

# Extensions to the C Programming Language for Enhanced Fault Detection

DAVID W. FLATER AND YELENA YESHA

*Computer Science Department, University of Maryland Baltimore County, Baltimore, MD  
21228, U.S.A.*

AND

E. K. PARK

*Computer Science Department, United States Naval Academy, Annapolis, MD 21402,  
U.S.A.*

## SUMMARY

The acceptance of the C programming language by academia and industry is partially responsible for the ‘software crisis’. The simple, trusting semantics of C mask many common faults, such as range violations, which would be detected and reported at run-time by programs coded in a robust language such as Ada.\* This needlessly complicates the debugging of C programs. Although the assert macro lets programmers add run-time consistency checks to their programs, the number of instantiations of this macro needed to make a C program robust makes it highly unlikely that any programmer could correctly perform the task. We make some unobtrusive extensions to the C language which support the efficient detection of faults at run-time without reducing the readability of the source code. Examples of the extensions are automatic checking of error codes returned by library routines, constrained subtypes and detection of references to uninitialized and/or non-existent array elements.

KEY WORDS: C Reliability Range checking Error checking

## INTRODUCTION

It has been written that C provides about 50–80 per cent of the facilities one would want from a programming language.<sup>1</sup> One of the missing facilities is run-time error checking. Errors such as exceeded array bounds, out-of-range data values, and I/O exceptions, which are reliably trapped by the code generated by most Pascal and Ada compilers, go completely undetected by the code generated by most C compilers. Incorrect pointer usage and references to uninitialized variables can sometimes be detected with separate tools for static analysis (*lint*), but static analysis cannot detect all of the most common programmer errors.

C will continue to be used despite its limitations, not only because the choice of language is often forced by external constraints,<sup>2</sup> but also because many people have

---

\* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

become addicted to the ‘worse is better’ approach to design and coding.<sup>1</sup> Reliability and correctness are often sacrificed in order to shorten the production time of software. Our task is to enhance C in such a way that reliability and correctness will be easier to achieve without removing the language’s addictive qualities.

One might claim that C++<sup>3</sup> is just such an enhancement. Gabriel points out that C++ is closer to what we need than C and predicts its success.<sup>1</sup> However, although it is possible to implement internal consistency checks using C++ classes, it still requires more effort than most programmers are willing to exert for the sake of quality assurance. For this reason, the extensions which we make to C will apply productively to C++ as well.

## MOTIVATION

The looseness of the C language has resulted in the emergence of classes of coding errors which are truly insidious in their frequency of occurrence and difficulty of detection. One such class of errors is array bounds violations. Although many believe that out of bounds pointers are the primary problem in C programs, it is our experience that array bounds violations are more frequent. The reason why these happen so often, particularly in academic settings, could be as follows:

1. Most programmers who learn to program in a college or university use Pascal as their first programming language. Those who know how to program already are nevertheless trained in the Pascal programming style.
2. Although Pascal allows significant flexibility in the indexing of arrays, the convention is to start indexing from 1. Furthermore, Pascal traps array bounds violations at run-time.
3. The programmer learns a habitual way of coding loops to scan array indices from 1 to  $N$ . The programmer also learns that he or she does not have to be extremely careful when coding them since errors in these loops usually result in abnormal termination with a helpful error message showing the source of the problem.
4. When this programmer is forced to switch to the C language, he or she has a strong tendency to scan array indices beginning with 1 instead of 0.
5. Unless the compiler or linker happens to place the array at the very end of a memory segment, so that a reference to the non-existent ‘element  $N$ ’ generates a *segmentation violation* or a similar hardware exception, the coding error has an extremely good chance of never being discovered.

The effects of an array bounds violation make sense only at the level of machine code. The identities of the data which are overwritten cannot be determined from the source since they depend on the arrangement of data in memory when the program is executing. There can be a different arrangement every single time the program runs if data are dynamically allocated; otherwise, the arrangement can change every time the program is recompiled.

The chances of the fault being discovered when the array is not at the end of a memory segment are poor. There are several subcases:

- (a) *The data which are clobbered (unintentionally overwritten) are never referenced after the clobbering occurs.* In this case, the fault is completely masked.
- (b) *The clobbered data have low sensitivity.* The word *sensitivity* is sometimes

used to indicate the probability that a failure will result from a particular fault in the software.<sup>4</sup> This probability was estimated in the cited paper using an analysis of code segments. For the types of faults we are considering, the probability that a fault will be masked is sometimes more directly related to the overall usage of a particular datum than to any particular code segment. We thus feel that it is valid to refer to the *sensitivity* of a datum in an executing program. It is our experience that the sensitivity of the vast majority of data referenced by a program quickly decreases as the size of the program increases; the unfortunate consequence is that failures resulting from array bounds violations are usually intermittent and difficult to replicate.

- (c) *The clobbered data have high sensitivity.* This is the exceptional case in which the programmer is lucky enough to have clobbered some data which are vital to the continued execution of the program. Failures may be intermittent, but they are likely to be systematic. Even so, it frequently happens that the fault is never located. Consider the following common scenario. A program crash occurs. The programmer enters a source-level debugger and steps through the program until he or she notices an error. The programmer corrects a fault, repeats the test case, and the program terminates normally. Conclusion: problem solved. Now consider the following description of what has actually occurred. The fault is an array bounds violation. The failure occurred because the datum following the end of the array was clobbered and was important to the execution of the program. The programmer located and repaired an unrelated problem, with the result that the arrangement of data in memory was slightly different after recompilation. Now the original fault is still present, but it is masked because the datum being clobbered has lower sensitivity.

Clearly the only case in which an array bounds violation is likely to cause the program to terminate abnormally *at the point where the violation occurs* is the hardware exception case. Many compiler and hardware dependent conditions must be met for this mode of failure to occur. If the hardware platform on which the software is being developed does not support protected memory segmentation, we have an even bigger problem. Not only have we lost our best indicator of array bounds violations, we have also gained an entire new class of failures. In those cases where a hardware exception might have been raised on another hardware platform, we instead have the possibility of *code corruption*.

## BACKGROUND AND RELATED WORK

In order to write robust programs in a fragile language, it is necessary to add code to explicitly perform the run-time checks which are needed. These checks are a form of executable assertion. Since it is prohibitively time-consuming and error prone for programmers to manually add assertions to test for the most common problems, it is preferable to have a tool to add them automatically. Whereas inserting code to test assertions at run-time is an old technique which has been proven to be very helpful in the development of reliable software,<sup>5,6</sup> the only other attempt we know of to extend the C language itself to provide a stronger semantic base for such assertions is *App*.<sup>7</sup> *App* uses a replacement preprocessor much like ours to convert annotations, which are effectively language extensions appearing within C comments, into constructs in the base language. The primary differences between

our work and *App* lie in the language extensions themselves and the kinds of errors they are designed to trap. *App* is first and foremost designed to check entry and exit conditions on functions; Robust C is more of a departure from the base language, but it provides *global* checking for out of range data, which is much more difficult to implement in the base language without automated tools.

A different approach which has been taken to improving C's robustness is strengthening the semantics of the language without changing the syntax. *Bcc*<sup>2,8</sup> appears to be the first utility to take this approach. *Bcc* traps a wide variety of probable errors in C code, including out-of-bounds pointers, dangling pointers, indirection through constants, misuse of null pointers, pointer arithmetic overflow and illegal array subscripts.<sup>2,8</sup> Unfortunately, the resulting programs are said to be intolerably slow.<sup>2</sup> *Saber-C*<sup>2,9</sup> traps errors using a C interpreter instead of a translator or an enhanced compiler, but it suffers from even greater slowdown.<sup>2</sup> Finally, *rtcc*, built out of the Portable C Compiler<sup>2,10</sup> in the spirit of *bcc*, checks array subscripts and pointer bounds with a slowdown factor of approximately ten.<sup>2</sup>

By allowing extensions to the C syntax, we can trap classes of errors which are impractical or impossible to trap without syntactic extensions. For example, it is impractical to automatically trap error conditions flagged by library functions unless there is a mechanism for the programmer to specify when such errors are to be handled explicitly or ignored. There is also no practical substitute for allowing the programmer to specify the legal values for a constrained subtype (a data type whose set of possible values is constrained<sup>11</sup>) immediately within its declaration. Extending the syntax enables us to use more efficient error trapping mechanisms as well. In the general case, detecting references to elements of arrays which have not been initialized requires the maintenance of a flag for every array element to indicate its status. When the element type is not a constrained subtype, there is no value to which the compiler can initialize array elements to indicate that they have not otherwise been initialized. However, if we give the programmer a way to provide an indicator value for uninitialized array elements, we can avoid the expense of maintaining flags. This has the added benefit that the programmer will be able to explicitly invalidate array elements which are no longer being used without generating a range violation.

## ROBUST C

In order to gain acceptance, Robust C must not impose upon the programmer.<sup>2,8</sup> It must follow the example of C++, supporting new constructs without changing the base syntax to such an extent that it becomes difficult to use the extended language. It should also be as portable as possible. We therefore implement the extension as a preprocessor which generates code conforming to ANSI/ISO C.<sup>12,13</sup> Since we translate the extended language into standard C, static analysis will continue to be done by *lint*; the only difference will be that *lint* will be run automatically on every translated source file. In the following subsections we will discuss some of the problems which Robust C is meant to obviate and their solutions.

### Array bounds violations

An array bounds violation occurs when an array is indexed by a value which is less than zero or greater than or equal to the number of elements in the array. The

latter error occurs with much greater frequency than the former, so we simplify our discussion below by discussing only the latter case. However, it should be understood that Robust C prevents both errors.

In practice, most arrays are of the simplest type, explicitly declared in the source code. Their size may be static or may depend on run-time parameters, but their size and number of dimensions will not change once they are known. In many programming languages, these are the only types of arrays that are supported. For these arrays we can resort to the direct approach of generating two modules, a 'read' module and a 'write' module. We will treat these modules as if they were implemented with function calls, but there is no reason not to insert code directly if this is more efficient. The read module would assert that all the array indices (passed as parameters to the module) are within range, then simply return the value of the referenced element. The write module would assert that all the indices are within range prior to setting the referenced element to the value passed as another parameter. All array references are then simply replaced with the appropriate function calls. This technique is illustrated in Figures 1 and 2.

A more powerful, and more expensive, approach is needed to handle arbitrary arrays which may change size at any time. The more thorough approach is to generate one read and one write module for each data type which appears as the element type of an array. These modules will accept a varying number of arguments (via the method defined in `stdarg.h` for ANSI/ISO C) to accommodate all arrays of that type regardless of the number of indices. The arrays themselves will be replaced with structures of the following form:

```
struct TYPEarr
{
    TYPE *array;
    unsigned long *nelm;
}
```

References to the arrays will then be changed into function calls which pass a pointer to one of these structures, a data value (where applicable), and a list of index values. `array` is a pointer to the start of the array. `nelm` is a pointer to the start of a linked list of values representing the upper bounds on the first, second, etc. indices of the array. The precompiler will recognize those situations in which

```
int playfair [PLAYFAIR_SIZE*PLAYFAIR_SIZE/4][2] =
    {{0,1}, {1,1}, {1,2}, {2,2}, {3,0}, {3,4}, {3,5}, {4,5}, {5,0}};

void add_playfair_letter (int letternumber, char letter)
{
    /* .... code deleted .... */
    row = playfair [number][0];
    col = playfair [number][1];
    /* .... code deleted .... */
}
```

*Figure 1. Fixed-size arrays (before processing)*

```

int playfair [PLAYFAIR_SIZE*PLAYFAIR_SIZE/4][2] =
  {{0,1}, {1,1}, {1,2}, {2,2}, {3,0}, {3,4}, {3,5}, {4,5}, {5,0}};

void add_playfair_letter (int letternumber, char letter)
{
  /* .... code deleted .... */
  row = read_playfair (number, 0);
  col = read_playfair (number, 1);
  /* .... code deleted .... */
}

int read_playfair (int a0,int a1)
{
  assert (a1>=0);
  assert (a1<2);
  assert (a0>=0);
  assert (a0<PLAYFAIR_SIZE*PLAYFAIR_SIZE/4);
#ifdef bogus_playfair
  assert (bogus_playfair != playfair[a0][a1]);
#endif
  return playfair[a0][a1];
}

void write_playfair (int a0,int a1,int value)
{
  assert (a1>=0);
  assert (a1<2);
  assert (a0>=0);
  assert (a0<PLAYFAIR_SIZE*PLAYFAIR_SIZE/4);
  playfair[a0][a1]=value;
}

```

*Figure 2. Fixed-size arrays (after processing)*

an array's character can change (e.g. calls to the alloc family of functions or any other assignments to the array pointer) and generate functions to perform the desired operations through the above array structure, modifying the array pointer and length fields as necessary.

Please note that these error checking techniques are not restricted to programs which are contained in one source file. If an external array is referenced, the preprocessor generates declarations for the external read and write functions and uses them within the current source file. If a pointer argument is used as an array, it is decoded by the appropriate TYPEarr function. Conditional compilation is used to prevent multiple copies of the same function from being seen by the compiler. Since arrays in the more efficient fixed-size format must be converted to the TYPEarr format when they are passed to functions expecting a pointer to an array of no particular size, it is important to provide complete declarations whenever possible.

Bounds checking for a particular array can be disabled with the `nocheck` flag. The programmer might want to disable checking for those rare cases where it is desirable to reference the same array in more than one way. (A dynamically allocated array in `foo[10][10]` may sometimes also be treated as in `foo[100]`).

### Undefined values

Although references to variables which have never been set are a problem for all kinds of variables, they are usually detected by static analysis unless arrays are involved. We will detect this error in the array case by adding an additional assertion to the read module described above which ensures that the element being read has been set previously.

To handle the (expensive) general case, it is necessary to allocate one bit for each element of each array to indicate whether or not the element has been set, clear all these flags at the time of allocation, set the appropriate bit when the write module is invoked, and verify that the bit is set when the read module is invoked. However, a much less expensive method is available in those cases where at least one ‘bogus’ value exists for the elements of an array. A bogus value is any out-of-range data value. The existence and identity of a bogus value depends entirely on the application, but it is usually easy for the programmer to find one. For example, an array intended to perform some transformation on ASCII text could use a bogus value of 4, since this is the EOT character and cannot appear in the input. The user enables inexpensive checking for references to undefined array element values by providing this forbidden value in the array’s declaration:

```
char global_character_map [MAP_SIZE] BOGUS (char)4;
```

The preprocessor generates code to initialize all elements of the array to the bogus value before they are otherwise used and enables the additional assertion in the read module, as shown in [Figure 3](#). An assertion is not added to the write module to ensure that the element is not explicitly being set to the forbidden value since it is useful to allow the programmer to thus specify that an array element no longer contains valid data.

### Ignored library error conditions

Nearly all of the commonly used functions in the standard C libraries have a mechanism for flagging error conditions with special return codes. In Pascal, the analogous functions will generally terminate the program if an error condition arises; in C, it is up to the programmer to test for the error conditions. While this adds much flexibility, it is the case that many C programmers do not bother to check return codes unless an error is *expected* to occur, such as trying to read past the end of a file. Since this is the exception and not the rule, we will have the preprocessor add assertions to check for error conditions except where the programmer specifies that this should not be done. The programmer disables error checking for a library function call by preceding its name by a tilde:

```
if ((~ fgets (whole[numrecs++], linelen, stdin))!=NULL) numrecs--;
```

```

char global_character_map [MAP_SIZE] ;
#define bogus_global_character_map (char)4
char read_global_character_map (int a0)
{
    assert (a0>=0);
    assert (a0<MAP_SIZE);
#ifdef bogus_global_character_map
    assert (bogus_global_character_map != global_character_map[a0]);
#endif
    return global_character_map[a0];
}
void write_global_character_map (int a0,char value)
{
    assert (a0>=0);
    assert (a0<MAP_SIZE);
    global_character_map[a0]=value;
}
#ifdef bogus_global_character_map
void init_global_character_map ()
{
    int a0;
    for (a0=0;a0<MAP_SIZE;a0++)
        global_character_map[a0]=bogus_global_character_map;
}
#endif
void robust_init ()
{
    /* .... code deleted .... */
#ifdef bogus_global_character_map
    init_global_character_map ();
#endif
    /* .... code deleted .... */
}

```

*Figure 3. Uninitialized element checking*

If the programmer had simply said

```
fgets (whole[numrecs++], linelen, stdin);
```

the preprocessor would have changed it to

```
assert (fgets (whole[numrecs++], linelen, stdin) != NULL);
```

If desired, the programmer may add the `#RCVERBOSE` pragma to the source, with the result that the program will call `perror` when applicable instead of simply terminating with an assertion failure.

### Incorrect loops

Loops of the form `for (a=0;a>FOO;a++)` are extremely common in C code. However, when the loop termination condition or the method of updating the loop variable become more complex, errors are frequently made which lead to the incorrect number of iterations.

In many cases the programmer can generate expressions for the upper and/or lower bounds on the number of iterations of the loop which should occur. If the programmer provides these expressions in the parenthesized part of the loop specification, it is trivial for the preprocessor to add an iteration counter and assert that the loop meets the specified criteria. In the most general case, a programmer may wish to specify an arbitrary assertion to serve as a loop invariant. We will allow these to be provided in the loop specification as well to preserve the structure of the source.

The following clauses may be added to a loop specification:

```
maxit n    Stop program if more than n iterations are made.
minit n    Stop program if loop is exited before n iterations are made.
invariant o Assert o on each iteration of the loop.
```

`maxit` and `minit` can be combined to ensure that exactly  $n$  iterations are made, or that the number of iterations falls within some range:

```
for (u=0;
    hash (u) != 42;
    u++;
    maxit PLAYFAIR_SIZE;
    minit 1;
    invariant ((u<=0)&&(t<=0))) { /* . . . . code deleted . . . . */ }
```

The above code translates to

```
{
  unsigned long itctr4;
  for (itctr4=0,u=0;hash(u)!=42;u++)
  {
    assert (((u<=0)&&(t<=0)));
    assert (++itctr4>=PLAYFAIR_SIZE);
    /* . . . . code deleted . . . . */
  }
  assert (itctr4 <= 1);
}
```

A `maxit` value being less than the associated `minit` value is a compile time error.

### Out-of-range data

Range checking is least intrusively added to standard C via the mechanism of constrained subtypes used in Ada. Our modified syntax is as follows:

```
typedef int (exclude -9999 9999) bignumbers;
bignumbers foo, bar, x, y, z;
```

The clauses which are legal inside the parenthesis are

```
include all    Legalize all values (default).
exclude all   Make all values illegal.
include x [y] Values between x and y inclusive are legal.
exclude x [y] Values between x and y inclusive are illegal.
```

Multiple clauses are processed from left to right. For example, the most common class of constrained subtype declaration would be coded similarly to the following:

```
typedef float (exclude all; include 0.0 250.0) car_velocity;
```

Although this syntax is slightly less convenient than Ada's, it facilitates the creation of such subtle beasts as a subtype to prevent division by zero:

```
typedef double (exclude 0.0) denominator;
```

To implement these subtypes in standard C, it is necessary to replace every assignment operation on a subtype with a function call (or code instantiation) so that the appropriate range checks can be made within the function. The same extension can be made more easily in C++; the preprocessor only needs to generate a new class for each constrained subtype and overload the operations so that each class performs its own range checking.

### Structured programming and the pointer problem

*Lint* traps most errors which involve the passing of a pointer to an object of the wrong type. This static checking, combined with the error checks on arrays provided by Robust C, is sufficient to trap pointer-related errors in structured programs. Unfortunately, many programs are not well-structured. Consider the following well-structured code segment:

```
{
    int looper;
    for (looper=0;looper>max;looper++)
        buffer [looper] = 5;
}
```

In order to achieve a marginal gain in efficiency, many programmers would express the same thought as follows:

```
{
    int *looper;
    for (looper=buffer;looper>buffer+max;looper++)
        *looper = 5;
}
```

This substitution preserves the functionality of a correct program, but it does not preserve the semantic information expressed by the first code segment. In the first segment, we are scanning an array; in the second, we are repeatedly writing through a pointer and then modifying it. Whereas we can be sure beyond a reasonable doubt that a reference to a non-existent array element in the first form is an error, we cannot be nearly so sure that writing through `looper` when it no longer points to territory allocated to `buffer` is an error. When programmers type `buffer [looper]`, they are saying, ‘I want to reference an element of the array called `buffer`’; when programmers type `*looper`, they are merely saying, ‘I want to reference the integer pointed to by `looper`’. The initial assignment of `buffer` to `looper` is not sufficient to conclude that an array is being scanned.

Robust C will trap a reference to `buffer [max]` without additional syntax, but in order to trap the analogous error in the semantically weaker code segment, a `maxit` clause must be present:

```
{
  int *looper;
  for (looper=buffer;looper<=buffer+max;looper++; maxit max) /* Error here */
    *looper = 5;
}
```

## THE PROTOTYPE

We have constructed a limited prototype which implements some of the features discussed above. Many of the examples you have seen in this paper were formatted input and output of the prototype preprocessor. The prototype is a one-pass preprocessor whose only input is a single source file and whose outputs are the preprocessed source along with a header file. The functionality of the prototype is limited by the fact that it is first in the chain of compilation, prior to the regular C preprocessor. This was done to keep the prototype small and simple. A full implementation should be in the form of a plug-in replacement for the C preprocessor. That way, any implementation conflicts which might arise between the Robust C functions and the regular C preprocessor functions can be transparently resolved.

We found that the task of the preprocessor was complex because there is a wide gap between the services provided by the C language and the services which we want. To make the same extensions to C++ is much simpler because the gap is not so wide; however, the procedure is still complex enough that we stand to gain much from automating it.

## CONCLUSIONS AND FUTURE WORK

We have presented an extended version of the C language which eliminates some of its weaknesses and makes it safer to use. Our syntactic extensions enable the efficient trapping of classes of errors which have not been handled effectively in the past. We have created a prototype preprocessor to implement some of the extensions. A full implementation, based on non-proprietary preprocessor source and distributed free of charge, could provide many benefits to C programmers world-wide.

Future development of Robust C will enable it to provide more services to

programmers and detect more classes of errors. The possibilities for enhancing C++ will be more fully explored, and several levels of automatic class generation will be provided for more concise object-oriented programming. Many possible improvements will no doubt suggest themselves once the first test implementation is made available.

#### ACKNOWLEDGEMENTS

We wish to thank the editor and referees for their very helpful comments and suggestions, and David Rosenblum for providing us with a copy of his work.

#### REFERENCES

1. R. P. Gabriel, 'Lisp: good news, bad news, how to win big', *AI Expert*, **6**, (6), 31–39 (1991).
2. J. L. Steffen, 'Adding run-time checking to the Portable C compiler', *Software—Practice and Experience*, **22**, (4), 305–316 (1992).
3. B. Stroustrup, 'The evolution of C++: 1985 to 1989', *Computing Systems*, **2**, (3), 191–250 (1989).
4. J. Voas, L. Morrel and K. Miller, 'Predicting where faults can hide from testing', *IEEE Software*, **8**, (2), 41–48 (1991).
5. D. M. Andrews and J. P. Benson, 'An automated program testing methodology and its implementation', *Tutorial: Software Testing & Validation Techniques*, IEEE Computer Society Press, 1981.
6. D. M. Andrews and J. P. Benson, 'An automated program testing methodology and its implementation', *Proceedings, 5th International Conference on Software Engineering*, 1981.
7. D. S. Rosenblum, 'Towards a method of programming with assertions', *Proceedings, 14th International Conference on Software Engineering*, Melbourne, Australia, 1992.
8. S. C. Kendall, 'Bcc: runtime checking for C programs', *USENIX Toronto 1983 Summer Conference Proceedings*, USENIX Association, El Cerrito, CA, 1983.
9. S. Kaufer, R. Lopez and S. Pratap, 'Saber-C: an interpreter-based programming environment for the C language', *USENIX San Francisco 1988 Summer Conference Proceedings*, USENIX Association, El Cerrito, CA, 1988.
10. S. C. Johnson, 'A portable compiler: theory and practice', *Fifth ACM Symposium on Principles of Programming Languages Conference Record*, ACM, New York, 1978.
11. United States Department of Defense, *Reference Manual for the Ada Programming Language*, U.S. Government Printing Office, Washington, D.C., ANSI/MIL-STD-1815A-1983.
12. P. J. Plauger and J. Brodie, *Standard C*, Microsoft Press, Redmond, Washington, 1989.
13. American National Standards Institute Technical Committee X3J11 on Programming Language C, *Draft Proposed American National Standard Programming Language C*, 1988.