CENTRE FOR INDUSTRIAL CONTROL SCIENCE

Department of Electrical and Computer Engineering

The University of Newcastle
N.S.W. 2308, Australia

THE CASE AGAINST C

P.J. MOYLAN

Technical Report EE9240
July 1992

THE CASE AGAINST C

P.J. Moylan
Department of Electrical and Computer Engineering
The University of Newcastle
N.S.W. 2308, Australia

eepjm@wombat.newcastle.edu.au
Fax: +61 49 60 1712

ABSTRACT

The programming language C has been in widespread use since the early 1970s, and it is probably the language most widely used by computer science professionals. The goal of this paper is to argue that it is time to retire C in favour of a more modern language.

The choice of a programming language is often an emotional issue which is not subject to rational discussion. Nevertheless it is hoped to show here that there are good objective reasons why C is not a good choice for large programming projects. These reasons are related primarily to the issues of software readability and programmer productivity.

Keywords: Programming languages, C, C++

Introduction

This note was written after I had found myself saying and writing the same things over and over again to different people. Rather than keep repeating myself, I thought I should summarize my thoughts in a single document.

Although the title may sound frivolous, this is a serious document. I am deeply concerned about the widespread use of C for serious computer programming. The language has spread far beyond its intended application area. Furthermore, the C enthusiasts seem to be largely in ignorance of the advances which have been made in language design in the last 20 years. The misplaced loyalty to C is, in my opinion, just as serious a problem among professionals as the BASIC problem is among amateurs.

It is not my intention in this note to debate the relative merits of procedural (e.g. Pascal or C), functional (e.g. Lisp), and declarative languages (e.g. Prolog). That is a separate issue. My intention, rather, is to urge people using a procedural language to give preference to high-level languages.

In what follows, I shall be using Modula-2 as an example of a modern programming language. This is simply because it is a language about which I can talk intelligently. I am not suggesting that Modula-2 is perfect; but it at least serves to illustrate that there are languages which do not have the failings of C.

I do not consider C++ to be one such language, by the way. The question of C++ will be considered in a later section. For now, it is worth pointing out that almost of the criticisms of C which will be listed in this note apply equally well to C++. The C++ language is of course an improvement on C, but it does not solve many of the serious problems which C has.

Some background

The first C compiler, on a PDP-11, appeared in about 1972. At the time the PDP-11 was a relatively new machine, and few programming languages were available for it; the choice was essentially limited to assembly language, BASIC, and Fortran IV. (Compilers and interpreters for some other languages had been written, but were not widely distributed.) Given the obvious limitations of these languages for systems-level programming, there was a clear need for a new language.

This was also the era in which software designers were coming to accept that operating systems need not be written in assembly language. The first version of Unix (1969-70) was written in assembly language, but subsequently almost all of it was rewritten in C. To make this feasible, however, it was necessary to have a language which could bypass some of the safety checks which are built in to most high-level languages, and allow one to do things which could otherwise be done only in assembly or machine language. This led to the concept of intermediate-level machine-oriented languages.

C was not the only such language, and certainly not the earliest. In fact, a whole rash of machine-oriented languages appeared at about that time. (I was the author of one such language, SGL, which was used for a number of projects within our department in the 1970s. It was retired, as being somewhat old-fashioned, in the early 1980s.) These languages had a strong family resemblance to one another; not because the authors were copying from one another (in my own case, SGL had reached a fairly advanced stage before I became aware of the existence of C), but because they were all influenced by the same pool of ideas which were common property at the time.

"One of the biggest obstacles to the future of computing is C. C is the last attempt of the high priesthood to control the computing business. It's like the scribes and the Pharisees who did not want the masses to learn how to read and write." – Jerry Pournelle.

Why C became popular

The history of C is inextricably linked with the history of Unix. The Unix operating system is itself written in C, as are the majority of utility programs which come with Unix; and to the best of my knowledge a C compiler comes with every distribution of Unix, whereas it is harder to get compilers for other languages under Unix. Thus, we need to look at the reasons for the rapid spread of Unix.

The obvious reasons are cost and availability. Unix was distributed at virtually no cost, and sources were available to make it easy to port it to other systems. A number of useful utilities were available within Unix – written in C, of course – and it was usually simpler to leave them in C than to translate them to another language. For a Unix user who wanted to do any programming, a competence in C was almost essential.

Since then, C has remained widespread for the same reasons as why Fortran has remained widespread: once a language has built up a large user base, it develops an unstoppable momentum. When people are asked "why do you use C?", the most common answers are (a) easy availability of inexpensive compilers; (b) extensive subroutine libraries and tools; (c) everyone else uses it. The ready availability of compilers, libraries, and support tools is, of course, a direct consequence of the large number of users. And, of course, each generation of programming educators teaches students its favourite language.

Portability is also given as a reason for the popularity of C, but in my opinion this is a red herring. A *subset* of C is portable, but it is almost impossible to convince programmers to stick to that subset. The C compiler which I use can generate warning messages concerning portability, but it is no effort at all to write a non-portable program which generates no compiler warnings.

Why C remains popular

With advances in compiler technology, the original motivation for designing medium-level languages – namely, object code efficiency – has largely disappeared. Most other machine-oriented languages which appeared at about the same time as C are now considered to be obsolete. Why, then, has C survived?

There is of course a belief that C is more appealing to the "macho" side of programmers, who enjoy the challenge of struggling with obscure bugs and of finding obscure and tricky ways of doing things.

The conciseness of C code is also a popular feature. C programmers seem to feel that being able to write a statement like

```
**p++^=q++=*r---s
```

is a major argument in favour of using C, since it saves keystrokes. A cynic might suggest that the saving will be offset by the need for additional comments, but a glance at some typical C programs will show that comments are also considered to be a waste of keystrokes, even among so-called professional programmers.

Another important factor is that initial program development is perceived to be faster in C than in a more structured language. (I don't agree with this belief, and will return later to this point.) The general perception is that a lot of forward planning is necessary in a language like Modula-2, whereas with C one can sit down and start coding immediately, giving more immediate gratification.

Do these reasons look familiar? Yes, they are almost identical to the arguments which were being trotted out a few years ago in favour of BASIC. Could it be that the current crop of C programmers are the same people who were playing with toy computers as adolescents? We said at the time that using BASIC as a first language would create bad habits which would be very difficult to eradicate. Now we're seeing the evidence of that.

C is a medium-level language combining the power of assembly language with the readability of assembly language.

Advances in language design

It would be a gargantuan task to track down and document the origin of what we know today about programming language design, and I'm not going to do that. Many of the good ideas first appeared in obscure languages, but did not become well-known until they were adopted into more popular languages. What I want to do in this section is simply note a few important landmarks, as they appeared in the better-known languages.

Undoubtedly the most important step forward was the concept of a high-level language, as exemplified in Fortran and Cobol. What these languages gave us were at least three important new principles: portability of programs across a range of machines; the ability to tackle new problems which were just too big or too difficult in assembly language; and the expansion of the pool of potential programmers beyond that small group willing and able to probe the obscure mysteries of how each individual processor worked.

Needless to say, there were those who felt that "real" programmers would continue to work in machine language. Those "real" programmers are still among us, and are still arguing that their special skills and superior virtue somehow compensate for their poor productivity.

The main faults of Fortran were a certain lack of regularity, some awkward restrictions which in hindsight were seen to be unnecessary, and some features which were imposed more because of machine dependencies than for programmer convenience. (The only justification for the three-way IF of Fortran II, for example, was that it mapped well into the machine language of a machine which is now obsolete.) Some of these faults were corrected in Algol 60, which in turn inspired a large number of Algol-like languages. The main conceptual advance in Algol was probably its introduction of nesting in control structures, which in turn led to cleaner control structures.

The structured programming revolution is sometimes considered to date from Dijkstra's famous "GOTO considered harmful" letter. Although this is an oversimplification, it is true that the realisation that the GOTO construct was unnecessary – and even undesirable – was an important part of the discovery that programming productivity was very much linked to having well-structured and readable programs. The effect this had on language design was a new emphasis on "economy of concept"; that is, on having languages which were regular in design and which avoided special cases and baroque, hard to read constructs.

The important contribution of Pascal was to extend these ideas from control structures to data structures. Although the various data structuring mechanisms had existed in earlier languages – even C has a way of declaring record structures – Pascal pulled them all together in an integrated way.

Pascal can still be considered to be a viable language, with a large number of users, but it has at least two conspicuous faults. First, it was standardized too early, which meant that some niggling shortcomings – the crude input/output arrangements, for example – were never fixed in the language standard. They are fixed in many implementations of Pascal, but the repairs go outside the standard and are therefore nonportable. The second major fault is that a Pascal program must (if one wants to conform to the standard) exist as a single file, which makes the language unsuitable for really large programs.

More recently, there has been a lot of emphasis on issues like reusable software and efficient management of large programs. The key idea here is modularity, and this will be discussed in the following section.

Now, where does C fit into this picture? The answer is that C is built around lessons which were learnt from Algol 60 and its early successors, and that it does not incorporate much that has been learnt since then. We *have* learnt some new things about language design in the last 20 years, and we do know that some of the things that seemed like a good idea at the time are in fact not such good ideas. Is it not time to move on to D, or even E?

Real programmers can write C in any language.

Modularity

In its very crudest sense, modularity means being able to break a large program into smaller, separately compiled sections. C allows this. Even Fortran II allowed it. This, however, is not enough.

What modularity is really about is data encapsulation and information hiding. The essential idea is that each module should take care of a particular sort of data, and that there should be no way of getting at that data except via the procedures provided by that module. The implementation details of the data structures should be hidden. There should be no way to call a procedure unless the module explicitly exports that procedure. Most importantly, callers of a module should not need to know anything about the module except for the declarations and comments in its "visible" section. It should be possible to develop a module without having any knowledge of the internal structure of any other module.

The advantages should be obvious. At any given time a programmer need only be concerned with a short section of program – typically a few pages long – without having to worry about side-effects elsewhere in the program. It is possible to work with complex data structures without having to worry about their internal detail. It is possible to replace a module with a newer version – and this even includes the possibility of a complete overhaul of the way a data structure is implemented – without having to alter or re-check the other modules. In a team programming situation, the coordination problems become a lot simpler.

If the hardware supports memory segmentation, then the data in each module are protected from accidental damage by other modules (except to the extent to which pointers are passed as procedure parameters). This makes errors easier to detect and to fix. Even without hardware protection the incidence of programming errors is reduced, because error rates depend on program complexity, and a module a few pages long is far less complex than a monolithic hundred-page program.

Now, modular programming *is* possible in C, but only if the programmer sticks to some fairly rigid rules:

- Exactly one header file per module. The header should contain the function prototypes and `typedef` declarations to be exported, and nothing else (except comments).
- The comments in a header file should be all that an external caller needs to know about the module. There should never be any need for writers to know anything about the module except what is contained in the header file.
- Every module must import its own header file, as a consistency check.
- Each module should contain `#include` lines for anything being imported from another module, *together with comments showing what is being imported*. The comments should be kept up-to-date. There should be no reliance on hidden imports which occur as a consequence of the nested `#include` lines which typically occur when a header file needs to import a type definition or a constant from elsewhere.
- Function prototypes should not be used except in header files. (This rule is needed because C has no mechanism for checking that a function is implemented in the same module as its prototype; so that the use of a prototype can mask a "missing function" error.)
- Every global variable in a module, and every function other than the functions exported via the header file, should be declared `static`.
- The compiler warning "function call without prototype" should be enabled, and any warning should be treated as an error.
- For each prototype given in a header file, the programmer should check that a non-private (i.e. non-static, in the usual C terminology) function with precisely the same name has its implementation *in the same module*. (Unfortunately, the nature of the C language makes an automatic check impossible.)
- Any use of `grep` should be viewed with suspicion. If a prototype is not in the obvious place, that's probably an error.
- Ideally, programmers working in a team should not have access to one another's

source files. They should share only object modules and header files.

Now, the obvious difficulty with these rules is that few people will stick to them, because the compiler does not enforce them. A great many people think of `#include` as a mechanism for hiding information, rather than as a mechanism for exporting information. (This is shown by the distressingly common practice of writing header files without comments.) The counter-intuitive meaning of `static` is a disincentive for using it properly. Function prototypes tend to be thrown into a program in a haphazard way, rather being confined to header files. Programmers who think of comments as things to be added after writing the code will hardly accept the discipline of keeping the comments on their `#include` lines up-to-date. A good many programmers prefer not to enable warning messages in their compilations, because it produces too many distracting and "unimportant" messages. Finally, the notion of having precisely one header file per module runs counter to traditions which have been built up among the community of C users.

And, what is worse, it takes only one programmer in a team to break the modularity of a project, and to force the rest of the team to waste time with `grep` and with mysterious errors caused by unexpected side-effects. I believe it is well known that almost every programming team will include at least one bad programmer. A modular programming language shields the good programmers from at least part of the chaos caused by the bad programmers. C doesn't.

To complicate matters, it is easy even for good programmers to violate, by accident, the rules for proper modularity. There is no mechanism in C for enforcing the rule that every prototype mentioned in a header file is matched by an implementation in the same module, or even for checking that the function names in the implementation module match those in the header file. It is easy to forget to make internal functions private, since the default behaviour is back to front: the default is to make all functions exportable, whether or not a prototype is used. It is easy, too, to lose track of what is being imported from where, because the crucial information is locked away in comments which the compiler doesn't check. The only way I know of for checking what is being imported is to comment out the `#include` lines temporarily, to see what error messages are produced.

In most modular programs, some or all modules will need an initialization section. (You can't initialize data structures from outside the module, since they aren't supposed to be visible from outside the module.) This means that the main program of a C program must arrange to call the initialization procedures in the correct order. The correct order is bottom-up: if module MA depends on module MB, then module MB must be initialized before module MA. Any language supporting modular programming will work this out for you, and perform the initialization in the correct order. (It will also report circular dependencies, which in most cases reflect an error in overall program design.) In C, you have to work this out by hand, which can be a tedious job when there are more than about a dozen modules. In practice, I have found that it is almost impossible to avoid circular dependencies in a large C program, whereas I have rarely struck such dependencies in Modula-2 programs. The reason is that the Modula-2 compiler/linker combination catches circularities at an early stage, before it has become too difficult to re-design the program. In C, such errors do not show up until mysterious errors appear in the final testing.

The hazards of #include

I have often heard it said that the `#include` directive in C has essentially the same functionality as the `IMPORT` of Modula-2 and similar languages. In fact there is a profound difference, as I shall now attempt to show.

Consider a header file `m2.h` which contains the lines

Much of the power of C comes from having a powerful preprocessor. The preprocessor is called a programmer.

```
#include <m1.h>
   /* FROM m1 IMPORT stInfo */

void AddToQueue (stInfo* p);
```

and suppose that several other modules contain a `#include <m2.h>`. Consider the following sequence of events, which could easily happen in any programming project:

(a) some of the modules which import from `m2` are compiled;

(b) as the result of a design change, the `typedef` defining `stInfo` in `m1.h` is altered;

(c) the remaining modules which import from `m2` are compiled.

At this point, the overall program is in an inconsistent state, since some of the modules were compiled with an obsolete definition; but the error will probably not be caught by the compiler or linker. If you ever wondered why you keep having to do a "Compile All" in order to eliminate a mysterious bug, this is part of the reason.

The reason why this problem occurs in C, whereas it does not occur in languages designed for modular programming, is that a C header file is a pure text file, with no provision for containing a "last compiled" time or other mechanism for consistency checking. (This is also why C compilers appear to be painfully slow when compared with, for example, a typical Modula-2 compiler. It usually takes longer to read a header file than it does to read a symbol file.)

Another nasty consequence of reading the header file literally is that information in the header file is treated as if it were in the file which contains the `#include`. There is no "fire wall" around the header file. Everything declared in one header file is automatically exported, in effect, to the header files mentioned in every following `#include`. This can lead to obscure errors which depend on the order of the `#include` lines. It also means that the effect of a header file is not under the full control of the person who wrote it, since its behaviour depends on what comes before it in the importing module.

Similar problems exist with other preprocessor directives, such as `#define`. This point is not always fully understood: the effects of a `#define` persist through an entire compilation, including any included files. There is no way in C to declare a local literal constant.

Have you ever had the experience of having the compiler report an error in a library function you're not even calling, where the real error turns out to be a misplaced semicolon in some completely unrelated file? Such non-local effects make a mockery of modularity.

Another problem with `#include` is that it is an all-or-nothing proposition. (This can be resolved by having multiple header files per module; but that means putting the header files under the control of the importer, not the exporter, which creates the risk of undetected discrepancies between a module and its header file(s). In any case, such a practice creates major headaches in terms of book-keeping and naming conventions.) How many programmers read the *whole* of a header file before deciding to include it? Very few, I suspect. The more likely situation is that the `#include` imports some names which the importer doesn't know about. This can be a disaster if, as sometimes happens, two functions happen to have the same name and the same parameter types. (If you think this is unlikely, just think of the obsolete versions of software which are left around when you copy files from place to place.) The compiler won't complain; it will simply assume you were in an expansive mood and decided to write a function prototype twice. The linker might complain, but you can't guarantee it.

As a result, it is possible to import a function which is different from the function you thought you were importing, and there is not necessarily any warning message. Part of the problem here is that the mechanism by which the linker chooses which functions to link in has no connection with the mechanism by which the

By analysis of usenet source, the hardest part of C to use is the comment.

compiler checks function prototypes. There is no way of specifying that a particular header file belongs to a particular module.

Note, too, that an unused prototype is never picked up as an error. (This is another reason for insisting that prototypes be used only in header files, and nowhere else. This does not solve the problem, but it reduces the amount of manual checking which has to be done.) While this will not cause a program to run incorrectly, it adds to the confusion to be faced by future maintainers of the program.

The speed of program development

A claim that is often heard is that initial program development is fast in C because it is easy to get to the point of the "first clean compile". (This is also an argument which is popular with the BASIC enthusiasts.) This property is contrasted with what happens with languages like Modula-2, where – it is said – a lot of forward planning is necessary before any progress is made on the coding. The conclusion is that C programmers get more immediate feedback.

This argument is silly in at least three ways. First, the claim relies at least partly on the fact that C compilers are more generous in accepting doubtful code than are compilers for higher-level languages. Where is the virtue in that? If the compilation of code containing errors is seen as a significant step forward, you can get that in any language. All you have to do is ignore the error messages.

Second, the "first clean compile" is a fairly meaningless measure of how far you have progressed. It might be a significant milestone if you follow an approach to programming where the coding is not started until most of the design work has been completed, but not otherwise. Under the "code, then debug" philosophy of programming, you still have most of your work ahead of you after the first compilation.

Finally, my own experience is that even the original statement is incorrect. I find that I reach the "first clean compile" stage within the first few minutes of starting work. This is because I prefer developing programs through stepwise refinement (also known as top-down design combined with top-down coding). The very first thing I compile consists of perhaps half a dozen lines of code, plus a couple of dozen lines of comments. It is so short that it will compile without errors either immediately, or after discovering errors which are obvious and easy to repair.

What about the subdivision of a large program into modules? This takes a lot less forward planning than is commonly supposed. With stepwise refinement, and with the philosophy that the function of a module is to look after a data type, one tends to discover what modules are needed as the program development proceeds. Furthermore, true modularity makes it very easy to construct and test the program in stages, because of the property that changes in the internal details of a module can be made independently of what is happening outside that module.

In cases where I have kept a log of the time I have spent on a project, I have found that I spend about twice the time to get a C program working as to solve a problem of equivalent complexity using Modula-2. The difference has nothing to do with typing speed – since the source files tend to be of about the same length – but in the time spent in debugging. In Modula-2, the job is essentially complete once I have typed in the last module, and debuggers are rarely needed. In C, a good debugger is indispensable.

To a project manager, this is a very important factor. In a big project, the cost of paying the programmers is typically the second-biggest budget item (after administrative overheads), and sometimes even the biggest. A productivity difference of 50% can make the difference between making a large profit or a large loss on the project.

Pointers: the GOTO of data structures

Despite all the advances which have been made in the theory and practice of data structures, pointers remain a thorn in everyone's side. Some languages (e.g. Fortran, Lisp) manage to get by without explicit pointers, but at the cost of complicating the representation of some data structures. (In Fortran, for example, you have to simulate everything using arrays.) For anyone working with almost any reasonably advanced application, it is hard to avoid the use of pointers.

This does not mean that we have to like them. Pointers are responsible for a significant amount of the time spent on program debugging, and a large proportion of the complexity which makes program development difficult. A major challenge for software designers in languages like Modula-2 is to restrict the pointer operations to the low-level modules, so that people working with the software don't have to deal with them. A major, and largely unsolved, problem for language designers is to find mechanisms which save programmers the trouble of having to use pointers.

Having said that, one can also say that a distinction can be drawn between essential and inessential pointers. An *essential pointer*, in the present context, is a pointer which is required in order to create and maintain a data structure. For example, a pointer is needed to link a queue element to its successor. (The language might or might not explicitly call it a pointer, but that is a separate issue. Whatever the language, there must be some way of implementing the "find successor" operation.) An *inessential pointer* is one which is not needed as part of implementing a data structure.

In a typical C program, the inessential pointers outnumber the essential pointers by a significant amount. There are two reasons for this. The first is that C traditions encourage programmers to create pointers even where equally good access methods already exist; for example, for stepping through the elements of an array. (Should we blame the language for the persistence of this bad habit? I don't know; I simply note that it is more prevalent among C programmers than among those who prefer other languages.)

The second reason is the C rule that all function parameters must be passed by value. When you need the equivalent of a Pascal VAR parameter or an Ada **inout** parameter, the only solution is to pass a pointer. This is a major contributor to the unreadability of C programs. (To be fair, it should be admitted that C++ does at least provide a solution for this problem.)

The situation worsens when it becomes necessary to pass an essential pointer as an **inout** parameter. In this case, a pointer to a pointer must be passed to the function, which is confusing for even the most experienced programmers.

Execution-time efficiency

There appears to be a widespread belief among C programmers that – because the language is close to machine language – a C program will produce more efficient object code than an equivalent program written in a high-level language.

I'm not aware of any detailed study of this question, but I have seen the results of a few informal studies comparing Modula-2 and C compilers. The results were that the code produced by the Modula-2 compilers was faster and more compact than that produced by the C compilers. This should not be taken as a definitive answer, since the studies were not extensive enough, but it does indicate that C programs might not be as efficient as is generally thought.

I believe I've also seen claims – although I can't recall the details at this distance in time – that C compilers produced better code than an assembly language programmer did. I observed a similar phenomenon when testing my SGL compiler many years ago. The reason in that case seemed to be that the compiler did a reasonably good job on things like register allocation, whereas one can suffer from lapses of concentration when having to concentrate too much on the fine detail.

C++ will do for C what Algol-68 did for Algol.

The general rule seems to be that a high-level language compiler will out-perform a lower-level language compiler, mainly because the high-level language compiler has more scope for making decisions about how to generate the code. If you do things in C like setting up a pointer to an array rather than using subscripts, you are taking that decision away from the compiler. Your approach *might* produce more efficient code, but to be sure of that you have to know quite a lot about the instruction timings on the machine you are using, and about the code generation strategies of your compiler. In addition, the decision is a non-portable one, potentially leading to major inefficiencies if you switch to another machine or another version of the compiler.

More importantly, the speed of a program tends to depend more on the global strategies adopted – what sort of data structures to use, what sorting algorithms to use, and so on – than the micro-efficiency issues related to precisely how each line of code is written. When working in a low-level language like C, it becomes harder to keep track of the global issues.

It *is* true that C compilers produced better code, in many cases, than the Fortran compilers of the early 1970s. This was because of the very close relationship between the C language and PDP-11 assembly language. (Constructs like `*p++` in C have the same justification as the three-way IF of Fortran II: they exploit a special feature of the instruction set architecture of one particular processor.) If your processor is not a PDP-11, this advantage is lost.

What about C++?

The language C++ is supposed to overcome some of the faults of C, and to a certain extent it does this. It does, however, have two major drawbacks. It is much more complex than it needs to be, which can lead programmers either to ignore the extended features or to use them in an inappropriate way. The second problem is that the language tries to maintain compatibility with C, and in so doing retains most of the unsafe features.

This second problem means that most of the faults discussed in earlier sections are also faults of C++. Type checking is still minimal, and programmers are still permitted to produce weird and baroque constructs which are hard to read. Strangest of all, there is still no support for modularity beyond the crude `#include` mechanism. This is a little surprising: modular programming and object-oriented programming complement each other very nicely, and given all the effort that the designers of C++ must have had to put into the object-oriented extensions it is rather disappointing that they did not put in that slight extra effort which could have resulted in a major improvement to the language.

Some of the features related to object-oriented programming are complicated, and open to misuse by programmers who do not fully understand them. For safety, I would prefer to see programmers learn object-oriented programming using a cleaner implementation (e.g. Smalltalk, Modula-3) before being let loose on C++.

The ability to pass function parameters by reference is a definite bonus, but the mechanism chosen for doing this is unnecessarily messy. The only motivation I can see for implementing it this way is to satisfy the Fortran programmers who miss the `EQUIVALENCE` construct.

Operator and function overloading is a mixed blessing. In the hands of a competent programmer it can be a major virtue; but when used by a sloppy programmer it could cause chaos. I would feel happier about this feature if C++ compilers had some way to detect sloppy programmers.

If there is a discrepancy between a function and its prototype, is this an error, or is it a deliberate overloading? Most commonly it will be an error, but in some such cases the C++ compiler will make the optimistic assumption. One has to be a little suspicious of a language improvement which increases the probability of undetected errors.

I'm not yet sure how to feel about multiple inheritance. It is powerful, in the same way that `goto` is powerful, but is it the sort of power we

Adding object orientation to C is like adding air conditioning to a bicycle.

want? I have a nagging suspicion that at some time in the future our guidelines for “clean programming” will include a rule that object inheritance should always be restricted to single inheritance. However, I’m prepared to admit that the evidence is not yet in on this question.

In brief, C++ introduces some new problems without really solving the original problems. The designers have opted to continue with the C tradition that “almost everything should be legal”. In my view, this was a mistake.

Libraries vs. language features

One of the popular features of C++ is the large set of library functions which is usually distributed with it. This is, indeed, a desirable feature, but it should not be confused with the inherent properties of the language. Good libraries can be written for any language; and in any case most reasonable compilers allow one to call “foreign” procedures written in other languages.

More generally, people sometimes say they like C because they like things like `argc` and `argv`, `printf`, and so on. (I don’t – I’ve had so much trouble with `printf`, `sscanf`, and the like that I’ve been forced into writing alternative I/O formatting functions – but that’s a separate issue.) In many cases, the functions they like, and point to as examples of “portable C” are peculiar to one particular compiler, and not even mentioned in whichever C standard they consider to be the standard standard. The desirability or otherwise of various library routines is a legitimate subject for debate, but it is an issue separate from that of language properties.

There is just one way in which these functions differ *as a result of genuine language differences* from the procedures available with other languages, and that is the C rule which permits functions with variable numbers and types of parameters. While this feature does have certain advantages, it necessarily involves a relaxation of type checking by the compiler. I personally have wasted hours of valuable debugging time over things like printing out a `long int` with a format appropriate to an `int`, and then not being able to discover why my computations were producing the wrong value. It would have been much faster, even if slightly more verbose, to call type-safe procedures.

Concluding remarks

Nothing in this document should be interpreted as a criticism of the original designers of C. I happen to believe that the language was an excellent invention for its time. I am simply suggesting that there have been some advances in the art and science of software design since that time, and that we ought to be taking advantage of them.

I am not so naive as to expect that diatribes such as this will cause the language to die out. Loyalty to a language is very largely an emotional issue which is not subject to rational debate. I would hope, however, that I can convince at least some people to re-think their positions.

I recognise, too, that factors other than the inherent quality of a language can be important. Compiler availability is one such factor. Re-use of existing software is another; it can dictate the continued use of a language even when it is clearly not the best choice on other grounds. (Indeed, I continue to use the language myself for some projects, mainly for this reason.) What we need to guard against, however, is making inappropriate choices through simple inertia.