Subject: Re: REDUCE
Posted by Richard Younger on Sun, 01 Apr 2001 18:55:31 GMT
View Forum Message <> Reply to Message

John-David Smith wrote:
>
> Kenneth Mankoff wrote:
>>
>>> The question, to all you C-programmers: is there a better way?
>> [snip]
>>> ...the code logic to compute the maximum will be the same, both
>>> symbolically for all types for many types, in the compiled code itself.
>>
>> Hi JD,
>>
>> hmmm... not 100% sure, but wouldn't c++ templates solve this problem?
>>
>> And for the cases where it is "symbolically" the same but not "compiled
>> the same", I'm not sure what this means, but I'm guessing you would handle
>> these cases with overloading your operators.
>>
>> Of course, C isn't C++, so this might not help.
>>
>> I can provide code examples and more info if you wish.
>
> Thanks for the suggestion. I had thought of that option, but I don't know much
> about templates, nor about linking C++ to IDL. I wonder whether the templates
> are just similar to my super macro for creating a different version for each
> type. Can you frame the maximum function I suggested in terms of a skeleton
> template which would operate on all the data types?
>
> My comment with respect to compiled and symbolic maybe wasn't clear. I really
> just meant that you have this same code replicated over and over, with minor
> changes in the types of the variables used, but otherwise logically and
> symbolically intact. I can imagine the compiler emitting different code for,
> e.g., multiplying two integers, vs. two floats, but I can also imagine other
> types where the codes emitted are exactly the same. Obviously, you can't get
> something for nothing, but if real repition exists within the compiled code, you
> should be able to eliminate it somehow.
>
> Thanks again,
>
> JD

Hi JD and Ken,

I agree with Ken that the most obvious and easiest C++ solution is
templates and operator overloading. I have a little experience DLMing

with C++, and it works just fine. The calling conventions of C and C++ can be set exactly the same, so the limitations are exactly the same.

What is the distinction between the different cases? Are you primarily worried with arithmetic, indirection, or member changes. e.g:

```
(float a * 2)  vs  (int a * 2) or
(float*)a  vs  (int*)a  or
2*value.f  vs  2*value.i
```

The overloading and template solutions work well on the first two problems, and not well at all on the last category, because AFAIK there's no good, compact way to make run-time distinctions with members. It's because different explicit symbols are used, as opposed to different implicit types. You end up using lots of switch - case statements. I suppose you could put the switch into an operator to extract the value of a data element, but then you end up switching every time you access an array element, instead of once at the beginning. I'd think it would be slower than your super-macro. Maybe someone else knows a better solution.

And I know that IDL uses unions in its variable definitions, so I suspect that you are running into the third problem.

As for using a nifty silver bullet C++ feature, the use of unions is generally discouraged in C++ for the above reasons. I suppose you could go heavy-duty C++ and start throwing around wrapper classes and Standard Template Library iterators, but I'm just barely familiar with that stuff (iterators anyway), and don't know enough to really know if that solution is simple or fast or not. Besides, I don't think JD is that interested in learning lots of C++ language semantics for limited return.

On a side note, Your REDUCE package seems to be very similar to a feature that I really would like RSI to implement; namely, Einstein-summation or dummy-index notation. Something that would result in operations like

```
epsilon = fltarr(3, 6, 9)
E_one = fltarr(9)
E_two = fltarr(6)

epsilon[%1, %2, %3]*E_one[%3]*E_two[%2]
```

would multiply the elements of epsilon by E_one on the corresponding (3rd) index and sum over that index. Especially for those of us working with lots of fields in tensor notation, it would save lots of for loops,

and I'm sure that a built-in facility would save time over the for loop.

Has anyone else found themselves yearning for such a feature?

Rich

--
Richard Younger
Assistant Technical Staff MIT Lincoln Laboratory
Electro-Optical Materials and Devices Mail Stop C-130
Email: younger@ll.mit.spmdecoy.edu 244 Wood St.
Phone: (781)981-4464 Lexington, MA 02144-9108
