

# Experimenting with generic programming features

Arjen Markus  
arjen.markus@deltares.nl

August 28, 2019

## 1 Introduction

Generic programming holds the promise of reducing the amount of code that needs to be written and maintained: rather than copying source code and adapting it for a new data type you simply let the compiler do that tedious job. Templates as featured in C++ are an *exemple par excellence* of generic programming. From the point of view of the programmer/user they are very easy to use<sup>1</sup>:

```
template <class T>
class mypair {
    T values [2];
public:
    mypair (T first , T second)
    {
        values[0]=first; values[1]=second;
    }
};
```

In this code fragment the capital T is a template parameter, representing a generic data type. To use this generic class you need to define an object with the template parameter filled in:

```
mypair<int> myobject (115, 36);
```

for a pair of integers or

```
mypair<double> myfloats (3.0, 2.18);
```

for a pair of double-precision reals. And of course you can also use other, non-intrinsic, data types.

The Standard Template Library (STL) in C++ is entirely based on such templates.

---

<sup>1</sup>Copied from <http://www.cplusplus.com/doc/oldtutorial/templates/>

While Fortran does not have such overt generic features, there is definitely more than meets the eye. For instance, we don't think about it as such, but array construction like:

```
x = [x, y]
```

works with any data type, both intrinsic and derived types, with no change to the code except for the declaration of the variables and literal values. The same holds for input or output, unless you need to control the formatting:

```
write(*,*) 'Value of x:', x
```

Two other generic features that Fortran supports are the *kind* mechanism and parametrised derived types. As remarked by Haverdaen et al. [2], many languages require a different keyword to select reals or integers of different range and precision. Fortran allows the programmer to arrange this via a *kind* parameter, thus reducing the need for this type of generic programming.

A recent paper by Haverdaen et al. [1] discusses the properties that any generic programming facility should ideally have. The experiments sketched in this note will be compared to these properties in section 5.

## 2 How to program a generic task?

### 2.1 Traversing a collection of data

When it comes to more extensive programming tasks, such as storing data in a flexible structure, a linked list, for instance, we need to do some work. This note examines a few possibilities to minimize that work. To make it more concrete, here is the problem we seek to solve:

Given a collection of data, print only those data that fulfill some user-defined criterium. We want the code to be as independent of the structure of the collection, the actual data and the criterium as possible.

For the collection<sup>2</sup> we require that there is a way to iterate over all its elements. For the sake of concreteness we consider ordinary arrays as well as *files on disk* that are read item by item. It should be possible to implement solutions for other types of collections as well, but with these two collection types we do not need much code to implement the desired behaviour.

If we restrict ourselves to arrays, then a simple solution could be:

```
do i = 1, size(array)
  if ( ... condition on array(i) ... ) then
    write(*,*) array(i)
  endif
enddo
```

or using a more advanced construct (showing that `associate` is more than just text substitution):

---

<sup>2</sup>In the STL the term is "container" and all manner of behaviour is defined for containers of various types. "Collection" seems a more neutral term.

```

associate( x => pack(array, ... condition on array ...) )
  do i = 1,size(x)
    write(*,*) x(i)
  enddo
end associate

```

These implementations have as the main drawback that they are only applicable to arrays. The type of solution we seek looks more like:

```

... define a suitable filter ...

do while ( collection%has_next() )
  write(*,*) collection%get()
enddo

```

## 2.2 Object-oriented solution

Using the object-oriented features introduced by the Fortran 2003 standard we can easily define derived types with associated methods. The collection should allow for some method to populate it – for arrays that will be quite different than for files, so leave that to the design of the specific collection – but both types should be able to store some kind of "filter" and should have a method `has_next()` and a method `get()`. This translates into a definition along these lines:

```

type collection
  type(data_type)      :: item
  logical               :: initialised = .false.
  logical               :: has_filter  = .false.
  logical               :: next_item   = .false.
  ... components for filtering ...
contains
  ! leave the create method to the implementation of
  ! specific collections
  !      procedure :: create    => create_generic
  !
  procedure :: has_next    => has_next_generic
  procedure :: get         => get_generic
  procedure :: set_filter  => set_filter_generic
end type generic_collection

```

This "skeleton" class anticipates that some generic data are required for the state of the collection – does it have data to work with? has a filter been set? etc.

For the filter we can use a separate derived type (class/object) or we can use a procedure pointer. There are good reasons to use a derived type, as you can store all manner of data in it for the parametrisation of the filter. However,

if you need a different filter in the program you would need to define a new type with an associated method to do the actual check that the item under examination is acceptable or not.

Therefore we use a procedure pointer instead: the procedure pointer can point to an internal routine and so use any data from the calling program unit. This is illustrated by the code fragment below:

```
program demo_filter
    ...
    !
    ! Set up the collection and the filter
    !
    call file%create( 'somedata.csv', skiplines = 1 )

    pattern = "NW1"
    call file%set_filter( contains_pattern )

    !
    ! Traverse the collection
    !
    do while ( file%has_next() )
        write(*,*) file%get()
    enddo

contains

    ! contains_pattern —
    !     Check if the string contains the pattern
    !
    logical function contains_pattern( string )
        type(data_type), intent(in) :: string

        contains_pattern = (index(string%value, trim(pattern)) > 0)
    end function contains_pattern
```

This program simply reads lines from a CSV file – without any interpretation – and prints those lines that contain the string "NW1". With a filter object that would require creating a class that holds the string and has a method with the above signature.

## 2.3 Making the code generic

During the preparation of this note various designs passed by, but I realised that the `get()` and `has_next()` methods are not enough if we want to make things "completely" generic:

- The method `has_next()` will examine the items in the file or the array one by one and return when an acceptable item is found. To make this

work for files and for arrays, we need a new method that is specific to the type of collection we are dealing with, but the *logic* of passing each item through a filter is generic:

*! Implementation of the has\_next() method*

```

logical function has_next_generic( this )
    class(collection), intent(inout) :: this

    type(data_type)                :: item
    logical                        :: success

    has_next_generic = .false.
    do
        call this%get_next( item , success )

        if ( .not. success ) then
            this%next_item = .false.
            exit
        endif
        if ( this%acceptable( item ) ) then
            has_next_generic = .true.
            this%next_item   = .true.
            this%item         = item
            exit
        endif
    enddo
end function has_next_generic

```

This routine is independent of the type of collection! It gets the next item using a specific method, `get_next` and stores the item once an acceptable one is encountered.

- The method `get()` needs to do nothing more than return the stored item. The condition of the loop ensures us that this method is only invoked when there is indeed such an item.

The method may be a function or a subroutine. A function is perhaps more direct (and that is the way it is shown above) but a subroutine is more natural if we want to return extra information, like: did we indeed succeed?<sup>3</sup>

The next issue to solve is that of the *type* of the data item that we store in the collection for retrieval. In the above code fragments we use the generic name `type(data_type)`. We cannot use a specific name or assume it is one of the intrinsic types: that would defeat the purpose of generic programming.

---

<sup>3</sup>There is no means to ensure at compile time that the routine `get()` is used in conjunction with `has_next()`. This might be a reason to check for run-time errors in `get()`.

We would need a copy of the source code for each type and kind we want to support. A straightforward solution is to use the *renaming facility* of the `use` statement and the `include` directive:

```

module collections
    use basic_types , data_type => string_type

    implicit none

    include 'collection_generic.f90'

    type, extends(collection) :: collection_file
        integer    :: lun
    contains
        procedure :: create    => create_file
        procedure :: get_next => get_next_file
    end type collection_file

contains

    include 'collection_implementation.f90'

    subroutine create_file( this , filename , skiplines )
        class(collection_file), intent(inout) :: this
        ...
    end subroutine create_file

    subroutine get_next_file( this , item , retrieved )
        class(collection_file), intent(inout) :: this
        type(data_type), intent(inout)          :: item
        logical , intent(out)                  :: retrieved

        character(len=100) :: line ! Arbitrary length
        integer            :: ierr

        read( this%lun , '(a)' , iostat = ierr ) line

        if ( ierr == 0 ) then
            retrieved = .true.
            item      = trim(line)
        else
            retrieved = .false.
        endif

    end subroutine get_next_file

```

```
end module collections
```

Here the type `string_type` is imported from a module providing various derived types and renamed to `data_type` so that the code in the module `collections` can be properly compiled. The two *include* files contain the declaration of the `collection` class and the implementation of the *completely generic* routines.

Via type extension we then create a new class, `collection_file` that reads lines from a file. The routine `get_next_file()` takes care of reading just one line and registering if this was successful or not.

In the actual implementation the type `string_type` is defined as:

```
type string_type
    character(len=:), allocatable :: value
end type string_type
```

to store character strings of arbitrary length. Unfortunately it is not possible to directly read a line from a file using an unallocated character string. Instead we use a string of fixed length and assign it to the component `value`.

A nice way to achieve this is to use a *defined assignment* (which is fairly trivial in this case):

```
interface assignment(=)
    module procedure assign_string
end interface

...
contains

subroutine assign_string( string , char )
    type(string_type), intent(inout) :: string
    character(len=*), intent(in)      :: char

    string%value = char
end subroutine assign_string
```

This mechanism can be usefully exploited by considering a data type that interprets the fields of the CSV file:

```
type station_data_type
    character(len=20) :: station
    character(len=10) :: date
    real              :: salinity
    real              :: temperature
end type station_data_type

interface assignment(=)
    module procedure assign_station_data
```

```

    end interface

    ...

contains

subroutine assign_station_data( station_data , char )
    type(station_data_type), intent(inout) :: station_data
    character(len=*), intent(in)           :: char

    read( char , * ) station_data%station , station_data%date , &
        station_data%salinity , station_data%temperature
end subroutine assign_station_data

```

The source code for the method `get_next` can remain exactly the same – via the renaming of the type `station_data_type` to `data_type` and the operator overloading that the compiler takes care of!

### 3 Limitation: intrinsic types

The methodology described here has one important drawback, namely the renaming feature is not applicable to intrinsic types. There is no way to rename, say, a plain integer to `data_type`, as we did with the derived types.

A workaround is to encapsulate intrinsic types into a derived type and provide defined assignments for these derived types to make it easier to work with:

```

type real_type
    real :: value
end type real_type

interface assignment(=)
    module procedure assign_sp_real
end interface

...

contains

elemental subroutine assign_sp_real( value , spvalue )
    type(real_type), intent(inout) :: value
    real , intent(in)              :: spvalue

    value%value = spvalue
end subroutine assign_sp_real

```

This is used among others in the main program for the array-based collections:

```

type(data_type), dimension(100) :: array_data
...
real(real_kind), dimension(100) :: random_value

!
! Set up the collection
!
call random_number( random_value )

array_data = random_value  ! Assign the values

call array%create( array_data )

```

## 4 Putting it all together

The various pieces can now be used to define dedicated collection types. For the two file-based collections, the only difference (except for the module that contains them) is the renaming of the actual underlying data type:

```

module m_collection_file
  use basic_types , only: data_type => string_type , &
    assignment(=)
  private
  public    :: collection_file , data_type

include 'collection_file_body.f90'

end module m_collection_file

module m_collection_station_data
  use basic_types , only: data_type => station_data_type , &
    assignment(=)
  private
  public    :: collection_file , data_type

include 'collection_file_body.f90'

end module m_collection_station_data

```

This same holds for the array-based collections – to make the using code independent of the chosen precision (single or double), we simply define the appropriate *kind* parameter:

```

module m_collection_real_array
  use basic_types , only: data_type => real_type , &
    assignment(=)

```

```

    private
    public    :: collection_array , data_type , real_kind , &
               assignment(=)
               ! Single precision
    integer , parameter :: real_kind = kind(1.0)

    include 'collection_array_body.f90'
end module m_collection_real_array

```

```

module m_collection_double_array
    use basic_types , only: data_type => double_type , &
        assignment(=)
    private
    public    :: collection_array , data_type , real_kind , &
               assignment(=)
               ! Double precision
    integer , parameter :: real_kind = kind(1.0d0)

    include 'collection_array_body.f90'
end module m_collection_double_array

```

The *include* file "collection\_file\_body.f90" looks like this, using in turn two other include files:

```

    include 'collection_generic.f90'

    type, extends(collection) :: collection_file
        integer                :: lun
    contains
        procedure              :: create    => create_file
        procedure              :: get_next  => get_next_file
    end type collection_file

contains

    include 'collection_implementation.f90'

    subroutine create_file( this , filename , skiplines )
        class(collection_file) , intent(inout) :: this
        character(len=*) , intent(in)          :: filename
        integer , intent(in) , optional        :: skiplines

        integer                                :: i

        open( newunit = this%lun , file = filename )

        if ( present(skiplines) ) then

```

```

        do i = 1,skiplines
            read( this%lun, * )
        enddo
    endif

    this%initialised = .true.
end subroutine create_file

subroutine get_next_file( this, item, retrieved )
    class( collection_file ), intent(inout) :: this
    type( data_type ), intent(inout) :: item
    logical, intent(out) :: retrieved

    character(len=100) :: line
    integer :: ierr

    read( this%lun, '(a)', iostat = ierr ) line

    if ( ierr == 0 ) then
        retrieved = .true.
        item = trim(line)
    else
        retrieved = .false.
    endif
end subroutine get_next_file

```

It defines:

- The abstract `collection` class via the two include files, "collection\_generic.f90" and "collection\_implementation.f90". That way the `data_type` derived type can ultimately be renamed.
- The generic class for file-based collections and the methods that belong to that. Note that the data type that is used to convey the data to the caller is `type(data_type)` and nothing else.

By putting the code for a particular collection type in a separate module, we achieve two things:

- *Reuse* of the renaming clauses – they only need to be put in the defining module and user code merely needs to `use` the module.
- If a program requires two or more collection types, then the renaming feature can be used to rename the collection types to avoid name clashes. This is also the reason for the `private` and `public` statements – only specific names are passed on.

The two main programs are shown side by side at the end of this note. As can be seen, the code is almost identical.

## 5 Comparison to C++ and the paper "Reflecting on generics"

The experiment described here relies on three features of Fortran:

- the `include` directive to avoid duplication of the source code;
- the renaming facility of the `use` statement, so that the same source code can be used for different data types;
- the overloading of assignment for specific data types, which enables the use of "type-agnostic" assignments.

The main limitation is seen with intrinsic types – they cannot be renamed and therefore an intermediate derived type is required. Other solutions such as unlimited polymorphic variables, are likely possible, but they have not been explored here.

In comparison to the template mechanism of C++ the construction of generic source code is slightly more complicated from the point of view of the user: you create a separate module for each specific collection type, although this comprises of no more than four or five lines of code. This needs to be done once and is independent from the using code. Therefore, there is always only one copy of the actual object code. In C++ this requires dedicated support from the compiler and linker, as discussed by Stone [3] and each instance of a templated class or function requires compilation of the same source code. (Another problem, noted by Haverdaen et al. [1], is that mistakes in the generic code may lead to convoluted and incomprehensible error messages. The Fortran construction presented here does not cause any more obscure error messages than usual Fortran code.)

The requirements that Haverdaen et al. [1] present regarding a generally useable generics mechanism are certainly not all met, though some have not been investigated here:

- Type safety is guaranteed within the bounds of Fortran, as the source code is actually straightforward Fortran.
- Intrinsic types are not treated in the same way as derived (user-defined) types, the major shortcoming.
- By *using* the modules that define specific collection types, it should be possible to extend the features, though this has not been demonstrated here. Modules like `m_collection_file` allow us to reuse the instantiation parameters and therefore long lists of types and kinds are not an issue.
- Renaming for instantiation and generic source code is at the heart of the generics mechanism presented here.

For setting up a repository of generic code an important step is not only the availability of a generics mechanism, but also an adequate distinction between

the generic and specific parts. Current-day Fortran, which does not exhibit its generic features, provides at least some of the necessary building blocks.

## Source code

The full source code is available at <https://sourceforge.net/p/flibs/svncode/HEAD/tree/trunk/experiments/generics/>

## References

- [1] Magne Haveraaen, Järvi Jaakko, and Damian Rouson. Reflecting on generics for Fortran, 2019.
- [2] Magne Haveraaen, Karla Morris, Damian Rouson, Hari Radhakrishnan, and Clayton Carson. High-performance design patterns for modern fortran. *Scientific Programming*, 2015:1–14, 2015.
- [3] Adrian Stone. Minimize code bloat: Template overspecialization, 2009.

```

program demo_filter
  use m_collection_file
  implicit none
  type(collection_file)          :: file

  type(data_type)                :: item

  character(len=20)              :: pattern
  logical                       :: retrieved

  ! Connect the file to the collection
  call file%create( 'somedata.csv', skiplines = 1 )

  ! Set up the filter
  pattern = "NW1"
  call file%set_filter( contains_pattern )

  ! Traverse the collection
  do while ( file%has_next() )
    call file%get( item, retrieved )
    write(*,*) item%value
  enddo

contains

logical function contains_pattern( string )
  type(data_type), intent(in) :: string

  contains_pattern = &
    ( index( string%value, trim(pattern) ) > 0 )
end function contains_pattern
end program demo_filter

```

```

program demo_filter
  use m_collection_real_array
  implicit none
  type(collection_array)          :: array
  type(data_type), dimension(100) :: array_data

  type(data_type)                :: item

  real(real_kind), dimension(100) :: random_value
  real(real_kind)                :: minimum
  logical                       :: retrieved

  ! Fill the collection with data (defined assignment)
  call random_number( random_value )
  array_data = random_value
  call array%create( array_data )

  ! Set up the filter
  minimum = 0.6_real_kind
  call array%set_filter( is_greater )

  ! Traverse the collection
  do while ( array%has_next() )
    call array%get( item, retrieved )
    write(*,*) item%value
  enddo

contains

logical function is_greater( value )
  type(data_type), intent(in) :: value

  is_greater = value%value > minimum

end function is_greater
end program demo_filter

```