is typical of "C". The same effect can be obtained using completely different types of syntax. But, while this provides an unnecessary flexibility of expression, it prohibits direct manipulation of pointers. Adding a value to a pointer is always treated as the same as indexing the pointer (a multiple of the object size is added to the object).

This might seem reasonable except that

1. it is unnecessary;
2. it takes no account of variable object sizes;
3. it does not allow position independent data structures.[8]

The prohibition of position independent data structures prevents structures that include internal pointers from being copied from place to place or from disk to memory or vice versa.

As there is a separate syntax for indexing pointers, re-defining the + and - operators so that addition and subtraction also index pointers is unnecessary and merely provides more opportunity for inconsistency while limiting the usefulness of the language.

"C" naturally has a bodge to get round this: you can "cast" your pointers to "char *", then the addition and subtraction operators revert to their normal meaning. This makes what should be quite a simple operation appear very complex while providing potential for undetectable errors on more complex operations.

These complaints of inconsistencies in "C" are not nit-picking. All I have done is to scratch the surface of the incoherence inherent in the "C" language. The deeper you look, the more inconsistencies you will find. Possibly the worst feature is the way that, in order to limit the number of symbols and keywords, "C" makes the meaning of many of the symbols and keywords strongly dependent on context. C++ brings some improvements to "C", but, at the same time, introduces more inconsistencies, and even more context dependencies.

---

7 The reference manual for the most recent version of C++ on the IBM PC merely states that "some non-lvalue identifiers ... are automatically converted to 'pointer to X' types when appearing in certain contexts" A software developer using this system has to guess which non-lvalue identifiers this automatic conversion applies to, in which contexts.

8 This prohibition of position independent data structures arises from the UNIX memory model which is assumed by "C". Unlike the real world, position independent structures are not *required* in the virtual UNIX world. As well as being essential in the real world, they can, however, be extremely useful in a virtual world.

"C" where we can go

It is a sad reflection on the state of the software development business that anyone would consider using "C". There is no possible justification for the inconsistencies, the confusions and the outright mess that characterise the "C" language. They do not give any benefit in terms of flexibility, code size or efficiency, they merely make it more difficult than necessary to produce quality code and encourage the worst in dirty tricks from the less disciplined software developers.

The DOD of the USA, the world's largest purchaser of software, rejected "C" a long time ago as being unsuited to quality software. They decided to require all their suppliers to use a new language. Their only real specification for this new language was that it must not be "C", and must not be based on "C". In its initial "not C" form, this new language, ADA, was indeed elegant until a committee of "experts" "improved" it.

A more important question than what sort of language should be used in place of "C" is whether the concept of programming language is really just a hangover from the days of punch-cards, paper tape, (glass) teletypes and line printers. If it is, what should replace programming languages?

In 1992, another instant panacea is being aggressively promoted: Object Oriented Programming. Having talked to a number of dedicated followers of this fashion, I think I need someone to explain to me why Object Oriented Programming is a good idea. I find the arguments in favour of Object Oriented Programming very confusing. To me the promoters of object oriented program seem rather like car salesmen trying to convince me of the benefits of buying a car with pneumatic tyres: I thought that my car was already equipped with them.

I could be wrong, but as far as I can tell, Object Oriented Programming is just a formal way of representing what we've been doing for years. Of course, for those conditioned to the oddities of "C", Object Oriented Programming might indeed seem to offer new possibilities and better way of creating software.

A formal approach can often improve clarity, although it will usually limit the flexibility of a technique. In the case of "C++", the methods used to represent a link between a data structure and the code used to process that data structure seem only to provide more opportunities for obscurantism. One example of over 300 lines of convoluted "C++" for the IBM PC could be written in six clear and explicit WBASIC statements or fifteen lines of MC680x0 assembler (including data declarations) for a rational operating system.

A recent promotional leaflet for C++, offered a "personal guide through the maze of C++ programming". If even the publishers of C++ compilers and development systems regard C++ as a maze, what hope is there for the average computer user? On the other hand, maybe I am being unfair to condemn Object Oriented Programming on the basis of the way it has been bodged into "C++".

Replies please

- Am I wrong? Is "C" really a well thought out language?
- Am I wrong? Is "C" really appropriate to modern computer systems?
- Do modern computer systems exist?
- Is the computer business really stuck in a 1960s time warp?
- Is Object Oriented Programming just an attempt to bypass some of the limitations of languages such COBOL, "C" and Pascal?