GOING TO Ceed?

P.H.Welch, Computing Laboratory, University of Kent

Introduction

These notes are a contribution to the debate on the initial programming language(s) to be taught to first year undergraduates. Specifically, they address the issue of the teaching of sound software engineering principles in the context of ‘‘classical’’ sequential procedural programming. They do not question the merits of this context, but assume that its importance is accepted (i.e. this is not an entry into the functional programming contest).

My major concern is over the suggestion that C should replace Pascal as the supporting language for these primary courses. Language has a major impact on the way we think and none more so than the first programming language we learn. It is my belief that C would have a bad effect on our students if it were encountered before proper software engineering principles have become engraved on their hearts.

C is a free-wheeling language that imposes few disciplines on programmers and allows them to do whatever they want — that is why so many people like it! However, this freedom is only useful if used wisely, otherwise all that happens is that we dig ourselves a big hole and jump in! Sadly, C seems to encourage us, through many of its features, to start digging that hole and undermine the very software engineering foundations we should be trying to establish.

Like all lower-level languages, C should only be used by good experienced programmers. It should not be used by novices to gain that experience — they will only gain frustration and some very unhelpful habits!

The rest of this paper attempts to explain my particular problems with C.

Statements or Expressions?

In procedural programming, we have explicit access to the ‘‘state’’ of the system being modelled. This state is represented by modifiable ‘‘variables’’, whose ‘‘values’’ make up the components of the state. In general, the system we are programming is not isolated but will need to communicate with other systems (e.g. file stores, users, ...).

There are just two fundamental activities in this kind of programming :-

Evaluate: What is the current value of this variable (i.e. state component)? More generally, what is the value of this (algebraic) expression (consisting of state variables and constant values)? This activity does not involve any change of system state.

Update: Change the (global) system state. Directly, by modifying the value of a variable — either by explicit assignment of the result of an internal evaluation, or by explicit input from an external system. Indirectly, by explicit output to an external system (which changes the state of that system and, hence, the global environment).

Pure functional programming only supports the concept of evaluation. The ordinary mathematical laws of algebra are obeyed by its expressions. This makes system design — both at a formal and informal level — much clearer. Hence, we have a greater chance of attaining several important software engineering goals: simplicity, verifyability and maintainability. Further, because the system algorithms obey rich mathematical laws, there is the opportunity to apply semantic-preserving transformations so as to tune for various criteria of efficiency.

The procedural programming language occam strictly separates the activities of evaluation and state update. As a result, its expressions are open to the same simple mathematical design, algebraic laws, analysis and transformations as those for pure functional programming. Further, the semantics of state update, because it is divorced from evaluation, also obeys a rich set of mathematical laws and has a simplicity that matches our intuitive feel for this mechanism — see [0, 1].

[Aside: I sometimes wonder whether the functional programming community, in their proper enthusiasm to obtain languages to which simple (mathematical) reasoning could be applied, did not throw out the baby with the bath water when they abandoned all procedural concepts. For ''real'' applications like operating systems or industrial process control, it seems to be necessary — even for functional programmers — to come to terms with the notion of system state! Yet, well-designed procedural languages (such as occam), that treat procedural mechanisms and functional evaluation as orthogonal features, show that these notions can all the time be reconciled with simple clean semantics.]

''Classical'' procedural languages (like FORTRAN,Pascal, C and Ada) allow these notions to become mixed. The usual place in which this happens is during the execution of bodies of function calls, where state can be changed (through input/output or assignment to global data structures) during expression evaluation.

[Aside: functions in occam are mathematical. They cannot change external state, maintain internal state (through ''static'' variables) or perform any input/output. They always return the same result when given the same arguments. Abstract data types that require internal state (such as a random number generator) are implemented via parallel processes — not functions.]

For most procedural languages, this is the only loophole whereby state update and evaluation can become entwined. C, however, positively boasts of merging these ideas and provides not only apparently ''algebraic'' operators (like ++) that cause state change, but also values for assignment statements precisely so that they may be embedded in expressions. Indeed, C does not really admit to the notion of assignment (=) as a statement, but pretends that it is an operator (like == or +).

The trouble with all this is that what appear, on the surface, to be algebraic expressions have a semantics that do not obey the laws of algebra! Instead, expression semantics become entangled with state update semantics and the resulting increase in complexity is of at least one order of magnitude.

Some Examples

Suddenly, we have to become concerned with the order in which the compiler chooses to have the evaluation performed. In C, the order of evaluation is not defined and so we are able to write quite legal programs whose meaning is unknown. At this point, we are dead and so, also, should be the language. What is the meaning of :-

```
x = a[i] + b[i++];
```

or :-

```
/* check the ordering on the next two characters */
test = (getchar() > getchar ());
```

If E is an integer expression, why is (E - E) not necessarily zero?

Why is the assignment ''operator'', =, nearly always interchangeable with the equality operator, ==, without making the program illegal (but having dramatic, but mysterious, effects on its behaviour)?

I am afraid I do not like compactness of coding for its own sake, if it results in obscurity. Hence, pinching some (more) examples from [2, 3], I worry about :-

```
total_cost = (area = width*height)*unit_cost - discount;
```

and much prefer :-

```
area = width*height;
total_cost = (area*unit_cost) - discount;
```

I tend to despair when confronted with :-

```
while (*str_1++ = *str_2++)
  ;
```

for copying one string on to another (assuming sufficient room). Rewriting this code so that its semantics becomes plain — an activity which otherwise has to be done in our heads (in natural language) when trying to understand the above — it becomes :-

```
copying = TRUE;                          /* a #defined constant */
while (copying)
{
  *str_1 = *str_2;
  copying = (*str_2 != ASCII_null);
  str_1 += 1;
  str_2 += 1;
}
```

The execution speed of the above should be the same as for the first version (apart from some slight extra code to represent properly the Boolean nature of the condition — see later).

The original code obscures the fact that the state updates straddle the decision as to whether the loop should be continued. The disguising of this semantic complexity by the original, apparently simple, coding does not contribute to any of the goals of software engineering. It is definitely not an example of what is meant by ‘‘information hiding’’!

These are not, to me, minor complaints with only marginal impact on the way we develop systems. The obscure and complex semantics that is their result overloads our capacity to comprehend and control — it is one of the root causes of software mis-engineering. To combat this in our teaching, some serious guidelines have to be enforced.

Some Guidelines

First, though, we need a couple of definitions so that we can separate the algebraic notion of ‘‘function’’ from the procedural notion of ‘‘procedure’’. C tries hard to make us view these concepts as the same thing. We define a C routine that contains a return statement as a ‘‘function’’ — otherwise, we categorise it as a ‘‘procedure’’.

Guideline 0:	A function should always declare the type of result it returns — no int default. A procedure should always be declared as having the ‘‘type’’ void.

That, at least, will give a hint to the compiler that it should prevent the use of procedures in expressions!

Guideline 1:	Within a function body, never change the global system state. [For instance, no input/output, no assignment to global variables or pointed-at objects, no static variables, ...]

Guideline 2:	Never use the ‘‘operators’’ ++ or - -.

Guideline 3:	Never use the assignment ‘‘operator’’, =, in an expression.

Guideline 1 is very difficult to adhere to since so many of the standard library routines (e.g. getchar, malloc, fork, ...) violate them. There are two ways out. Either provide a ‘‘let-out’’ clause :-

Guideline 1.0:	If you violate guideline 1, only one such ‘‘function’’ may be used in any ‘‘expression’’.

Or (and I would prefer) provide a different set of ‘‘local standard’’ library routines that obeyed guideline 1 — i.e. the routines would be procedures, not functions. With great trepidation, I would carefully use the standard ‘‘functions’’ in their implementation — e.g. :-

```
void UKC_fork (pid_pointer)
int *pid_pointer;
{
  *pid_pointer = fork();                /* follows guideline 1.0 */
}
```

Then, the user would write :-

```
int pid;
...
UKC_fork (&pid);
...
```

This requires :-

Guideline 1.1:	Do not use the standard library ‘‘functions’’. Use the corresponding local library ‘‘procedures’’.

I do not think these guidelines would be tolerated by existing C programmers and I do not think we could enforce them for very long on our students. In Pascal or Ada, only guideline 1 is relevant and there are no standard library functions that violate it — the guideline could, therefore, be enforced.

A Few Other Little Difficulties — Booleans

Booleans are the most primitive data type in computing — the building block for all other structures. Yet C provides no direct support for them! Instead, there is an arbitrary mapping on to integers (zero and non-zero) for TRUE and FALSE (or is it the other way around — who can remember?).

We define a ‘‘Boolean variable’’ as one that is only to be used to hold (integer representations of) Boolean values. Use #define to give you the Boolean constants TRUE and FALSE. We define a ‘‘Boolean expression’’ as one that would yield a Boolean value in Pascal — i.e. one made up from Boolean variables, Boolean constants, Boolean operators (between Boolean expressions) and arithmetic comparison operators (between ‘‘integer expressions’’ or ‘‘real expressions’’). Define an ‘‘integer expression’’ as one that would yield an integer (or character) value in Pascal. A ‘‘real expression’’ is one that would yield an real value in Pascal.

Guideline 4:	Document ‘‘Boolean’’ variables when they are declared (usually as integers). Never use ‘‘integer’’ or ‘‘character’’ expressions for conditions, use ‘‘Boolean’’ expressions. Maintain by yourself strong typing rules to keep ‘‘Boolean’’ and ‘‘integer’’ items distinct.

Thus, in an earlier example, we used (*str_2 != ASCII_null) instead of just *str_2.

This mixing up of Booleans with integers and characters is as bad as mixing up procedures with functions, or state update with expression evaluation.

For Loops

Once upon a time, Bill and Ben were designing a programming language. This, however, was not what they really enjoyed doing — their first love was writing syntax analysers. They used to have great fun in setting each other more and more complex problems. One day, Ben was feeling particularly pleased with himself — he had just invented the C ‘‘for-loop’’ :-

‘‘OK, Bill’’, said Ben, ‘‘bet you can’t parse this!’’

Unfortunately, Bill (who was too clever by half) managed it — at least, no counter-example was found during testing. They decided to incorporate it into their language on the grounds that it would make life harder for third-party compiler vendors.

To justify its inclusion, the myth was generated that it was more powerful than the Pascal for-loop. In actuality, it was not comparable. A for-loop is a mechanism for iterating a section of code through a pre-determined sequence of values. A for-loop always terminates. C does not have a for-loop!

The C ‘‘for-loop’’ is a funny kind of while-loop, with syntactic structure for supplying some ‘‘initialisation’’ code (before loop entry) and ‘‘increment’’ code (at the end of each loop). It is not exactly the same as an ‘‘equivalent’’ C while-loop, since a continue statement in the loop body behaves slightly differently.

I do not believe that :-

```
for (sum = 0, num = 1; num < odd; num += 2)
{
  sum += num;
}
```

is any easier to read, write or understand than :-

```
sum = 0;
num = 1;
while (num < odd)
{
  sum += num;
  num += 2;
}
```

which at least has the merit of writing out the code in the same order that the code is executed! Further, the object code produced from the second version should be just as tight as that produced from the first.

We have no guarantee of termination from a C ‘‘for-loop’’. There is nothing to stop us altering the variables used in the termination condition during the loop. Thus, even if we restrict our use of the ‘‘for-construct’’ to set up a sequence of values for iteration, we have to inspect not only the header line but also the body of the loop to see if those iterations will actually take place. The security benefits of a proper for-loop are non-existent in C.

Pascal (and C itself) can get the same flexibility from a while-loop as C can get from its ‘‘for-loop’’. A decent compiler will not cause loss of efficiency in the object code.

[Aside: it was no oversight that Pascal only allowed increments by a single step (up or down). Much agonising went into this decision — after all, Algol_60 had allowed arbitrary (but fixed) step sizes. Ada for-loops follow exactly the same rules as those for Pascal. Occam finally corrects all this gross laxity and only allows positive increments by 1.]

Guideline 5:	Never use the C ‘‘for-loop’’. Use a while-loop.

Do-While Loops

Occam's Razor is :-

> ''Entia non sunt multiplicanda praeter necessitatem''

All engineers must respect this — not least software engineers. One interpretation is that if a machine doesn't need a gizmo, it is necessary not to put one in.

The do-while-loop can always be re-written as a while-loop, quite often with an improvement in clarity! The number zero is often thought to be inferior to other numbers. This is not a good thought — it leads to silly end errors. Too often, we assume that some repetitive activity must occur at least once. Why does C, that sensibly indexes its arrays from zero, allow this?

Guideline 6:	Never use the do-while-loop. Use a while-loop.

[Aside: of course, we should never use the repeat-loop in Pascal.]

Break Statements

This destroys the guarantee that when a while-loop terminates, the while-condition is false. The semantics of the loop are lost.

Guideline 7:	Never use a break-statement to exit from a loop.

The use of break-statements to exit the switch-construct is sad but unavoidable. It could have been avoided if the switch-construct had a more conventional semantics — i.e. no ''fall-through''.

Guideline 8:	Always use a break-statement to terminate each non-empty clause of a switch-construct — never ''fall through'' to the next case.

Continue Statements

I cannot think of a good reason to ban this, but remain suspicious.

Guideline 9:	Do not use continue-statements gratuitously.

Return Statements

Embedded inside loops, these destroy their semantics in the same way as breaks.

Guideline 10:	Never use a return-statement within a loop. For preference, a return-statement should always and only be used as the syntactically last statement of a function body.

Goto Statements

Despite all its loop constructs, its break, continue and return statements, C does not have the confidence to deny itself the goto! We shall!!

Guideline 11:	Never use the goto statement.

Security Checks on Routine Calls

There aren't any! A C routine may be called with any number of arguments of any type, regardless of the parameter structure in its definition!! The semantics of mis-matched calls is undefined. Is this tolerable in a teaching language? Is this tolerable in a language for (safety-critical or finance-critical) applications?? What sort of example does this set for impressionable students???

Guideline 12:	Check the types and number of arguments in every routine call against the corresponding routine specification. Check also the result type of functions and that they are used correctly. Do this at least three times.

Ideally, a tool should be used. The compiler should have been that tool. How good is lint at this job?

Returning Array Structures from Functions

You can't. Neither can you pass arrays to routines as parameters. Array names in C are just pointers to an undefined structure — there is no way to de-reference them to the array value.

C boasts of having a more complete semantics than Pascal. The type of result returnable from a function is the usual illustration. In Pascal, only atomic data types may be returned but, in C, anything can be. Anything, that is, except an array!

The trouble is that an array is a second-class object in C, which is somewhat unfortunate for such a fundamental data-type.

Guideline 13:	Do not declare array objects directly. Turn them into first-class objects by hiding them inside a struct.

For example, instead of :-

```
int A[100];

... A[i] ...
```

we will have :-

```
typedef struct { int array[100]; } int_100;

int_100 A;

... A.array[i] ...                    /* object code same as before */
```

''Array'' values may now be passed to and returned from routines. Pointers to these ''array'' structs will work exactly as pointers to raw arrays — so the usual C tricks still work. The real benefit is that ''arrays'' are now first-class objects. For instance :-

```
int_100 A, B;
...
A = B;                                /* copies array B on to A */
...
```

This also removes another anomaly from the language. Array names declared as ''variables'' are in fact constants and may not be assigned (or incremented). Array names declared in parameter lists are in fact pointer variables and may be assigned (or incremented)! Our new ''array'' variables may be assigned, passed to and returned from routines as ordinary objects.

Of course, we are now left with Pascal's problem that the size of an array object is part of its type. This attempt to promote arrays to first-class objects leaves us in a logical straight-jacket. However, since C never checks types on parameter lists, some of us may be tempted to ignore this little difficulty. This is not a happy scene. Let us move on.

Array Bounds Checking

There isn't any! Array bound violation is an extremely common mistake when learning to program. Without automatic run-time checking, students will have grave problems in spotting those bugs.

Actually, array bound violation is probably a common mistake in most production software (e.g. operating systems, window managers, network interconnections, flight-control, navigation, ...). It's just that by the time people get around to writing ''real'' systems software, they have long discarded ''nanny'' languages like Pascal and write in ''efficient'' and ''usable'' languages like C, where their errors go unreported. If you are lucky, when the crash happens, the evidence is destroyed as well!

The time is fast coming when software engineers, like civil engineers, will be held legally responsible for the consequnces of their mistakes — the complexities of the failed system being no defence. Will we then be quite so keen to work with C?

Guideline 14:	Array indexing (and pointer incrementing) is dangerous. Check, re-check and re-re-check. When you think you have got it right, check again with a friend.

[Aside: in occam, array values and array references may be freely passed to routines. Array parameters need not specify their size. Despite this, array bounds checking is always performed — either by the compiler or by run-time code.]

[Aside: occam array values can always be passed by reference (no copying), since the language rules ensure that the routine does not alter them — i.e. we get both efficiency and security. Occam reference parameters do not yield dangerous aliases for data objects (such as proliferate in C), since the language rules only allow the routine to access the referenced object via its parameter name and not its global name. Thus, reference parameter semantics are independent of implementation mechanism — 'value/result', 'direct address reference' or 'name' all yield the same answers. This eliminates a whole class of subtle bugs. I know of no other procedural language with this level of security. These rules are fully enforced by the compiler.]

[Aside: at this point it must be confessed that occam functions cannot return array values — but then, occam does not (yet) claim to be a complete language!]

Operator Precedence and Associativity

There are far too many operators ever to remember their precedences. Worse still, some associate to the left and some to the right — who knows which?

Guideline 15:	Never rely on operator precedence or association rules to describe an expression. Always use brackets.

[Aside: if you are designing a programming language, do not try to define such precedence — you will never get people to agree with you! Look at the mess that was made in Pascal!!]

Beware also that computable operators for arithmetic (like +) are not associative. For instance, it is not always true that (a + (b + c)) is equal to ((a + b) + c). We should always use fully bracketed forms to express what we mean and never write (a + b + c).

On the other hand, computable operators for Boolean algebra are normally associative. However, C has a ''short-circuit'' semantics for Boolean operators that destroys this mathematical associativity. This re-introduces procedural concepts into expressions that would otherwise have a ''pure'' evaluation semantics.

Guideline 16:	Never exploit the ''short-circuit'' semantics of && and ||.

Always assume that both operands are evaluated. If this might lead to a memory violation and core-dump, go away and re-design the loop properly.

See [4] for further comments on this.

Automatic Type Coercion

C allows all its integer and floating-point types to be freely mixed in expressions, assignment statements and routine calls. The compiler must generate code automatically to convert not only between differing basic types, but also between their various levels of precision. Arithmetic operations are always carried out at the highest level of precision.

This again leads to security, portability and semantic problems. A classic example is that :-

```
char ch;                    /* (say) 8 bits */
int i, j;                   /* (say) 32 bits */
...
ch = 255;
i = ch;
```

sets i to -1 on some systems and to 255 on others!

Cavalier disregard for range errors during coercion is also risky. For example :-

```
short s;                    /* (say) 16 bits */
...
s = 40000
```

sets s to -25536 (on our VAXenand Orions) instead of raising an overflow signal. Such a signal would have to be generated by software on most machines. Nevertheless, coercion (whether automatic or explicit) ought to check the validity of its operation. C makes no such requirement.

[Aside: disregarding run-time errors at any time (the default attitude taken by most C systems) is always dangerous. If (i == 1073741824) and (j == -1073741824), then (i > j) would be ‘‘true’’ but ((i - j) > 0) would be ‘‘false’’. Simple code transformations no longer preserve semantics!]

Even when no range violations occur, automatic coercion between precisions still cause semantic break-down. For instance :-

```
float a, b, x, y, z;        /* (say) 32 bits */
...
z = b*x;
a = (b*x) + y;
```

The assignments may not be transformed (by the programmer or a clever semantic-preserving optimising tool) into :-

```
z = b*x;
a = z + y;
```

or, as the C hacker would prefer (but we ban) :-

```
a = (z = b*x) + y;
```

The last two versions, of course, after multiplying at double-precision, coerce the intermediate result down to single-precision (losing information) before returning to double-precision for the final addition. The original code performed all the arithmetic in the computation of a at double precision..

We have to be able to reason (formally or, at least, informally) about the programs we write in any language. Yet, ‘‘obvious’’ transformations are invalid with C’s rules for arithmetic. Once again, simple semantics has been destroyed by decisions that were presumably taken to make life ‘‘easier’’ for the programmers and the results ‘‘more accurate’’!

Instead, the reverse is true. The programmer has to become very aware of what the compiler does as the compiler ‘‘interprets’’ the programmer’s intentions. Both parties play a game of guessing what each other is guessing. Why don’t they just tell each other?!!

[Aside: in occam, operands on either side of an operator must have the same type (including precision level) and the result of the operation has that same type (including precision level). The same strong type checking is enforced for assignment and parameter passing. If type coercion is necessary, the programmer must provide it explicitly. The programmer states exactly what he or she wants — the compiler obeys. Overflow during coercion is checked and an error flag is raised if detected, stopping that process. The security, portability and semantic problems of C arithmetic do not exist. The results are computed to the accuracy stated by the programmer and optimising program transformations work!]

As David Barnes says in [2], in relation to syntactic difficulties regarding semi-colons and curly brackets :-

> ''It is unfortunate that a complete grasp of the syntactic rules for programming languages often requires the programmer to have taken a course in basic compiling techniques.''

This is not really unfortunate — it is a result of bad language design. Language semantics must be defined independently from implementation techniques. Otherwise, semantic complexities make the language a very dangerous instrument. Quoting Hoare [5] :-

> ''If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of the solution''

For further discussion of these points, see [6, 7].

The Macro Pre-Processor and the Proper Use of Tools

This is an insecure, but quaint, tool. Its main use seems to be for providing named constants and ''in-line'' routine definitions. Other languages provide the former as part of the language and the latter either through explicit compiler directives or implicitly through optimising compilers.

The problem with this pre-processor is that it is a general one and has no knowledge of C. Therefore, the user of the tool has to ensure that its use will not violate the syntactic and semantic rules of C (such as they exist!) or else debug what the pre-processor has accepted as perfectly valid macros.

This is bad engineering. We should not use tool A in conjunction with tool B, if A can legally introduce effects that B cannot handle. For tools that properly fit together, a mistake in the use of one tool should be reported by that tool — not by another one. The other tool will not have the information needed to provide meaningful diagnostics and has to give misleading ones!

The C pre-processor will accept macros that generate illegal syntax — even turn legal programs into illegal ones! This is not a secure combination of tools!! If we wish to use such a general macro pre-processor for C source text, its macros should be filtered through another C-specific tool that validates them first.

Guideline 17:	Write a filter to check the validity of #define macros. Never use macros without it.

[Aside: The generic parameter concept within Ada provides a secure way of abstracting on language entities (e.g. constants, variables, types, procedures, ...). Use of the abstractions (c/f ''macro calls'') is checked (and compiled!) by the compiler without reference to their implementations. Implementation of the abstractions (c/f ''macro definitions'') is checked (and compiled) by the compiler without reference to their use. A change in implementation does not imply re-compilation of application — and vice-versa. This separation of concerns is good software engineering with major benefits both for programming ''in-the-small'' and ''in-the-large''.]

Concurrency and Other Language Extensions

The special routines (fork, vfork, exec, signal, setjmp, longjmp, malloc, free, ...) introduce new semantics into the language. They are, therefore, language extensions despite being disguised as ‘‘ordinary’’ routines — try and write them yourself just using ‘‘ordinary’’ C! Their semantics are defined by a lot of natural language and hand waving — does anyone know of any formal definitions?

It is dangerous to use something that nobody properly understands. I suppose it is OK for University operating systems, but what would happen if they were used in something really important — like an avionics system or a bank???

It would be hard to ban the use of malloc in C (which is really an unstructured version of new in Pascal or Ada), but I really would object to the concurrency primitives. There are much simpler means for teaching parallel processing and communications for which the formal semantics is precisely understood (... occam again!)

Debugging and All That

C just cares about getting results — any results will do! Its semantics are so weak (writing = in place of ==, and vice-versa, is generally acceptable), perforated (i.e. many syntactically legal programs have no meaning) and complex (e.g. the impact of automatic type coercion, the operator ++, ...) that the programmer only has limited and somewhat fuzzy control over what he or she writes. For the same reason, the compiler and run-time system can do little to help — ‘‘syntax error: line <n>’’ or ‘‘program error: core dumped’’ is about all we get! Is this what we want to give beginners?

When things don’t work (i.e. sooner rather than later), we resort to manual debugging or, even worse, a full-blown multi-window breakpoint-setting variable-modifying core-dump-analysing interactive debugger. We should never write anything whose meaning we do not understand. Debuggers are for programmers who do not understand what they write. Debuggers are for C programmers!

David May (of INMOS) tells a story about a C programmer acquaintance — we shall call him Chuck. Chuck always starts every new project by running his hello.world program — this is now fully debugged. ‘‘Hello world’’, says the program. ‘‘Hello’’, says Chuck, ‘‘that’s not supposed to happen!’’. Chuck now has his excuse to enter the debugger, where he stays for the rest of the project, hacking away at his code until it approximates the required specification. Chuck is an experienced C programmer — he knows too well its little surprises. He is only really comfortable when working from inside the debugger where he can get immediate feedback on what his creation actually means. Chuck never puts pen to paper or consults the reference manual — there is little point. When developing C, it is best just to interact with the machine — thinking comes later!

Higher Level Software Engineering

The principles of abstraction, information hiding, high cohesion, low coupling, completeness and consistency, together with supporting techniques such as modules, abstract data types, abstract state machines, parallel processing and object oriented design are not acquired simply through the ability for separate compilation. Yet those principles and techniques are necessary in order to achieve the goals of software engineering (i.e. simplicity, correctness, efficiency, modifyability, portability, reusability, ...) for ‘‘large’’ systems.

C offers us only separate compilation at this level, together with a simple flagging mechanism for indicating which variables or routines are being exported (you mark those that are not for export as being static). The exporting mechanism is even incomplete since typedef, struct, union and enum definitions cannot be exported!

For programming ‘‘in-the-large’’, C offers little software engineering support. Separate compilation may be used with Pascal (at least, it can with our pc compiler) and achieves as many benefits as can be claimed for C.

Common Pascal extensions (e.g. UCSD or Turbo Pascal) provide reasonable language support for separate compilation, including separation of the exported interface from its implementation. The whole thing is done properly in Modula–2 and Ada. In particular, Ada provides direct mechanisms for abstract data types (whose instances, of course, are abstract state machines) and good support for object oriented design.

Ada

These notes outline some concerns about using C for teaching software engineering. My worries about Ada as a practical engineering tool for large-scale safety-critical applications would form a much longer document. However, for teaching basic software engineering, Ada could — with some careful censoring — be very good.

Ada was mostly designed in the late 1970s. On the one hand, it integrates in a consistent style all the best lessons that had at that time been absorbed about software engineering. For example, there are good control and flexible data structures, initialising declarations, structured constants, strong polymorphic (but only first order) type checking, abstract data types, secure separate compilation and an integrated library mechanism (independent of operating system). All these fundamental aspects, it does rather well — certainly much better than Pascal and much much better than C.

On the other hand, Ada experimented with several emerging techniques for software engineering that were definitely not mature (e.g. exception handling, generic parameters, parallel processing, ...). It is in these areas where many of the decisions that were made seem now (after 8 years experience) to have damaging consequences. Like any good speculative research tool, valuable lessons have been learned from this process and we could now design a much better language.

This does not detract from the practicality of the language so long as we restrict ourselves to the secure parts. Ada is used in the ‘‘real’’ world and, given the political and financial push that is still behind it, is likely to overtake FORTRAN and C in terms of how much software is commissioned. I am concerned that the wrong parts will be used and that Ada will thereby escalate the software crisis. The problem is that the Ada industry, having invested so much in the current Ada standard, will not tolerate the corrections and simplifications necessary to make the whole language secure. I hate standards — they always perpetuate design errors!

In the context of teaching basic software engineering (over, say, the first four terms), Ada would be a better vehicle than Pascal or C.

Occam

In contrast to Ada, occam did not incorporate any features whose software engineering benefits were speculative. Occam features are secure both for parallel and sequential programming. None of the guidelines in this document are needed since they (plus a few more) are already required by the language definition and enforced by the compiler. Occam will only allow us to play safely — which is why some C programmers give up on it so quickly!

For the moment, I would not want to advocate occam as our prime language for software engineering, because it currently does not support some major areas (e.g. record structures, pointers, user-defined data-types and recursion). On the other hand, it does offer high-level and secure support for concurrency, communications, user-defined protocols, non-determinism and interrupts! It also executes rather quickly!!

[Aside: there are good reasons for the present omissions from occam of ‘‘well understood’’ features — they are only ‘‘well understood’’ in a sequential discipline. In a parallel environment with finite resources (i.e. the ‘‘real world’’), they raise serious efficiency challenges and security problems (i.e. undefined behaviour). Many people are resorting to various ‘‘parallel’’ extensions of C to acquire these features. But this does not, in some magic way, evade those problems. They are still there — although perhaps C programmers, having long grown accustomed to living with ambiguity, do not notice!]

Concluding Remarks

If we had the resources, I should like to see a secure subset of Ada used to support the first year software engineering courses.

Failing that, I should like to use Modula–2 — again if a suitable compiler/interpreter were available.

Failing that, we should continue to use Pascal.

On no account, should we use C during the first year.

C++ is an attractive thought. There is no reason why object-oriented principles should not be taught as part of elementary programming practice. Unfortunately, it is badly compromised by relying on C for its basic actions. [The name of the language also suggests that its designers were not too concerned about encouraging secure semantics!]

When C is eventually taught, we should impose a set of guidelines for secure semantics such as outlined in these notes. This will not be easy! C, through its basic features, actively seduces us into dangerous ways. We should never say to people :-

> ''OK! Now you can forget all about Pascal. No-one uses it in the real world. We use C for most of our systems engineering and you must learn to speak C as a real C person.''

Instead, we should say :-

> ''OK! Now you must remember very carefully all the lessons about software engineering we have taught you in Pascal. You are going to use C now. It's a lower level language, but it gives you access to certain things in our system that our Pascal does not know about. On the other hand, it is very dangerous. Write in C, but continue to think in Pascal. Follow the guidelines, maintain discipline yourself — nobody else will — and you might just get things to work.''

Our students are going to be asked to analyse, design, implement, maintain and work with systems (many of them life-critical) whose complexity greatly exceeds those of the present day. At the programming language level, these challenges cannot be answered without simple, powerful and precisely understood semantics that are amenable to formal reasoning and tools. From the perspective of Pascal, occam lies in the right direction. Going to C moves us in entirely the wrong one.

REFERENCES

[0] C.A.R.Hoare et al. : ''Laws of Programming''; Commun. ACM 30, 8, pp.672-686; August 1987.

[1] A.W.Roscoe and C.A.R.Hoare: ''Laws of Occam Programming''; Technical Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, Oxford, OX1 3QD, ENGLAND; 1986.

[2] D.C.Barnes: ''Going to C''; Internal Documentation, Computing Laboratory, The University, Canterbury, Kent - CT2 7NF; 1988.

[3] J.D.Bovey: ''An Introduction to C Programming''; Internal Documentation, Computing Laboratory, The University, Canterbury, Kent - CT2 7NF; 1988.

[4] Dijkstra, E.W., ''GOTO One More Time'', Commun. ACM 30, 8 (Aug. 1987), 661-662.

[5] C.A.R.Hoare: ''The Emperor's Old Clothes''; 1980 ACM Turing Award Lecture, Commun. ACM 24, 2, pp.75-83; February 1981.

[6] R.Shepherd: ''Compiler Support for Floating-point Computation''; Letter to the Editor, Software — Practice and Experience, 18, p1193; December 1988.

[7] R.Shepherd: ''Security Aspects of Occam''; Technical Note 28 (72-TCH-028-00); INMOS Ltd., Bristol; 1987.

DATED

January, 1989, University of Kent at Canterbury