Department of Computer Science
University of Manchester

Manchester M13 9PL, England

Technical Report Series

UMCS–92–12–3

Ian D. Cottam

idC: A Subset of Standard C for Initial Teaching

idC: A Subset of Standard C for Initial Teaching*

Ian D. Cottam

Department of Computer Science
University of Manchester
Oxford Rd., Manchester, U.K.
`ian@cs.man.ac.uk`

Abstract

A subset of the programming language ISO Standard C is defined informally. It is claimed that the resulting language (called *idC*) is suitable for introductory programming classes for university-level students. The claim comes with a caveat that the language requires a certain level of software support not normally mandated (e.g. by the C standard). After justifying each restriction, a sketch of the first prototype implementation is presented. *idC* may well be a suitable 'workhorse' language to introduce concurrently with, or shortly after, a functional programming language such as Haskell or SML.

Technical Reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `m1.cs.man.ac.uk` (130.88.13.4) in the directory `/pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, U.K.

1 Introduction

Over the last few years many university departments of computer science have become dissatisfied with their introductory programming courses using the Pascal language. There is a feeling that the language is past its prime, possibly because it does not support any of the latest ideas of modularity, information hiding and object-oriented programming. Several departments now teach non-standard languages such as Modula-N (where N is usually 2 but occasionally 3) or Tu*b*-Pascal. Some have even made the brave leap to C++. If one believes strongly in the importance of programming languages backed by international standards bodies, independent of manufacturer, and well-supported by standard conforming compilers, two languages appear to be the main contenders for Pascal's teaching crown: Ada and C.

After trying Ada for myself, it was rejected on the grounds that it is too complex and, in some areas (e.g. exceptions) difficult to subset sensibly. In a small department, where all the staff used and believed in Ada's worth, the problems can be overcome. In large, diverse departments, such as Manchester, with no "Ada culture" to speak of, it cannot. A frustratingly expensive problem, the world over, is that of interfacing students' programs to database systems, window managers, etc. Ada must be better than Pascal for this, but, at least in my environment, there is no belief in the worth of changing to Ada. Let us turn, therefore, to the other contender: "C."

C [1] is the workhorse language of choice for commercial software engineers, and– let us admit it publicly–for many academics too. Why are many academics concerned about the wide-spread use of C? In a nutshell, it is because twenty years of perceived wisdom about structured programming was ignored by the designer(s) of the language. Their seminal, and otherwise beautifully written, textbook [2] preaches the joys of programming via side-effecting expressions and one-entry/multi-exit statements. All subsequent books on C, and there are lots of them, follow the masters.

People (especially men) delight in mastering the C language. It is a badge that says all that dogma about structured programming was for children and beginners. "I have left behind my bicycle-with-stabalisers and now ride a 1000cc BMW; I live dangerously and look I'm not dead yet." When others reply "but, it is possible to have C's pragmatic advantages of portability and flexibility without abandoning the laws of structured programming" they are shouted down by cries of "if you want Pascal, you know where to find it" and "we need the speed that $while(*d++ = *s++)$; gives us" and "if you can't cope with full C, you'll never be accepted as a competent, professional, software engineer."

Real academics should not be forgotten either. Real academics would not dirty their hands with C. They are content to let their juniors solve all the trivial, pragmatic problems of setting-up and running programming laboratories, writing library bindings year after year, and coping with all the dialects of their beautiful, academic language.

On Structured Programming

There is less debate about structured programming than in years gone by. Perhaps partly because it is assumed to be so widely understood that the word 'structured' is superfluous, partly because the phrase 'object oriented' has replaced it as the phrase on the lips of the world's programmers. It is also used to mean something like "any well-structured software system at the level of decomposition into modules." Herein, structured programming is taken to be the simple idea(s) summarised below:

We would like the complexity of our program modules to be related linearly to the length of the program text. We would like to reason about the value of expressions using 'school algebra'; for example, simplifying by substituting equals for equals. A trivially simple way to achieve these two precious aims, in the traditional, imperative style, is that all the statements in a program must have a single "point of entry" and a single "point of exit", and expressions should be "free of side effects" (i.e. no program variable may have its value changed by an expression).

The beauty of these simple rules is that they enable programmers to reason about the behaviour of their programs logically, rather than operationally. Further, there is no giant leap when one wants to argue formally about the correctness of (part of) a program: the rules of the game ("Hoare logic") simply encode the ideas of structured programming espoused above.

It is possible to relax the one-entry/one-exit rule for statements and still reason logically about program behaviour. However, the cautionary words of E. C. R. Hehner in his book [3] should be heeded:

> "Some programming languages include a command to escape from the middle of an iterative construct. It may be called exit, leave, break, escape, or return. Without it, one often needs to introduce a logical variable, set it, and test it each time round the loop in order to get out of the loop at the right time; this complicates the program and makes it less efficient. Unfortunately, including the escape construct is even worse. Although its operational explanation is simple, its semantic definition is unuseable in program composition, and proofs become difficult."

Hidden away inside Standard C is a subset that obeys the structured programming laws. The proposal is that its boundaries should be respected. Rather immodestly it is called *idC* for "Ian's Disciplined C." The author has used *idC* for a long time, purely as a discipline, but for beginners and people tempted to stray a conforming processor has been implemented. The next section discusses the language restrictions, and the section after that is a report on the current implementation at Manchester.

2 *idC*: The Language

For good engineering reasons, like portability, it should be stressed that it is crucially important that *idC* is a proper subset of Standard C. This, hopefully, explains some of the design decisions that follow. This major section is divided into three: Expressions, Statements, and Macro Processing.

2.1 Expressions

All flavours of assignment (that is `x= exp; x += exp; --x`; etc.) are treated as if they were statements and not expressions. An assignment operator may only appear as the primary operator of an expression in statement context. Note that init and step in

```
for (init; test; step)
        body;
```

are regarded as being statement context with respect to the assignment operators.

This restriction also has the effect of catching one of the more common mistakes in neophyte C programmers' code: using the assignment operator when equality was intended.

```
t1.c:
1:      void procedure(void)
2:      {       int x= 0;
3:
4:
5:              while ( x = 0 )         /* should be == */
6:                      /* skip */;
7:      }
t1.c:5: assignment within expression considered harmful!
```

C has the useful notion of sequential execution of expressions, using the comma operator. An exception to the rule is made here in that assignments may occur in the left-hand operand of "," even in pure expression contexts. This allows idioms such as:

```
while (ch= getc(stdin), ch != EOF)
        process(ch);
```

with equivalent meaning (at least in *idC*) to:

```
for (ch= getc(stdin); ch != EOF; ch= getc(stdin))
        process(ch);
```

Getc(), getchar(), putc() and putchar() are usually macros that expand to inline code for reasons of efficiency. They also, typically, include assignment operators. The processor for *idC* carefully declares them as proper functions in order to avoid several error messages about assignments within expressions.[1] However, it must be admitted that this is a possible problem area as providers of interfaces to applications—such as window managers—may well have felt entirely free to provide 'getc-like' side-effecting macros.

2.2 Statements

goto

Any use of goto results in an error message. Labels are allowed, but their only use is for annotation. In Standard C, non-local jumps are only possible via the pair of library functions setjmp() and longjmp(). Currently, *idC* does not object to their use on the pragmatic grounds that any student that can understand their semantics is welcome to try and use them! (It would be better, presumably, to at least warn about their use.)

continue

Any use of continue results in an error message.

break

Use of break (to terminate a loop) results in an error message. Standard C uses the same break statement to terminate switch case arms. An *idC* processor must be careful to allow the latter whilst banning the former. In fact, switch breaks are mandatory in *idC* (see below). As well as supporting the one-entry/one-exit rule, this restriction removes another frustrating source of error from student programs; that is, the programmer's mistaken belief that a break inside a loop body will always terminate the loop—even when the break is inside a switch!

return

A return statement, if present in a function block, must be textually the last statement before the block's closing '}'. Standard C uses the library function exit() to leave the entire program. Programmers often use it, in place of return, in the main() function. It is also common to use it in user-written 'abort' functions. Currently, *idC* permits exit(), but a warning might be sensible. The intent is to supply students with a single function—AbortIf()—that calls exit() on error.

[1]Thanks to Richard O'Keefe for this idea.

Here it is:

```
# include "abortif.h"
# include <stdio.h>
# include <stdlib.h>
# include <stdarg.h>

void
AbortIf(int failure, char *fmt, ...)
{
        va_list args;
        extern char *prog__name;

        if (failure) {
                va_start(args, fmt);
                fprintf(stderr, "%s: ", prog__name);
                vfprintf(stderr, fmt, args);
                fprintf(stderr, "\n");
                va_end(args);
                exit(EXIT_FAILURE);
        }
}
```

No other use of exit() is necessary.

switch

Every case alternative of a switch statement must be terminated by a break. Even the textually last case (often the default) must have a break. In C when a case is common to several alternatives it is written, for example:

```
        switch (expr) {
         case 'a':
         case 'e':
         case 'i':
         case 'o':
         case 'u':
                /* process vowel */
                break;
         default:
                /* ... */
                break;
        }
```

and is, of course, still permitted in *idC*.

2.3 Macro Processing

idC source modules may not contain any macro preprocessor line (i.e. where the first significant character is a '#') except #include. In other words, students may not #define macros or indulge in conditional compilation.

Macros

Macro semantics are difficult and very different to functions—although students assume they are equivalent because they look equivalent. The fact that normal block-structuring scope rules do not apply to macros is particularly upsetting. The restriction prevents another devastating error common to neophyte C users:

```
# define N 42
# define N_SCALED N+100

float vector[N_SCALED*10];
```

The array vector has just been defined to have 1042 elements, and not 1420. Why? Because, after macro expansion, the compiler sees:

```
float vector[42+100*10];
```

In *idC* one would give the array definition as:

```
enum {
        N        = 42,
        N_SCALED = N+100
};

float vector[N_SCALED*10];
```

It is a little sad that const in Standard C means 'read-only initialised variable' and hence, the more intuitive formulation,

```
const int N        = 42;
const int N_SCALED = N+100;
float vector[N_SCALED*10];
```

fails because $N\_SCALED * 10$ is not a constant expression. Const can, however, be used in other contexts.

Not having #define also stops beginners from playing syntactic tricks, such as making C look like another programming language:

```
# define begin {
# define end   }
```

etc. Recently a cry for help was posted to Usenet as to why the following C code resulted in several error messages:

```
# define NOT !

if (a NOT= 2)
        return 1;
```

(The answer is left as a simple exercise for the reader.)

Conditional Compilation

Avoiding the #if family saves the student from yet another unnecessary and confusing complication. The use of “#if 0” or “#if DEBUG” for debugging can also be replaced by source-level debugging tools.

3 Implementation

Most of the *idC* restrictions are implemented via additions to the *lcc*[4] portable Standard C compiler from Princeton University and AT&T Bell Laboratories. No *lcc* code was deleted or modified; the extra checks are all pure extensions and may be conditionally included. This technique has several advantages.

- Hopefully, *lcc*'s standard conforming code is unaffected.
- When new releases of *lcc* are received, it should be simple to re-do the extensions.
- *lcc* is small, fast and highly portable; *idC* does not make one dependent on SUN Microsystems (the current platform for teaching at Manchester).

Although it might seem easier to have implemented some checks by simple deletions in the scanner or parser, two problems would have had to be solved. The first is that precise error messages would have been harder to guarantee, and the second is that goto, continue, etc., must not be allowed as identifiers in *idC* programs.

At Manchester we don't have a code generator for *lcc*. However, this all helps to demonstrate the portability of *idC*. First the modified *lcc* is run and then, if there are no errors, the normal Standard C compiler (such as *gcc*) is used as a code generator. *lcc*'s front-end is so fast that the extra time to compile, over say *gcc*, is negligible. What is a little slow in the current implementation is that the controlling program is a shell script that is also used to detect macro use. This should be trivial to speed up when the need arises.

3.1 Current Limitations and Restrictions

- A student could put macro definitions in a header file and #include them without detection.
- The syntax of the C switch statement is so general that it is a little tricky to match case/break points exactly. Until this code is better understood (proven?), only a warning rather that a hard error is produced.
- Very occasionally the side-effect checking code produces the same error message twice.

3.2 Other Supporting Tools

C, even *idC*, requires excellent auxiliary support tools, especially in a student environment. Fortunately, they are more or less generally available. Below is a list of the kinds of tools together with our current choice.

- A source-level debugger [*ups*].
- A post-mortem dump analyser [*pmd*: a shell script written by the author that calls either *dbx* or *sdb* to translate a core dump into a source-related call trace plus local variables].
- A run-time system that checks for unitialised variables and illegal memory accesses [*purify*].
- A "pretty printer" [*indent*].
- A call-graph profiler [*gprof*].
- A cross-module interface checker [*lint*].

SUN have an unbundled *lint* and there exist versions from independent suppliers. *purify* has a couple of problems too. It is SPARC dependent, and the current release does not always trap a common student error: that of being off by one when indexing a local array. The suppliers of *purify* aim to add this capability in a future release.

4 Conclusion

More and more universities are using (full) C as a workhorse programming language in introductory courses. Many of them see it as an exposure to 'reality' following on, or concurrently with, a first programming course using a clean, functional language such as Haskell or SML. Practically all universities allow final year students to teach themselves C for project work (with wildly varying results). *idC* is an attempt to retain all the virtues of C without losing the virtues of structured programming. But let us not forget the problems we face:

- *idC*, as a teaching language, is not proven;

- The ease of interfacing to applications, written in full C, is not proven;
- Many students (and tutors?) will rebel against using a subset language;
- Practically all textbooks use full C in the 'unstructured' style;
- Information hiding and modular design in the abstract data type style still require a highly disciplined use of the language;
- Students will have to master pointers a little earlier than is the case with, say, Pascal (i.e. call-by-reference simulation);
- Arrays remain second class objects;
- Numerical analysis (in the sense of matrix algebra) is harder than in, say, Pascal (with conformant arrays, ISO optional level 1);
- The 'obvious' vehicle for teaching object-oriented concepts in the second or subsequent years would be C++.

Despite the above list, there is a real trend in academia to abandon the 'Pascal compromise' and teach the beauty of programming via a simple functional language and the ugly reality via C. The author supports this trend, but hopes we can survive with a little less mess on our hands by grasping a new compromise—*idC*.

5 Acknowledgements

The author is grateful to many of his colleagues at Manchester for numerous suggestions; particular thanks go to Mario Wolczko. Outside of Manchester, I would like to thank Richard O'Keefe, who agreed to review a draft of this paper, and Christopher Fraser and David Hanson for making *lcc* available.

6 References

[1] ANSI. *Programming Language C, X3.159-1989.* American National Standards Institute, 1989; also adopted as *ISO/IEC 9899:1990.*

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall International, 2nd edition, 1988.

[3] E. C. R. Hehner. *The Logic of Programming*. Prentice Hall International, 1984.

[4] Christopher W. Fraser and David R. Hanson. *A Retargetable Compiler for ANSI C*. Princeton University, Department of Computer Science Research Report CS-TR-303-91, 1991.