Doctor, it hurts when I do this!

Submitted by Steve Lionel (Intel) (https://software.intel.com/en-us/user/512685) on March 31, 2008

It is often said that you can write bad code in any language, and I certainly can't argue with that. I do find, though, that the worst-looking code comes from programmers who are more familiar with another programming language. One can often tell that a C programmer wrote Fortran code, or that a Fortran programmer wrote C code (my C code probably looks like the latter.)

tainly can write clear and understandable code in Fortran, and many do. I see a lot of code from people cross my desk each day, and I've learned to recognize certain "idioms" in Fortran code Meaningful variable names, use of mixed-case ding practice can bite you. Here's one I ran

ay-oriented features. Few languages offer the often you can do an array operation without a

ace required!
S/forums/permalink/116158/116170
rew up on FORTRAN 77 or even FORTRAN
array is an array, so they'll write code that

rt of like the Dave Barry joke that an
in grocery store signs.) In Fortran syntax, (:)
nds at the last element - the whole array, in

compiler has to work extra hard to recognize piece of it. In the past, unnecessary use of (:) the case, thanks to hard work by the compiler me, the (:) is "harmless" in that it does not change the overall meaning of the program, since an array section is "definable" (can be stored into), as long as you don't use a vector subscript such as A([1,3,5,7]).

But there's another case where the (:) can bite you. If you are calling a procedure where the corresponding dummy argument is a deferred-shape array, then there is a difference in meaning to passing the array name A and passing A(:). If you just pass A, then the lower bound(s) are passed along and are reflected in the dummy argument inside the called routine. However, if you pass A(:), that's an array section and the lower bound is 1, no matter what you declared it as originally. This will change how the array is indexed in the called procedure and can cause array bounds violation errors.

In the recent case, the customer had written something like:

A(:) = func(B)

where A was an ALLOCATABLE array and function "func" returned an array. In Fortran 90 and 95, the language required that A already be allocated and have the same shape (dimensions) as the function return value.

Fortran 2003, however, added a new twist. In F2003, if the allocatable array on the left of the assignment is not currently allocated with a shape matching the right-side, it is automatically deallocated (if needed) and then allocated with the correct shape. This can make code a lot cleaner as you don't have to worry about knowing the shape in advance.

The downside, though, is that the checking required to support this is a lot of extra code, and applications where it is known that the array is already allocated to the correct shape don't need this check which would just slow them down. This is why Intel Fortran does not support this F2003 feature by default - you have to ask for it with the option /assume:realloc_lhs, or for Linux and Mac users, -assume realloc_lhs. ("lhs" here means "left hand side".)

The programmer who wrote the above code had in fact used this feature and depended on it, with array A starting out unallocated. He was therefore surprised to find that his program, when it got to the above assignment, complained that array bounds were violated!

It turned out that it was the use of the "helpful" (:) that caused the problem. This changed the meaning of the left-hand-side from an "allocatable variable" to an array section, and as such, it did not qualify for automatic reallocation. Consider, for example, if the code had read:

A(2:5) = func(B)

you could not reallocate some elements of an array section!

The solution in this case was to remove the superfluous (:), in which case the program ran as expected. On my advice, the customer also removed unnecessary (:) in other places as they could impede optimization.

So, Doctor Fortran's advice if you put (:) on your arrays? Don't do that!

Got any other examples of Fortran coding practices that can hurt you?

Categories: Fortran (/en-us/search/site/field_programming_language/fortran-20804/language/en)

Comments (11)

Top

Sergio said on Thu, 12/05/2013 - 10:43

I am so happy I found this. Every time I run into some weird problem, I end up in here learning about some obscure Fortran

The problem I found is that Fortran compilers seem to be very inconsistent on the bounds of LHS.

I have found two problematic cases: a) when the lower bound of the returned allocatable array is different than 1, and b) when LHS is allocated and has the same size as RHS, but different bounds. The following snippet to illustrate it:

```
program return_allocatable
  implicit none

  real, allocatable :: a(:)

  real, parameter :: b(-2:4)=[1,2,3,4,5,6,7]

  a=foo(3)
  print*,lbound(a),':',ubound(a)

  a=b
  print*,lbound(a),':',ubound(a)

contains
  function foo(n)
    integer :: n
    real, allocatable :: foo(:)

    allocate(foo(-3:n))
```

Rate Us

nailed it, but version 12.0.4 only gets the second test right! But in a different
nd 13.1.0) fail BOTH tests!

s allocation on assignment (but only if the shape does not match), one should use

the shape conformance check of the compiler. In other places than on the LHS of
oe avoided.

not performance sensitive (measure - don't guess!), sticking to the named object
easonable.

an use it (with bounds checking enabled) to check for mismatches which are not
would do) but logically wrong.

Anonymous said on Tue, 08/25/2009 - 04:22

Hi Steve,

I started to update my code with intents. Is there some way to tell the compiler to assume intent(inout) for all non-explicit intents. Otherwise I run into trouble if I forget to declare an explicit intent in a subroutine. This seems to be a save thing to do, or am I mistaken?

Regards,
Mike

Steve Lionel (Intel) said on Wed, 05/13/2009 - 04:42

Since there is an explicit interface for foo visible to the caller (you indicate there is), and dummy argument B is declared as assumed-shape, then no copy would be made even if the actual argument was not contiguous. The only time a copy is made is if the caller believes that the argument is accepted as a non-assumed-size array (and the slice being passed is not contiguous.)

In your case, A(:,1) is contiguous (since A is not a POINTER).

Steve

Anonymous said on Tue, 05/12/2009 - 15:44

I have been using a somewhat similar construct to send a single array from a multi-dimensional array to a subroutine (see example below). Will this construct suffer from compiler inefficiencies, or is it immediately recognized that the colon in A(:,1) represents the first array element?

```
program bar
real, dimension(:,:) :: A
allocate (A(1000,2))
call foo(A(:,1))
.
.
end program bar

subroutine foo(B)
! explicit interface
real, dimension(:) :: B
.
.
end subroutine foo
```

reinhold-bader said on Fri, 08/01/2008 - 23:55

Note that array intrinsics sometimes change the shape of their argument (PACK, TRANSFER) in which case both "A = " and
"A(:) = " won't work. In fact, with Fortran 95 semantics, you typically need to call the intrinsic twice, e.g.

: ! allocate a if allocatable, or check mysize <= size(a)
a(1:mysize) = pack(b, ...)
Fortran 2003 of course fixes this for the "A = " idiom, but only if A is allocatable. So I've wondered whether it might not be more appropriate to change the standard to simply throw away excess elements of the result, or leave excess elements of the LHS undefined instead of insisting on "same shape".

Steve Lionel (Intel) said on Tue, 07/29/2008 - 05:59

If you're talking about a 10-element array, there's no point in discussing optimization. Now if it's a 1000-element array or larger, it may be worth paying attention to.

If you are initializing an array, you want the initialization to be vectorized. You'll have the best chance of doing that with a whole-array assignment. Initializing a slice may introduce alignment issues that could inhibit vectorization. But if you know that you'll later be setting most elements of the array, it might pay to not initialize all of them. If your analysis shows that this section of code is a "hot spot" (using Intel VTune or some other profiler), then you can look at it closer.

My advice here is to turn on the vectorization optimization reports (See the Optimizing Applications section of the documentation) and see where the compiler can and can't vectorize your loops.

Rate Us

What I wanted to get across in this article was to avoid cluttering your code with unnecessary syntax.

ıot. I wonder how much of a difference these optimizations make? If I have a 1D
ıts- where my boundary conditions are applied after I update my interior points, I

ent array

A that are changing

ıputations (barring something really brilliant on behalf of the compiler), but if it
ıously as my domain size grows and the ratio of interior points to boundary points
ıding the boundary points on the first go-around drops off, but it'd be interesting to
ne time in the next week or two, I'll try to code up a simple test.

Given this situation, though, are there any recommendations you make regarding this sort of problem?

Thanks again!
- Brian

Steve Lionel (Intel) said on Mon, 07/28/2008 - 06:39

The difference is whether you specify a subscript or not. If you do, no matter what it is, the compiler has to do extra work to understand it and, in the case of (:) where it means the same thing as leaving it off, undo some of the assumptions it makes. The part of the compiler that parses the syntax does not always know at that time whether it's safe to remove the (:). Depending on where in the compiler this is done, it can inhibit optimization. We try to take care of such issues when we find them. Using (1:x) will definitely block some optimizations.

Since the (:) does change the semantics of the code, I recommend leaving it off unless there's a good reason to include it.

Steve

Anonymous said on Mon, 07/28/2008 - 06:30

Hi Steve,

I just stumbled across this page by accident, and I find I'm often guilty of coding my array operations with the explicit colon. I do this because many of the people with whom I work are still stuck on F77 syntax, and making array syntax lines explicit seems to help their understanding of the code. (In fact, in some cases I'll even specify the bounds - eg, A(1:x) ..)

Now, I've never written a Fortran compiler, but it would seem to me that it's pretty easy to have the compiler check for the common-but-special situations where the colon is explicit, no? Checking that the specified bounds (in the 1:x example) are the full bounds of the array would be quite a bit harder, I imagine, but what causes problems with the compiler for just checking the colon?

I'll re-write some of my code soon and see if this makes a difference. I trust the Doctor's advice, but wouldn't mind knowing more about why the medicine he prescribes is needed. ;)

Cheers,
- Brian

Add a Comment

Top

(For technical discussions visit our developer forums. For site or software product issues contact support.)

Please sign in to add a comment. Not a member? Join today ›