

# Backwards-compatible bounds checking for arrays and pointers in C programs

Richard W M Jones and Paul H J Kelly

Department of Computing

Imperial College of Science, Technology and Medicine

180 Queen's Gate, London SW7 2BZ

## Abstract

This paper presents a new approach to enforcing array bounds and pointer checking in the C language. Checking is rigorous in the sense that the result of pointer arithmetic must refer to the same object as the original pointer (this object is sometimes called the ‘intended referent’). The novel aspect of this work is that checked code can inter-operate without restriction with unchecked code, without interface problems, with some effective checking, and without false alarms. This ‘backwards compatibility’ property allows the overheads of checking to be confined to suspect modules, and also facilitates the use of libraries for which source code is not available. The paper describes the scheme, its prototype implementation (as an extension to the GNU C compiler), presents experimental results to evaluate its effectiveness, and discusses performance issues and the effectiveness of some simple optimisations.

## 1 Introduction and related work

C is unusual among programming languages in providing the programmer with the full power of pointers. Languages in the Pascal/Algol family have arrays and pointers, with the restriction that arithmetic on pointers is disallowed. Languages like BCPL allow arbitrary operations on pointers, but lack types and so require clumsy scaling by object sizes.

An advantage of the Pascal/Algol approach is that array references can be checked at run-time fairly efficiently, in fact so efficiently that there is a good case for bounds-checking in production code. Bounds checking is easy for arrays because the array subscript syntax specifies both the address calculation and the array

within which the resulting pointer should point.

A pointer in C can be used in a context divorced from the name of the storage region for which it is valid, its ‘intended referent’, and this has prevented a fully satisfactory bounds checking mechanism from being developed. There is overwhelming evidence that bounds checking is desirable, and a number of schemes have been presented. The main difference between our work and Kendall’s `bcc`[13] and Steffen’s `rtcc`[7] is that in our scheme the representation of pointers is unchanged. This is crucial, since it means that inter-operation with non-checked modules and libraries still works (and much checking is still possible). Compared with interpretative schemes like *Sabre-C*[14], we offer the potential for much higher performance. Patil and Fischer [10, 11] present a sophisticated technique with very low overheads, using a second CPU to perform checking in parallel. Unfortunately, their scheme requires function interfaces to be changed to carry information about pointers, so also has the inter-operation problem.

Another approach is exemplified by the commercially-available checking package Purify [6]. Purify processes the binary representation of the software, so can handle binary-only code. Each memory access instruction is modified to maintain a bit map of valid storage regions, and whether each byte has been initialised. Accesses to unallocated or uninitialised locations are reported as errors. Purify catches many important bugs, and is fairly efficient. However, Purify does not catch abuse of pointer arithmetic which yields a pointer to a valid region which is not the intended referent. Fischer and Patil [10, 11] provide evidence for the importance of this refinement.

Our goals in this paper are to describe a method of bounds checking C programs that fulfills the following criteria:

- Backwards compatibility — the ability to mix checked code and unchecked libraries (for which the source may be proprietary or otherwise unavailable)
- Works with all common C programming styles

























