Chairperson: *Joseph Yao*
Hadron, Inc.

Bcc: Runtime Checking for C Programs

Samuel C. Kendall

Delft Consulting Corporation

165 West 91st Street, Suite 2A
New York, NY 10024
(212) 624-1149
decvax!genrad!wjh12!kendall

bcc is a preprocessor which generates bounds checking code for C programs. It will check array indices, pointers into arrays, and pointers into space allocated by *malloc*().

bcc generates diagnostics that tell the user which subscript or pointer variable overstepped the bounds of which array on which line of your C source program. Quite a bit nicer than the usual core dump.

With *bcc*, your program will take about 3 times longer to compile and ten times longer to run. As this is not a production compiler, speed isn't a primary consideration anyway.

bcc can be used with makefiles, but it's a little tricky.

bcc's job in life is to detect bounds check errors, and it's especially valuable, because boundary errors don't always cause a program to fail immediately. *bcc* can be used to test program integrity and uncover errors that would otherwise lay dormant, thereby saving many bleary-eyed hours staring at the CRT screen.

Bcc: Runtime Checking for C Programs

Samuel C. Kendall
Delft Consulting Corporation
165 West 91st Street
New York, NY 10024
(212) 362-0753
!decvax!genrad!wjh12!kendall

Abstract

Runtime error checking is more difficult for C than it is for most computer languages, but is no less necessary. Bcc* is a new software product which performs runtime checking for C. We discuss what errors are caught by bcc, and the implementation of bcc; examples of its use are also presented.

1. Introduction

The C language{Ritchie} has been in use for ten years, but no runtime checking facilities have been available. This contrasts with languages such as Pascal{Joy, Graham and Haley}{LeBlanc and Fischer} and Fortran, which boast runtime checking tools (typically checkout compilers) on many systems, and Ada**{Ada Standard}, which specifies that implementations must perform many runtime checks. It is no surprise, then, that runtime checking is much more difficult for C than for Pascal, Fortran or Ada. This difficulty is due to the free availability of pointers and pointer arithmetic in C; in contrast, pointers are restricted in Pascal and Ada, and nonexistent in Fortran. Pointers are the source of most runtime errors in C programs, as well as the source of the difficulty in runtime checking.

But as C moves far beyond its history as a replacement for assembly language, this lack of runtime checking becomes less and less justifiable.

The program in Figure 1a attempts to zero the elements of an array using a for loop on an array index. In actual usage there would be no printf in the loop to show us the index values each iteration; the printf would be added in an attempt to find the bug. This sort of ad hoc debugging is often necessary in current C programming, but it is a time-consuming and not always reliable diagnostic tool. For instance, the problem with this program would have been more evident had the printf been placed above, rather than below, the assignment on line 5.

*Bcc is a trademark of Delft Consulting Corporation.

**Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

FIGURE 1a

```
$ cat -n autoloop.c
     1  main(){
     2          int i, a[3];
     3
     4          for(i = Ø; i <= 3; ++ i){
     5                  a[i] = Ø;
     6                  printf("%d ", i);
     7          }
     8  }
$ cc autoloop.c          # compilation
$ a.out                  # execution
Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1
2 Ø 1 2 Ø 1 2 Ø 1 2 Ø 1 2 Ø ^C
$
```

FIGURE 1b

```
$ bcc autoloopl.c        # bcc compilation
$ a.out                  # bcc execution
Ø 1 2
autoloopl.c(5): array index too large

autoloopl.c(5):
        a[i] = Ø;
----->     ^

1 element past high end of auto array[3] of 4-byte elements
$
```

This program typifies a kind of error that occurs frequently in C programs which compile and even lint correctly: an array index which is out of bounds, or the indirection of a pointer which is null or out of bounds.

Bcc is a new software product which detects this kind of error. The bcc command is used exactly as cc(1), the C compiler command, is used. This paper discusses what we call a runtime error, and how this checking is implemented in bcc. We also present examples of erroneous C programs; Figure 1a is the first example compiled normally, and Figure 1b shows the same program compiled using bcc.

As shown in Figure 1b, the bcc command, like the cc command, produces an executable file a.out. This file, when run, outputs the three legal index values before terminating in a bcc error. The message given on the error output stream is always a concise, one-line summary of the problem, followed by various additional information: a longer explanation, if necessary (it is not necessary for "array index too large"); a display of the exact place on the line where the error occurred; and more information about the offending pointer or array index value. The octal or hexadecimal pointer value is optionally shown; we have chosen to omit it in our examples.

Figures 2a and 2b show the same erroneous program, rewritten to use a pointer directly instead of an array index. The program's normal

FIGURE 2a

```
$ cat -n autoloop2.c
     1  main(){
     2          int *p, a[3];
     3
     4          for(p = a; p <= a + 3; ++ p){
     5                  *p = Ø;
     6                  printf("%d ", p - a);
     7          }
     8  }
$ cc autoloop2.c
$ a.out
Ø 1 2 -5Ø7529 Memory fault - core dumped
$
```

FIGURE 2b

```
$ bcc autoloop2.c
$ a.out
Ø 1 2
autoloop2.c(5): attempt to indirect pointer exceeding high bound

Attempt to indirect a pointer which points past the end of the
object it should point into.

autoloop2.c(5):
        *p = Ø;
----->  ^

1 element past high end of auto array[3] of 4-byte elements
$
```

behavior (Figure 2a) is an interesting C puzzle.[1] The error message in Figure 2b is simply the pointer analogue of the array error message in Figure 1b.

2. Errors Caught by Bcc

It is clear that the errors detected in Figures 1b and 2b are properly errors. But some cases are less clear. For instance, should it be an error for pointer arithmetic to take a pointer beyond the end of an array, even if the resulting out-of-bounds pointer is never indirected? Probably not, since it is common practice for just that to happen in loops. But how far should a pointer be allowed to stray, since it will eventually reach the highest or lowest address and wrap around, causing pointer comparisons to yield the wrong result? Bcc checks for wraparound every time an integer is added to or subtracted from a pointer. The defect of this approach is that a program which barely avoids wraparound on one machine may not avoid it on another machine.

TABLE 1: OPERATIONS PRODUCING POINTERS

Operation	Bounds Enclose
address of array element	the outermost array
address of function	the function entry point
address of any other object	the object
malloc(3) allocation	the allocated region
brk(2) allocation, or &end	the entire heap (high end dynamic)
pointer cast from constant integer	no object (no indirection allowed)
pointer cast from non-constant integer	the entire address space
all others	(no change in bounds)

[1]The run in Figure 2a took place on computer where small addresses, including Ø, cause a "memory fault". The program's behavior on machines where small addresses are legal would be different, but still bizarre.

TABLE 2: OPERATIONS ON POINTER VALUES

		Possible Error Conditions			
Operation	Operators	null	out of bounds	out of alignment	others
indirection	* -> []	x	x	x	
plus or minus an integer	+ - ++ -- [] += -=	x			1
subtraction	-	x		x	2
equality	== !=				
relational	<= < > >=	x			
copying	= () return				
pointer cast	(type *)			x	
other cast	(type)				
use as a truth value	! && || ?: if() etc.				

1. Wraparound
2. Relative bounds and relative alignment

Unfortunately, the C Reference Manual does not cover this point, leaving actual usage the de facto definition. And actual usage does not permit bcc to check more stringently.[2]

We established rules for what is an error and what is not. Some of our choices are debatable; there are few guidelines on this matter in the C Reference Manual{Ritchie}, and some of the guidelines there are incorrect.

Our basic principle is that every pointer has a runtime association with the object it points into. In order to remember this association every pointer is made to include a pair of bounds, a low bound and a high bound, which enclose the associated object. There are two classes

[2]Problems of this sort, where we desire stricter checking than usage in existing programs permits, will be solved in later versions of bcc by implementing the stricter checking, but making it optional.

of C operations about which decisions had to be made: those which produce pointers and set bounds, and those which act on pointers and which may check bounds (check the pointer against its bounds), or check other error conditions. Table 1 shows the operations which produce pointers, and what bounds are assigned in each case.

An important principle that can be abstracted from these rules is that whenever pointer bounds enclose an array element, they also enclose the entire array, be it single-dimensional or multi-dimensional; but the same is not true for structure elements and structures.

Table 2 shows the operations on pointers, and what kinds of errors can be generated from them. For instance, if you indirect a pointer, an error can be given because the pointer is null, or because it is out of bounds, or because it is out of alignment.

The checks covered in the column "others" are: "relative bounds", which fails for pointers the bounds of which do not enclose the same array, and "relative alignment", which fails for pointers to objects not an integral number of object-sizes apart; and "wraparound", which fails for arithmetic operations that cause a pointer to "wrap around" the address space.

An interesting decision that had to be made is shown in Table 2 by the difference in error checking between pointer subtraction and pointer relational operators. The conditions for subtraction are quite restrictive. But the only condition checked for relationals is that neither pointer is null, and even the reason for this condition—that the C Reference Manual does not guarantee a null pointer value to be less than all other pointer values—is debatable. We differentiate between pointer subtraction and pointer relationals because subtraction makes sense only with pointers that point into the same array, whereas there are portable uses for comparing pointers that point into different arrays—despite what the C Reference Manual claims.

Uninitialized pointers are not explicitly checked for, but are almost always caught, since it is very unlikely that garbage will look like a legal pointer and bounds. Some errors, most notably dangling pointer, integer overflow, and uninitialized non-pointer, are not caught by the current version of _bcc_. Nonetheless, _bcc_ does catch the overwhelming majority of C runtime errors.

3. Examples

Figures 3a and 3b demonstrate the _bcc_ versions of _malloc_ and a few other standard library functions which use pointers. All pointer usage in library functions is checked.

The program in Figure 3a seems to lack errors—it runs without fault—but it actually allocates an area one byte too small on line 5. Errors of this nature can and do remain in production programs, surfacing only as occasional reliability or security problems, very difficult to find. To find such errors, _bcc_ can be used for reliability testing as well as for more ordinary debugging. The _bcc_'d version of a program

FIGURE 3a

```
$ cat -n usemalloc.c
     1  char s[] = "A string.";
     2  main(){
     3          char *p, *malloc(), *strcpy();
     4
     5          p = malloc(strlen(s));
     6          printf("Copied: %s\n", strcpy(p, s));
     7  }
$ cc usemalloc.c
$ a.out
Copied: A string.
$
```

can be run through a test suite, or even installed as the production version for a time in order to test the program against normal (or malicious) user data.

As in Figure 3b, the error messages given for errors involving pointers to heap storage are particularly informative, since they

FIGURE 3b

```
$ bcc usemalloc.c
$ a.out

usemalloc.c(6): lib function strcpy( arg 1 ): array too small

The library function expected the array to be larger than it
actually is.

called from usemalloc.c(6):
        printf("Copied: %s\n", strcpy(p, s));
----->                              ^

expected:  array[1Ø] of 1-byte elements

element Ø of heap array[9] of 1-byte elements
allocated in usemalloc.c(5):
        p = malloc(strlen(s));
----->            ^
$
```

display exactly where the storage was allocated, as well as where the error occurred.

Our fourth and last example, Figures 4a and 4b, displays an often-found nonportable code fragment. This access to location zero gives a memory fault on most VAX-11* and MC68ØØØ systems. But on the PDP-11*, one is allowed to access location zero (in defiance of the Reference Manual), which is why so much code was written that does exactly what this example does. This coding practice can be extremely annoying to UNIX** porters; for instance, the abstract for one porting talk at the January 1983 UNICOM sarcastically promised to describe "the true meaning

FIGURE 4a

```
$ cat -n nullptr.c
     1  main(argc, argv)  int argc; char *argv[];
     2  {
     3          if(argv[1][Ø] == '-' && argv[1][1] == 'a')
     4                  printf("-a option given.\n");
     5  }
$ cc nullptr.c
$ a.out -a
-a option given.
$ a.out
Memory fault - core dumped
$
```

FIGURE 4b

```
$ bcc nullptr.c
$ a.out -a
-a option given.
$ a.out

nullptr.c(3): attempt to indirect a null pointer

nullptr.c(3):
        if(argv[1][Ø] == '-' && argv[1][1] == 'o')
------>           ^
$
```

*PDP-11 and VAX-11 are trademarks of Digital Equipment Corporation.

**UNIX is a trademark of Bell Laboratories.

of location zero and its relevance to UNIX." Figure 4b shows bcc's response to the problem.

One might want to simulate a machine where access to location zero is legal, to simply give a warning message when location zero is accessed. Bcc's responses to certain error situations, this one among them, can be changed by setting an environment variable.

4. Implementation

Bcc is a source-to-source transformer which invokes the normal C compiler on the result of a transformation. The actual source-to-source transformation program is called the "mapper", for historical reasons. Figure 5 depicts the logical structure of the mapper; the three phases shown are logical divisions of the code, rather than separate processes. The mapper actually consists of mapc, a table-driven source-to-source transformation tool, and "the map", the table of parse tree transformation rules that drives it. Given an empty map, mapc makes essentially no change in a program; but given the map used for bcc, it makes pervasive changes, as illustrated in Figure 6.

In addition to the mapper, there is an extensive library of encapsulations. Each function in the standard library which takes or returns

FIGURE 5

```
                                   ---------- BCC's MAPPER ----------
                                   |                                |
                                   |          -------------         |
(from             preprocessed C   | text     |           |         |
 C              ---------------------------->|  PARSING  |         |
 preprocessor)                     |          |           |         |
                                   |          -------------         |
                                   |          parse | trees         |
                                   |                V               |
                                   |          -------------         |
                                   |          |           |         |
                                   |          |  MAPPING  |         |
                                   |          |           |         |
                                   |          -------------         |
                                   |          parse | trees         |
                                   |                V               |
                                   |          -------------         |
(to                     mapped C   | text     |           |         |
 C              <-----------------------------| UNPARSING |         |
 compiler)                         |          |           |         |
                                   |          -------------         |
                                   ----------------------------------
```

FIGURE 6: BEFORE AND AFTER SOURCE-TO-SOURCE TRANSFORMATION

This transformation was targeted to an MC68ØØØ, on which ints and pointers are 4 bytes long but are aligned only on even-byte boundaries.

BEFORE:

```
f(){     /* nonsense function */
         int a[1Ø], i, *p;
         *p++ = a[i];
}
```

AFTER:

```
 int f(Zfilenam,Ztokno) char (*Zfilenam); int Ztokno;
{register char (*(*Zta));
auto int a[1Ø];auto int i;auto int (*p[3]);
*(int (*))Zchkind(Zincr(p, 4, 2, "file.c", 18), 4, 2,
"file.c", 16) = *(int (*))Zchkab(a, 4Ø, 4, 2, i,
"file.c", 21);
}
```

a pointer must be separately encapsulated. In the bcc world, pointers include a low and high bound; each encapsulation is an error-checking interface between the bcc world and a normal library function.

5. Conclusion

We chose the source-to-source approach because it is a portable approach. Bcc is almost trivially portable except for the differences in standard libraries between versions of UNIX.[3] However, this source-to-source approach has its costs. A bcc compilation takes about three times as long as a normal compilation; and the resultant executable file is 2 to 3 times larger, and executes an order of magnitude slower, than the corresponding result of a normal compilation.

The C programmer, valuing efficiency, may be tempted to ask whether he or she can afford runtime checking. The C programmer values efficiency One can first note that bcc is not intended to be used for every compilation; nor is the bcc executable intended for production use. Runtime errors are usually found near the start of execution; in our experience, the execution slowdown has not been a problem.

[3]Because each library function which takes or returns a pointer must be separately encapsulated, bcc is much easier to port between different CPUs running the same version of UNIX than it is to port between identical CPUs running different versions of UNIX.

Moreover, one cannot ignore human efficiency. Every C programmer can probably remember times when he or she spent hours, even days, searching for a bug which could have been located almost immediately with runtime checking. And equally important, runtime checking can find bugs whose existence would not otherwise have been noticed, bugs which would have become reliability or security problems.

Acknowledgements

Chaim Schaap was responsible for many key insights and design principles. The advice of Mike Douglas was also invaluable. Chaim Schaap, Jahanshah More, Allen Wolovsky, and Charles Perkins were helpful in shaping this paper.

6. Bibliography

Ada Standard
Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A (January 22, 1983).

Johnson
Johnson, S. C., Lint, a C Program Checker, in various UNIX documentation collections (1978).

Joy, Graham and Haley
Joy, William N., Susan L. Graham, and Charles B. Haley, Berkeley Pascal User's Manual—Version 2.Ø (198Ø).

LeBlanc and Fischer
LeBlanc, R. J., and C. N. Fischer, "A Case Study of Run-Time Errors in Pascal Programs", Software—Practice and Experience, Vol. 12, 825-834 (1982).

Ritchie
Ritchie, D. M., "C Reference Manual", Appendix A of B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, Inc. (1978).