— March 8, 2025 —
Introduction to Fortran Templates
Accumulator Example

Magne Haveraaen[1], Thomas Clune[2], Brad Richardson[3], Aury Shafran[4], Ondřej Čertík[5], and Damian Rouson[6]

[1] Bergen Language Design Laboratory
Department of Computer Science, University of Bergen, Norway
https://bldl.ii.uib.no/
[2] NASA GSFC — thomas.l.clune@nasa.gov
[3] NERSC — brad.richardson@lbl.gov
[4] Intel — aury.shafran@intel.com
[5] LFortran — ondrej@certik.us
[6] Lawrence Berkeley National Laboratories, University of California — rouson@lbl.gov

Abstract. This a document in a series of pamphlets introducing Fortran 2028 templates. Each pamphlet is centred around an example, showing how to templetise it.
This pamphlet uses the *accumulate* operator as its motivating example. Accumulate is known by instances such as *sum*, *product*, or finding the least or largest element in an array.

1 Introduction

Accumulate, in its many instances (e.g., sum, product, max, min, general reduction), is a fundamental function in numerical computing. It is a core part of matrix multiply, and central to artificial intelligence and graph algorithms. Fortran has intrinsic support for sum, product, max and min of arrays of intrinsic number types, and special reduction functions on arrays of the intrinsic logical type. It also has a general reduction function for intrinsic arrays of any element type and user defined binary function. Fortran's support for accumulate functions does not cover collection types other than intrinsic arrays, e.g., sparse arrays are not supported. Sparse arrays and sparse matrices are important for numerical solvers, AI, graph algorithms, and more, and applications in these domains require a varied collection of accumulate functions.

Here we develop a templated accumulate function using Fortran's proposed type-safe template mechanism. For simplicity we implement accumulate for standard arrays of elements of arbitrary types, and use this setting to discuss various aspects of Fortran's type-safe templates.

1.1 Background: Fortran Templates

For decades the ISO and INCITS Fortran committees, informally known as WG5 and J3, respectively, have considered proposals for template generics.[7] The work towards a generics mechanism got traction at the Fortran meeting July 2019 in Tokyo with the presentation of the paper [1].

Dan Nagle (1953-2023) from NCAR, head of J3 at the time, initiated a "generics" subgroup with Tom Clune as its lead. This work has lead to a *type-safe template* proposal.[8]

The LFortran project provides a prototype compiler for the Fortran template mechanism.[9] LFortran's template syntax deviates from the official INCITS Fortran proposed notation. The examples in this paper uses the LFortran notation, and can be compiled and run with the LFortran compiler.

[7] Peruse the archive of the INCITS Fortran committee at https://www.j3-fortran.org for these.
[8] See the relevant papers on https://j3-fortran.org/doc/year/25
[9] See https://lfortran.org for the LFortran project. On https://dev.lfortran.org an in-browser version of the compiler and runtime system is available. The in-browser compiler also contains a selection of template Fortran example codes.

2 The accumulate function

The accumulate function reduces (folds) the values in an array down to a single value. It can do so by applying a binary function to the elements of the array. If we apply an operation which is associative and commutative, the order we apply it to the array elements does not matter. Mathematical functions, such as `operator(+)` and `operator(*)`, are associative and commutative, hence sum and product are well defined independently of which order addition and multiplication are applied to the array elements.[10] However, if we wanted to use an operation like `operator(-)` which lacks these properties, we would have to be really careful in defining which order the subtractions should take place on the array elements.

In the remainder of this pamphlet, we will assume the binary function for accumulate will be associative and commutative. Then we will not have to worry about in which order the operation is performed. We can still instantiate the template with `operator(-)`, but the result would then be very dependent on the internal algorithm used for accumulate. Such algorithms may be changed without notice, since the proper use of accumulate is not sensitive to the traversal order for the elements.

The rest of this section does the following.

- In the first subsection we present a summation function on integers and discuss how to make it more generally applicable.
- In the second subsection we develop reusable requirements for the accumulator. These allow us to use compact declarations of template arguments later on.
- In the third subsection we turn the integer summation function into a templetised accumulator.

2.1 Summing integers

We start with a straight forward implementation of a summation function for integers.

```
10        function isum ( arg ) result(res)
11            integer, intent(in) :: arg(:)
12            integer, intent(out) :: res
13
14            integer :: acc
15            integer :: i
16            acc = arg(1)
17            do i = 2, size(arg)
18              acc = acc + arg(i)
19            end do
20            res = acc
21        end function
```

We can make this generic in the accumulator function by replacing the addition operation with a template argument binary function `bin`, and replacing the hardcoded array element type `integer` with a generic type argument `T`.

2.2 Defining a requirement

To do full type-checking of a template body, the Fortran template mechanism requires that all template arguments are fully declared. We do the declarations as *requirements*. A `requirement` provides a compact reusable collection of declarations. Below we declare a requirement for a binary function. It allows each of the two arguments to have different types, and the return value to have a third type.

[10] Note that associativity does not hold for floating point numbers. A follow-up pamphlet will look at accuracy and overflow problems and suggest one approach to deal with them.

```
10      requirement binop_r( T, U, V, op_func )
11          type, deferred :: T
12          type, deferred :: U
13          type, deferred :: V
14          pure elemental function op_func(lhs, rhs) result(res)
15              type(T), intent(in) :: lhs
16              type(U), intent(in) :: rhs
17              type(V) :: res
18          end function
19      end requirement
```

Note the `deferred` keyword on the lines introducing the type names. These indicate that these types are dummy template arguments. In newer revisions of the generics proposal the function declaration would be in an interface block marked as deferred.

2.3 A plain template implementation

Here we generalise the integer summation function above to an accumulate function. The `accumulate` function can accumulate arrays with elements of type `T` using a binary function `bin` which takes arguments of type `T` returning values of the same type.

```
10      template accumulator_t ( T, bin )
11          require :: binop_r ( T, T, T, bin)
12        contains
13          function accumulate( arg ) result(res)
14              type(T), intent(in) :: arg(:)
15              type(T) :: res
16
17              type(T) :: acc
18              integer :: i
19              acc = arg(1)
20              do i = 2, size(arg)
21                acc = bin ( acc, arg(i) )
22              end do
23              res = acc
24          end function
25      end template
```

The structure of the `accumulate` function is the same as that of the `isum` function, There is a minor adjustment changing the infix call of `operator(+)` to calling the function `bin`, and changing the type of the array alements (and accumulator variable `acc`) to `T`. The function is now defined inside a template `accumulator_t` which takes both the type `T` and function `bin` as arguments. The declarations of these arguments are given by the require statement using the requirement `binop_r`.

We can now instantiate the template for summing integers defining the function with name `my_sum_i`. To show the flexibility, we also instantiate the template for summing reals and multiplying complex numbers, giving the functions with name `my_sum_r` and `my_prod_c`, respectively. After each instantiate statement, we have placed a print statement with the result of calling the instantiated function.

```
10    instantiate accumulator_t ( integer, operator(+) ), &
11        only : my_sum_i => accumulate
12    print *, my_sum_i(idata)
13    instantiate accumulator_t ( real, operator(+) ), &
14        only : my_sum_r => accumulate
15    print *, my_sum_r(rdata)
16    instantiate accumulator_t ( complex, operator(*) ), &
```

```
17        only : my_prod_c => accumulate
18    print *, my_prod_c(cdata)
```

We assume that arrays `idata`, `rdata` and `cdata` contain integer, real and complex data respectively.

Note that we do not need to wrap the addition operator or the multiplication operator for instantiating the template. The precise declarations of the template arguments by the requirement `binop_r` allows the compiler to decide which function from an overloaded set is to be used. In Fortran overload sets, such as those for the operators, are said to have *generic names*. This handling of generic names is different from passing a function to a procedure, where a *specific* name (in Fortran terminology) has to be used. Ensuring specific names often forces us to provide wrappers for functions and operators.

3 Summary

With the accumulator example we have shown how similar template programming in Fortran is to normal programming. We have also shown how flexible the instantiation mechanism is. It allows us to choose the type and the binary operation to be used, and can resolve generic names (Fortran terminology for overload sets) without the need for wrappers.

Fortran's template mechanism is type-safe, giving the same error messages on template code as you would expect for ordinary code.[11] Such type-safety is a productivity factor for writing template code. The appendix contains the full source code for the templated accumulate function and some instantiations. This can be compiled and run with the LFortran compiler.

This pamphlet does not cover accuracy and overflow issue for array reductions. These will be discussed in a follow-up pamphlet.

Fortran already has intrinsic support for accumulate functions `sum`, `product` etc for intrinsic arrays. The benefit of an accumulate template becomes more obvious when we consider user defined array data structures, such as those for large sparse arrays.[12] In such cases the traversal algorithm for the accumulate function becomes non-trivial. Implementing the function once, then instantiating it for the relevant binary functions boosts productivity and eliminates copy-paste errors.

References

1. Haveraaen, M., Järvi, J., Rouson, D.: Reflecting on generics for Fortran (2019), {`https://j3-fortran.org/doc/year/19/19-188.pdf`}, presented at the Fortran Standardisation Meeting, ANSI PL22.3 (J3) & ISO/IEC JTC 1/SC 22/WG 5 in Tokyo (Japan), 2019-08-05 through 2019-08-09.

A Code for the templated accumulate function with tests

Code for the template accumulator function with instantiation examples and some simple tests.

```
module Requirements_m
    !! Declaration of a binary function
    !! Take arguments of type T and U and return a value of type V
    requirement binop_r( T, U, V, op_func )
        type, deferred :: T
        type, deferred :: U
        type, deferred :: V
        pure elemental function op_func(lhs, rhs) result(res)
            type(T), intent(in) :: lhs
            type(U), intent(in) :: rhs
```

[11] This is a significant deviation from the template mechanism in C++ that many developers are familiar with.

[12] See GraphBLAS at `https://graphblas.org` for examples of such codes.

```
          type(V) :: res
       end function
    end requirement
end module


module Accumulator_m
    use Requirements_m
    implicit none
    private
    public :: accumulator_t
    !! Template for the accumulate function
    !! Takes a type template argument T and a binary function template argument bin
    template accumulator_t ( T, bin )
       require :: binop_r ( T, T, T, bin)
      contains
       !! The accumulate function
       !! Takes an array of argument and
       !!  accumulates the elements using the binary function bin
       function accumulate( arg ) result(res)
          type(T) :: res
          type(T), intent(in) :: arg(:)

          type(T) :: acc
          integer :: i
          acc = arg(1)
          do i = 2, size(arg)
            acc = bin ( acc, arg(i) )
          end do
          res = acc
       end function
    end template
end module

program TestAccumulator
  use Accumulator_m
  integer :: idata(5)
  real :: rdata(5)
  complex :: cdata(5)

  print *, "TestAccumulatorTemplate"
  idata = [ 1, 2, 3, 4, 5 ]
  rdata = idata
  cdata = rdata
  instantiate accumulator_t ( integer, operator(+) ), only : my_sum_i => accumulate
  print *, my_sum_i(idata), "= sum(", idata, ")"
  instantiate accumulator_t ( real, operator(+) ), only : my_sum_r => accumulate
  print *, my_sum_r(rdata), "= sum(", rdata, ")"
  instantiate accumulator_t ( complex, operator(*) ), only : my_prod_c => accumulate
  print *, my_prod_c(cdata), "= prod(", cdata, ")"
end program
```

Output from compiling and running the test program using the LFortran processor version 0.36.0 on a Linux system.

```
TestAccumulatorTemplate
15 = sum( 1 2 3 4 5
)
1.50000000e+01 = sum( 1.00000000e+00 2.00000000e+00 3.00000000e+00 4.00000000e+00
    5.00000000e+00
)
(120.000000,0.000000) = prod( (1.000000,0.000000) (2.000000,0.000000)
    (3.000000,0.000000) (4.000000,0.000000) (5.000000,0.000000)
)
```