Prototypes and properties

Arjen Markus
arjen.markus@deltares.nl

May 30, 2019

1 Introduction

It may not be obvious at a first glance, but object-oriented programming comes in a wide variety of flavours. While dominated perhaps by the C++ and Java flavours – which differ in subtle ways – there are also such methodologies as advocated by the *Self* language.[1] Instead of *classes* that have a fixed set of components and methods and that have to be extended by defining subclasses, the *Self* language uses so-called protoypes: an object inherits the components and methods from any object you like and you can expand that set per object. This makes the objects in question very flexible.

I was inspired for the code described in this note by both the *Self* language and by a blog by Steve Yegge[2]. The implementation has a large number of limitations, as it is merely a proof of concept, but what it does is quite in the spirit of this *prototypes* pattern. What it does in short:

- The `prototype` derived type allows you to define properties that can be retrieved by name.
- The properties can have any type you need. If new (derived) types need to be supported, you define two additional subroutines and link them to the existing set via an interface statement. There is no need to define an extended derived type (a subclass).
- The list of properties stored in a `prototype` variable can simply be copied to another variable of this derived type.
- One limitation with respect to the *prototypes* pattern as described by Yegge is that properties are not inherited from a parent, but are simply copied. The reason for this limitation is that you would have to be very careful about keeping parent objects alive, as otherwise you can easily create "dangling pointers".

[1] See for instance `https://en.wikipedia.org/wiki/Self_(programming_language)`.
[2] See `https://steve-yegge.blogspot.com/2008/10/universal-design-pattern.html`.

2 The interface and implementation

The module `prototypes` defines all the functionality that is needed to store and retrieve properties:

- The relevant derived type is `prototype`. It is not defined as a "class" in the Fortran sense, that is, it has no type-bound procedures. The reason for that is explained in section 3.
- There are two public routines: `prototype_set` and `prototype_get`. These form the entire set of explicitly defined methods for the derived type. (For debugging purposes there is also a routine `prototype_print` that prints the list of properties.)
- Due to the way the derived type has been defined, namely with *allocatable components*, to copy the list of properties, you can use default assignment:

```
use prototypes

type(prototype) :: p1, p2
integer         :: start, end
logical         :: found

!
! Fill properties for variable p1
!
call prototype_set( p1, "Start", 1 )
call prototype_set( p1, "End", 10 )
call prototype_set( p1, "Stepsize", 3 )

!
! We need a copy, reset one property:
!

p2 = p1         ! This relies on automatic reallocation!

call property_set( p2, "Start", 2 )

call property_get( p2, "Start", start, found )
call property_get( p2, "End", end, found )

write(*,*) "Start and end for object p2: ", start, end

call property_get( p1, "Start", start, found )
call property_get( p1, "End", end, found )
write(*,*) "Start and end for object p1: ", start, end
```

The `prototype` derived type holds an array of type `property_type`:

```
type property_type
    character(len=:), allocatable :: name
    class(*), allocatable         :: value
end type property_type
```

The essential aspect of this derived type is the use of an unlimited polymorphic component, `value`. Together with a few other advanced features, this makes it possible to implement the specific versions of `prototype_set` in the following fashion:

```
subroutine prototype_set_int( p, name, value )
    type(prototype), intent(inout) :: p
    character(len=*), intent(in)   :: name
    integer, intent(in)            :: value

    integer :: indx

    call find_index( p, name, indx )

    if ( allocated(p%list(indx)%value) ) then
        deallocate( p%list(indx)%value )
    endif
    allocate( p%list(indx)%value, source = value )
end subroutine prototype_set_int
```

(The implementation of the private routine `find_index` is not important – it may simply search the list of names and insert a new name if it has not already been defined or it may use a hash table to speed up searches.)

All that needs to be changed for other types of data to be stored is the declaration of the `value` argument.

As only the `allocatable` attribute is used for variables and components that have a dynamic size, memory management is particularly simple: we can rely on the rules defined for such variables.

The last important feature to be noted is the use of "sourced" allocation: this causes the polymorphic variable to take on the right (dynamic) type as well as the right value.

The retrieval routines, `prototype_get`, are slightly more complicated: the polymorphic variable has to be "cast" to the right type and it may be that no key by the requested name has been defined yet:

```
subroutine prototype_get_int( p, name, value, found )
    type(prototype), intent(in)  :: p
    character(len=*), intent(in) :: name
    integer, intent(out)         :: value
    logical, intent(out)         :: found

    integer :: indx
```

```
    call find_existing_index( p, name, indx )

    found = .false.
    value = -999
    if ( indx > -1 ) then
        select type( v => p%list(indx)%value )
            type is (integer)
                found = .true.
                value = v
            class default
                write(*,*) 'Value for "', name, '" not an integer'
        end select
    else
        write(*,*) 'Property "', name, '" does not exist'
    endif
end subroutine prototype_get_int
```

(As the name suggests, the private routine `find_existing_index` does not add a new entry, if no key by the requested name was found.)

How to deal with non-existing keys or keys that have an associated value that is not compatible with the requested data type, is a vexing question, for which there exists no universally acceptable answer, although in programming languages like C++ and Java, you would probably use *exceptions* to transfer control to a calling routine.

The various specific routines for storing and retrieving properties are bundled into two interfaces:

```
interface prototype_set
    module procedure prototype_set_int
    module procedure prototype_set_char
    ...
end interface

interface prototype_get
    module procedure prototype_get_int
    module procedure prototype_get_char
    ...
end interface
```

3 Supporting new basic and derived types

As indicated in the introduction, it is fairly easy to extend the current `prototypes` module to include new basic or derived types. All you need to do is:

- Define specific versions of the `prototype_set` and `prototype_get` routines.

- Extend the generic interfaces.

This can in fact be done *without* modifying the original module:

```
module prototypes_additional
    use prototypes

    type iterate
        integer :: start    = 1
        integer :: end      = 0
        integer :: stepsize = 1
    end type iterate

    interface prototype_set
        module procedure prototype_set_type_iterate
    end interface

    interface prototype_get
        module procedure prototype_get_type_iterate
    end interface
contains
subroutine prototype_set_type_iterate
    ...
end subroutine prototype_set_type_iterate

subroutine prototype_get_type_iterate
    ...
end subroutine prototype_get_type_iterate
end module
```

This is possible, because generic interfaces with the same name are merged.

If the routines were made *methods* of a class `prototype`, then generic interfaces are also possible, but they cannot be extended as easily: instead of via a different module, we would need to *extend* the `prototype` class:

```
type prototype
    type(property_type), allocatable, dimension(:) :: list
    ...
contains
    procedure :: get_int  => prototype_get_int
    procedure :: get_char => prototype_get_char
    ...
    generic :: get => get_int, get_char, ...
end type prototype

type(prototype) :: prototype_extended
contains
```

```
    procedure :: get_iterate => prototype_get_iterate
    ...
    generic :: get => get_iterate
end type prototype_extended
```

Of course, such an implementation enables the programmer to use code like:

```
call p%get( "Start", start, found )
```

One might also consider using a *function* form instead of a *subroutine* for retrieving property values. The problem with functions, however, is that the return type is not used to identify the specific implementation in a particular context:

```
integer :: start
real    :: initial_value

! For simplicity: ignore the "not found" case

start         = p%get( "Start" )
initial_value = p%get( "Initial" )  ! Ambiguity!
```

A workaround would be to let the `prototype_get` function return some special derived type and define specific assignments to convert from that specific type to whatever the type of the left-hand side variable. Whether this is an attractive alternative is partly a matter of taste.

4 Storing functions and subroutines as properties

The presentation sofar has focussed on storing data, both basic data like integers and strings and derived types. But the method can be extended to store and retrieve pointers to *functions* and *subroutines* as well. Just as for data, you can define a derived type to hold the name and the actual pointer:[3]

```
type proc_property_type
    character(len=:), allocatable :: name
    procedure(), pointer, nopass  :: proc
end type proc_property_type
```

The subsequent use is (almost) the same as for data:

```
!
! "Cast" the subroutine name to a procedure pointer
! The subroutine "printing" takes two integer arguments
!
```

[3]It does not seem possible to use sourced allocation with a procedure pointer as source.

```
proc => printing
call prototype_set( p, "Print", proc )

!
! Retrieve the pointer and call the routine
!
proc => null()
call prototype_get( p, "Print", proc, found )

if ( found ) then
    call proc( 10, 11 )
else
    write(*,*) 'Unknown method - "Print"'
endif
```

One caveat though: there is no check whether the argument list to the actual call is correct in terms of number and type of arguments.

While here we have used procedure pointers independent of an object, you can also use this technique to control the *actual* implementation of object methods. After all, the components of an object can be procedure pointers too.

5 Conclusion

Using unlimited polymorphic variables and sourced allocation makes it possible to implement an almost generic type of storage and retrieval mechanisms. The module described here is merely an example of a much wider class of such mechanisms.

6 Beyond a demonstration

This note illustrates how you could implement the *prototypes* pattern in Fortran. The implementation is far from complete. To make it more useful:

- Add more basic types to the code.
- Implement support for procedure pointers.

The source code can be found at `https://sourceforge.net/p/flibs/svncode/HEAD/tree/trunk/experiments/prototypes.f90`