22. Cb: A Low-level Subset of C

W.M. McKeeman
Digital Equipment Corporation
110 Spitbrook Road
Nashua, NH 03062

Abstract

A subset of Standard C is presented together with a set of transformations for taking Standard C into the subset. The intended purpose of the subset is to enable a software vendor to distribute a C program in source form without disclosing proprietary ideas that would normally be obvious to someone reading the source. Subsidiary uses are an enhancement to ANDF, the definition of some C semantics, a low-level target for language processors, and a C compiler test generator.

Introduction

The distribution of software in source form is attractive from the viewpoint of portability (standard library headers are target dependent) and user-chosen application tailoring (implemented with #ifdef). On the other hand availability of source code normally reveals more proprietary information about the software than the vendor would like. One solution is to mangle the source for human readers without changing the meaning of the program. This paper presents some largely untired ideas for creative mangling.

The idea of mangling source has been around a long time. The Stanford Computer Center compressed Algol source to save space and card stock in the metal file cabinet which held the run-time library. Randy Meyers tells me that Digital distributed some of the PDP-11 operating system without comments in MACRO assembler source. Jim Gimpel mentioned some distribution of obscured Fortran. The C Shroud\textsuperscript{4} product is an existing example of a program to obscure C source [3, 4].

This paper introduces a very small but complete subset of standard C [1]. The subset, a kind of opposite to C++, is an elaboration of the C Shroud technology. The monogram for C++ could have been C^\# because it is a little bit higher than C. I have therefore chosen the name C\# for C with its structure flattened. It is my intent that the name C\# be generic in the sense that it can be

\textsuperscript{4}C Shroud is a trademark of Gimpel Software.
used for any language designed in the spirit of C#. C# can be compiled by any standard-compliant compiler or alternatively by a (simpler) C# compiler. The use of C# is as a target intermediate language. It can also be used to generate a new kind of test for compilers.

The intent of flattening a C program down to C# is to obscure the engineering decisions that went into the program without degrading the eventual executable. The C# form of the source can then be safely used as a distribution medium. C# is principally a post-preprocessor source and target language. A special hack is added to allow the inclusion of standard library headers to be deferred, allowing the #include to be passed into C#. All other preprocessing is completed prior to flattening. C# therefore does not provide the capability for the user to make build-time choices by setting -D flags for the compiler.

ANDF, standing for ‘Architecture Neutral Distribution Format’ is a compiler intermediate language with some special features to parameterize it relative to different target machines [2]. One of the requirements for ANDF was security of proprietary engineering information. What was achieved is the protection naturally inherent in tree-structured compiler intermediate code (which is to say, not much). To remedy this defect in ANDF, C# can be used as a preprocessor for ANDF input to enhance the protection and make reverse engineering more difficult. One should still not expect too much. While removal of comments and renaming of variables is not mechanically reversible, much of the structure of the program could be largely recovered from flow-graph analysis, which is an inherent part of optimizing compilation.

A set of transformations from Standard C into C# is the principal contribution of this paper. A well-engineered C program can be processed into C# leaving an ugly but still standard compliant C program. Because the transformations introduce a large number of explicit gotos, the eventual compiler of C# must have excellent flow analysis and optimization. There is the chance that a compiler would be able to compile some program but not its flattened form. That possibility reduces the utility of the idea proposed here and also motivates fixing any errant compilers.

The unit of translation is the module. In its flattest form, the translated module contains no comments and no pretty white space. One static variable without initialization replaces all the uninitialized statics. Each function contains the declaration of one local variable followed by a sequence of so-called flat fragments. The flat fragments are position-independent, and therefore can be sorted into an arbitrary order. The enabling technology of the transformations is a traditional front-end for C including scanner, preprocessor, parser, and symbol table. The flattener itself would be about as complicated as a straightforward C compiler. I have not implemented it. There probably exist correct C programs that cannot be flattened, requiring some adjustments to user source.

The order of presentation below of obscuring transformations is not the algorithmic order of application.
An Example

A hand-prepared before/after, C/C\# example follows. As a courtesy to the reader, white space has not been entirely suppressed. The after form shows the result of a few of the obscurations. They are pretty horrible for the human reader. C compilers, on the other hand, face this kind of stuff all the time. The introduced typedef names below may force the programmer to change some program names to avoid complaints from the mangler.

/* before scrambling */
#include <stdio.h>
int main(int argc, char **argv) {
    int i, j;
    struct {
        char a;
        wchar_t b;
    } c;
    i = c.b = argc--;
    {long j = 0;
      while(--i) j++;
    }
    exit(0);
}

/* after scrambling */
typedef char T1;
typedef signed char T2;
typedef unsigned char T3;
typedef short T4;
typedef unsigned short T5;
typedef int T6;
typedef unsigned int T7;
typedef long T8;
typedef unsigned long T9;
typedef float T10;
typedef double T11;
typedef long double T12;
#include <stdio.h>
int main(T6 LENF021812, T1 **LEDE0111B8) {
    T3 L[
    3*sizeof(T6)
    +sizeof(wchar_t)
    +sizeof(T8)
}

The Header Hack

At the cost of losing the portability parameterization of standard library headers, those headers can be expanded and their contents obscured along with the rest of the program. This is the simplest way to handle headers. If preserving the #include directives into the C\# target is deemed important, one can imagine a two stage elaboration to the flattening process.

The insertions from the standard library headers are in fact macro definitions and declarations of various kinds. These header contents are inserted as usual into the C source file during the C\# translation process, using a vanilla version of the header. Any name defined in the inserted material is marked *un touch able* (not be be mangled and not to be expanded). In the final target image, the inserted material is deleted and the original #include directive restored in its place. This hack will not work for headers in general because not all headers are structured as simply as those in the standard library.

Boring Names and Expression

One of the tasks of the flattener is generating names that convey no information to replace the names carefully crafted by the programmer. I propose names of the form LXXXXXXX where the Xs are random hexadecimal digits. They are reasonably simple to generate using rand() and sprintf() with format \%AX. In C\# generated names are used temporarily for variables but in the final output
appear mostly as labels. The fixed-width format is used because it is convenient in the final scrambling sort of executable statements.

**Variable Renaming**

Because of scoping and name spaces, one name can be used for more than one thing in a C module. The symbol table is the mechanism that separates the uses based on context. Suppose that every use of every name local to the module (e.g. no externs) and not defined in a library header is consistently given a unique LXXXXXXX name. The resulting C program still has the same behavior as before the substitution.

Automatic variable declarations in function scope or deeper may or may not have initializers. If they do, the initializers can be dropped and replaced with one or more explicit assignments just following the declarations. Now any declaration in inner function scopes can be physically moved to the outermost function scope without introducing any name conflicts or changing the meaning of the program, or (excepting performance) changing its behavior. Other declarations in local contexts are treated similarly. The declaration transformations allows the simpler rule:

```
compound-statement:
  { statement-listopt }
```

dropping all local declarations, to be used except for function-definition itself.

**Local Variable Clotting**

The next set of transformations eliminates most declaration and expression information. It is described now because it fits with the previous material but in fact must be carried out later because declarations and expressions subject to these transformations are generated in some later processing steps.

Each function scope contains zero or more variable declarations. Delete all the variable declarations (except for formal parameters to the function, structures with bit-fields, and anything carrying the qualifier volatile). Replace the declarations with a single declaration:

```
T3 L[n];
```

where \( n \) is the total accumulated size of the locals as expressed as a sum of \( \text{sizeof}(\text{type}) \) primaries. Now replace every reference to a local variable with an expression of the form

```
*(\text{type } *)(L + \text{frame-offset } + \text{local-offset})
```

where \( L \) becomes a base address for the local frame, the first offset is a (perhaps null) sum of \( \text{sizeof} \) operands giving the start of the variable in that frame, and the second offset is another sum of \( \text{sizeof} \) operands or an \( \text{offsetof} \) operand
giving the offset within the variable (for a structure member). The cast restores the type information lost when the declaration was deleted. Avoiding the generation of explicit constant values keeps the transformed code target independent. It is reasonable to assume that L, being the first variable in a frame, is aligned on an efficient memory boundary, as are all subsequent local variables. The offsets take this into account by rounding up to the size of the addressed data item as necessary.

Variables with volatile can be included in clotting if the whole array L is given the attribute volatile. This may, however, cause optimizers real trouble.

Structure selection '.' has disappeared except for bit-fields. All subscripts a[e] are turned into their equivalent form *(a+(e)). All structure pointer indirections e->n are turned into their equivalent *(e) + local-offset).

Expression Flattening

Turn all compound assignments into a pair of simple assignments
(i.e. ++ -- *= /= %= += -= <<= >>= &= ^= |=):

- before-
  ++a  (a+=1)
  --a  (a-=1)
  a++  (p=&a, t=*p, *p+=1, t)
  a--  (p=&a, t=*p, *p-=1, t)
  a op= a  (p=&a, *p=*p op e)

and transform expression statements not in the form:

    ;
    p;
    a = p;
    a = q op r;

Here a is an identifier or dereferenced identifier; p is an identifier, constant or function invocation; and q and r are identifiers. Transform these statements into a sequence of expression statements in the above forms by introducing temporary variables. All parameters for function invocations are therefore generated names, to which the actual values have just previously been assigned.

The consequence of these transformations is to replace all local names with anonymous references to the local frame (except for formal parameters which just get boring names), and to replace all expression statements with the equivalent of compiler triples. The transformations never cause side-effect changes because each side-effect causing construct is moved out of its containing expression into an assignment to a temporary variable.

Because expression flattening introduces new variables and variable clotting introduces new expressions, these two processes are best carried out together. The example starting this paper, in fact, left generated local variables uncotted, which is a viable option.
Static Variable Clotting

Uninitialized static variables are collected analogously to variables in a frame. The clot is

```c
static T3 S[n];
```

Initialized static variables are renamed LXXXXXXX but otherwise left alone. After these transformations only the external names and names from library headers have their original mnemonnic form.

Transforming the Function Body

A series of transformations is applied to the executable statements in the function body, outside in. In each case the outer construct is matched to a template and replaced with a simpler, and perhaps more verbose, equivalent. The final objective is to turn the entire function body into a non-nested list of fragments, each having one of the following six forms where e is an expression:

- `Lxxxxxxx: goto Lxxxxxxx;
- Lxxxxxxx: return;
- Lxxxxxxx: return e;
- Lxxxxxxx: e; goto Lxxxxxxx;
- Lxxxxxxx: if (e) goto Lxxxxxxx; else goto Lxxxxxxx;
- Lxxxxxxx: switch (e) { transfer vector } goto Lxxxxxxx;

The `transfer vector` is a list labeled statements of the form `case e: goto Lxxxxxxx;` followed by one more `goto` See the transformations for `switch` below for more detail.

These constructs are called flat fragments. Part of the process is generating label names. Since there is no value in obscuring the meaning of this paper from the reader, the more readable names `L00, L01, L02 ...` will be used in the examples instead of random values for LXXXXXXX. The flat fragments are position-independent within the function body, and therefore may be sorted into a meaningless order before exposing them to examination. Since the labels are randomly generated, the labels become a convenient key for the sort.

The first transformation is applied to the `statement-list` in the function body. The `statement-list` is transformed as follows:

-Before-

```c
{local declarations; 
S0;
S1;
...
S9; 
}
```

-After-

```c
{local declarations; goto L00;
L00:S0; goto L01;
L01:S1; goto L02;
...
L03:S9; goto L04;
L04:return;
}
```
This transformation may generate unreachable code; never mind—it will be taken care of later. The generated return may be inconsistent with the function itself. If the original program conforms to the standard, an inconsistent return cannot be reached, so never mind. The resulting fragments are position-independent but not necessarily flat because each of the statements \( S_1, S_2, \ldots, S_3 \) above may be complex. Thus there are some transformations left to do. Each statement can be processed independently of its neighbors, which means one at a time, or even in parallel.

### Transforming Statements

The total set of statements that must be transformed is described in the C Standard [1]. Some of the transformations are purely textual; some require context analysis (such as associating break with its correct span).

#### Transforming break Statements

The break statement requires special preprocessing before the other transformations are carried out. Suppose \( S_5 \) is an iteration-statement or switch

\[
\text{L05: } S_5; \text{ goto L06;}
\]

and \( S_5 \) contains one or more occurrences of break not nested within a deeper iteration or switch. Then the text goto \( \text{L06} \) is substituted for each such break.

#### Transforming Empty Statements

An empty expression statement is shortened:

- \( \text{L07: } \); goto L08; \rightarrow \text{L07: goto L08;}

#### Transforming goto Statements

A goto statement is shortened:

- \( \text{L09: } \text{goto L0A; goto L0B; } \rightarrow \text{L09: goto L0A; } \)

#### Transforming return Statements

A simple return without a return value drops the following goto to get into flat form.

- \( \text{L0C: return; goto L0D; } \rightarrow \text{L0C: return; } \)
If there is an expression, it may be separated out into an assignment that can be flattened using the expression algorithm detailed above.

\[-\text{before-}\]
L05: return c; goto L0F;
L11: return L10;
\[-\text{after-}\]
L05: L10 = c; goto L11;

Transforming Labeled Statements

A labeled statement is flattened as follows:

\[-\text{before-}\]
L12: L13: S6; goto L14;
\[-\text{after-}\]
L12: goto L13;
L13: S6; goto L14;

Transforming Compound Statements

A compound statement has no declarations because of the initial flattening of the declarations. The transformation removes the curly braces.

\[-\text{before-}\]
L15: \{ S7; S8; \ldots; S9; \} goto L16;
\[-\text{after-}\]
L15: S7; goto L17;
L17: S8; goto L18;
\ldots
L19: S9; goto L16;

Transforming switch Statements

Replace the switch expression with a generated variable and precede the switch with a flat assignment of the expression to the generated variable. Then transform the switch by inserting a new switch statement body just before the original switch statement body. The new body is a transfer vector in the form of a compound-statement listing all the original case labels (including default), each followed by a goto to a new generated label, and one final unlabeled goto to the label following the original switch body. The original switch body itself gets a new generated label and each case label in it is replaced by the corresponding generated label in the transfer vector. This restores the program to flat form, although the generated assignment and substituted original switch body now need to be flattened. As before, this may generate some unreachable code. Never mind.
-before-
L20:switch(e) S10; goto L21;
L20:L22 = e; goto L23;
L23:switch(L22) {
    case C1: goto L24;
    case C2: goto L26;
    ...
} goto L21;
L27:substituted S10; goto L21;

-after-

Alternatively, a switch statement can be mangled into a sequence of if statements, but this places an unreasonable load on the optimizer of the eventual compiler.

Transforming Iteration Statements

An iteration-statement has one of four forms: two for for and one each for do and while. For all of them the continue statement must be processed as part of the flattening transformation. The details of removing continue are slightly different for the four cases.

The while statement is treated much as in C Shroud and also the standard.

-before-
L28:while(e) S11; goto L29;
-after-
L28:if(e) goto L2A; else goto L29;
L2A:S11; goto L28;

Any continue statements in S11 (and not in a more deeply nested iteration-statement) are replaced with goto L28 either before or after the above transformation.

The do statement is similar:

-before-
L2B:do S12; while(e); goto L2C;
-after-
L2B:S12; goto L2D;
L2D:if(e) goto L2B; else goto L2C;

The continue statements in S12 turn into goto L2D.

The for statement with a non-null limit expression e2 translates as follows. Either e1 or e3 or both can be null.

-before-
L2E:for(e1; e2; e3) S13; goto L2F;
-after-
L2E:e1; goto L30;
L2E:e2; goto L31; else goto L2F;
L31:S13; goto L32;
L32:e3; goto L30;

The continue statements in S13 (not in more deeply nested loops) turn into goto L32.

The for statement with a null limit expression translates as follows. As above, either e1 or e3 can be null.
-before-
L33: for(e1; e3) S14; goto L34;  
L35: e1; goto L35;
L36: S14; goto L36;
L36: e3; goto L35;

-the statements in S14 (and not in nested loops) turn into goto L36.

Transforming if and else

There are two forms of the if statement, with and without else. The control
expression may be separated out to make them available to the expression
statement flattener.

-before-
L37: if(e) S15;
else S16; goto L38;

-after-
L37: L39 = e; goto L3A;
L3A: if(L39) goto L3B; else goto L3C;
L3B: S15; goto L3B;
L3C: S16; goto L3B;

-before-
L3D: if(e) S17; goto L3E;

-after-
L3D: L3F = e; goto L40;
L40: if(L3F) goto L41; else goto L3E;
L41: S17; goto L3E;

Dead Code

Programs in C may have unavoidable dead code. For example, the statement
following a switch clause cannot be reached because the switch is an uncondi-
tional transfer. After the flattening transformations, this dead code, and
other dead code introduced by the transformations, shows up as labels that
are never referenced. The flat fragments starting with unreachable labels may
be deleted without changing the effect of the program (and perhaps avoiding
compiler-generated or link-generated complaints). Mechanical means to recover
the original text may be defeated by such removal—something must be invented
to replace the removed stuff.

Similarly, arbitrary dead code can be inserted behind unreachable labels to
further obscure meaning.

Order of Flattening

Flattening takes place in a specific order because of the interdependence of the
steps.

1. Rename all but external names with LXXXXXXX names.
2. One function at a time:

(a) Replace all initializations with assignments.
(b) Move all declarations to the outermost function scope.
(c) Label the statements in the outermost statement list.
(d) Flatten one statement at a time, applying transformations until the statement is a sequence of flat fragments.
(e) Remove dead code.
(f) Clot local variables and flatten expressions.
(g) Sort the flat fragments by leading label.

3. Clot uninitialized static variables.

Testing Compilers

Testing requires generating a test, running the test, and evaluating the result against a criterion for correctness. The first and third steps are often labor intensive, so much so that providing them is a reasonable commercial activity. For compilers in particular, because there is no one correct output (object file), the test criterion is correct behavior of the compiled program. Any substantial testing of compilers, except for correctness and quality of error diagnosis, is necessarily indirect.

Suppose some arbitrary C program compiles and runs, producing some evidence of its behavior (trace, output, dump). The $\Phi$ transformation may be applied to the C program, the program compiled and run again. Now it is required that the two outputs are identical. Thus the problem of generating a test is reduced to selecting any existing C program. The problem of evaluating the test is reduced to a file comparison. When the flattener has optional transformations, more than one test can be derived from each C program.

The assumption underlying this test strategy is that flattening causes the compiler to take wildly different paths, and that comparing the results of those paths allows the compiler to be tested against itself. It is also true that flattened code would never be written by a programmer, so it tests otherwise unexercised functionality in the compiler.

In fact each use of $\Phi$ for making a distributable program falls into the above pattern. Surely the engineer is going to test the transformed program against the original. When a difference does appear it is not clear how one would go about locating the problem.

Summary

Nearly every Standard C program can be transformed into an equivalent program in $\Phi$. $\Phi$ is a reasonable low-level intermediate language, obscures engi-
neering design information, can be compiled by standard compliant compilers, and provides a new kind of compiler testing.

References


William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. He can be reached at mckeeman@tle.dec.com.