Object-Oriented Programming in Fortran 2003 Part 2: Data Polymorphism

Original article by Mark Leair, PGI Compiler Engineer

> Note: This article was revised in March 2015 and again in January 2016 to bring it up-to-date with the production software release and to correct errors in the examples.

This is Part 2 of a series of articles: * Part 1: Code Reusability * Part 2: Data Polymorphism

1. Introduction

This is the second part to a series of articles that explore Object-Oriented Programming (OOP) in Fortran 2003 (F2003). The first installment introduced the OOP paradigm and three important features to OOP: inheritance, polymorphism, and information hiding. F2003 supports inheritance through type extension, polymorphism through its CLASS keyword, and information hiding through its **PUBLIC/PRIVATE** keywords/binding-attributes.

There are two basic types of polymorphism: procedure polymorphism and data polymorphism. Part one of the series covered procedure polymorphism, which deals with procedures that can operate on a variety of data types and values. Data polymorphism, a topic for this article, deals with program variables that can store and operate on a variety of data types and values.

In addition to data polymorphism, we will also examine F2003's typed allocation, sourced allocation, unlimited polymorphic objects, generic type-bound procedures, abstract types, and deferred bindings.

2. Data Polymorphism in F2003

The **CLASS** keyword allows F2003 programmers to create polymorphic variables. A polymorphic variable is a variable whose data type is dynamic at runtime. It must be a pointer variable, allocatable variable, or a dummy argument. Below are some examples of polymorphic variables:

```
subroutine init(sh)
  class(shape) :: sh              ! polymorphic dummy argument
  class(shape), pointer :: p      ! polymorphic pointer variable
  class(shape), allocatable:: als ! polymorphic allocatable variable
end subroutine init
```

In the example above, the `sh`, `p`, and `als` polymorphic variables can each hold values of type shape or any type extension of shape. The `sh` dummy argument receives its type and value from the actual argument to `sh` of subroutine `init()`. Just as how polymorphic dummy arguments form the basis to procedure polymorphism, polymorphic pointer and allocatable variables form the basis to data polymorphism.

The polymorphic pointer variable `p` above can point to an object of type `shape` or any of its extensions. For example,

```
subroutine init(sh)
  class(shape),target :: sh
  class(shape), pointer :: p

  select type (sh)
  type is (shape)
    p => sh
    :  ! shape specific code here
  type is (rectangle)
    p => sh
    :  ! rectangle specific code here
  type is (square)
    p => sh
    :  ! square specific code here
  class default
    p => null()
  end select
:
end subroutine init
```

We used the **select type** construct above to illustrate the fact that the polymorphic pointer, `p`, can take on several types. In this case, `p` can point to a `shape`, `rectangle`, or `square` object. The dynamic type of pointer `p` is not known until the pointer assignment (i.e., `p => sh`) is executed.

An allocatable polymorphic variable receives its type and optionally its value at the point of its allocation. For example,

```
class(shape), allocatable :: als
allocate(als)
```

In the example above, we allocate polymorphic variable `als`. By default, the dynamic type of a polymorphic allocatable variable is the same as its declared type after executing an allocate statement. In the example above, variable `als` receives dynamic type `shape` after we execute the allocate statement.

Obviously there is not much use for polymorphic allocatable variables if we can only specify the declared type in an **allocate** statement. Therefore, F2003 provides typed allocation to allow the programmer to specify a type other than the declared type in an **allocate** statement. Below is an example:

```
class(shape), allocatable :: als
allocate(rectangle::als)
```

In the allocate statement above, we specify a type followed by a `::` and the variable name. In this case, we specify `rectangle` as the dynamic type of variable `als`.

> Note that the declared type of `als` is still `shape`. Also the type specification must be the same or a type extension of the declared type of the allocatable variable.

Below is another example that illustrates how we may allocate a polymorphic variable with the same type of another object:

```
subroutine init(sh)
  class(shape) :: sh
  class(shape), allocatable :: als

  select type (sh)
  type is (shape)
    allocate(shape::als)
  type is (rectangle)
    allocate(rectangle::als)
  type is (square)
    allocate(square::als)
  end select
:
end subroutine init
```

Now, let's illustrate how we may expand our example above to create a "copy" of an object.

```
subroutine init(sh)
  class(shape) :: sh
  class(shape), allocatable :: als

  select type (sh)
  type is (shape)
    allocate(shape::als)
    select type(a)
    type is (shape)
      als = sh   ! copy sh
    end select
  type is (rectangle)
    allocate(rectangle::als)
     select type (als)
     type is (rectangle)
       als = sh ! copy sh
     end select
  type is (square)
    allocate(square::als)
    select type (als)
    type is (square)
      als = sh   ! copy sh
    end select
  end select
:
end subroutine init
```

Recall that the programmer can only access the components of the declared type by default. Therefore, in the example above, we can only access the `shape` components for object `als` by default. In order to access the components of the dynamic type of object `als`, a nested select type is used for object `als`.

The previous example presents an interesting application of data polymorphism; making a copy or a clone of an object. Unfortunately, the previous example does not scale well if shape has several type extensions. Also whenever we add a type extension to shape, we need to `update` our `init()` subroutine to include the new type extension. To address this problem, F2003 provides sourced allocation, as illustrated below:

```
subroutine init(sh)
  class(shape) :: sh
  class(shape), allocatable :: als
  allocate(als, source=sh) ! als becomes a clone of sh
  :
end subroutine
```

The optional `source=` argument to allocate specifies sourced allocation. In our example above, the **allocate** statement will allocate `als` with the same dynamic type as `sh` and with the same value(s) of `sh`.

> Note that the declared type of the `source=` must be the same or a type extension of the allocate argument (e.g., `als`).

3. Unlimited Polymorphic Objects

Our discussion on data polymorphism so far has been limited to derived types and their type extensions. While this satisfies most applications, sometimes we may want to write a procedure or a data structure that can operate on any type including any intrinsic or derived type. F2003 provides unlimited polymorphic objects for just that purpose. Below are some examples of unlimited polymorphic objects:

```
subroutine init(sh)
  class(*) :: sh              ! unlimited polymorphic dummy argument
  class(*), pointer :: p      ! unlimited polymorphic pointer variable
  class(*), allocatable:: als ! unlimited polymorphic allocatable variable
end subroutine init
```

The `class(*)` keyword is used to specify an unlimited polymorphic object declaration. The operations for unlimited polymorphic objects are similar to those mentioned for "limited" polymorphic objects in the previous section. However, unlike "limited" polymorphic objects, their "unlimited" counterparts can take any F2003 type. Below is an example of unlimited polymorphic objects used with procedure polymorphism:

```
subroutine init(sh)
  class(*) :: sh

  select type(sh)
  class is (shape)
    :   ! shape specific code
  type is (integer)
    : ! integer specific code
  type is (real)
    : ! real specific code
  type is (complex)
    : ! complex specific code
  end select
end subroutine init
```

Similarly, any pointer or target can be assigned to an unlimited polymorphic pointer, regardless of type. For example,

```
subroutine init(sh)
  class(*),target :: sh
  class(*), pointer :: p

  p => sh

  select type(p)
  class is (shape)
    :   ! shape specific code
  type is (integer)
    : ! integer specific code
  type is (real)
    : ! real specific code
  type is (complex)
    : ! complex specific code
  end select

end subroutine init
```

The example above shows `sh` assigned to pointer `p` and then a **select type** construct is used to query the dynamic type of pointer `p`.

Unlimited polymorphic objects can also be used with typed allocation. In fact, a type (or `source=` argument) must be specified with the **allocate** statement since there is no default type for `class(*)`. However, unlike their "limited" counterparts, any F2003 type, intrinsic or derived, can be specified. For example,

```
subroutine init(sh)
  class(*) :: sh
  class(*), allocatable :: als

  select type(sh)
  type is (shape)
    allocate(shape::als)
  type is (integer)
    allocate(integer::als)
  type is (real)
    allocate(real::als)
  type is (complex)
    allocate(complex::als)
  end select
:
end subroutine init
```

Sourced allocation can also operate on unlimited polymorphic objects:

```
subroutine init(sh)
  class(*) :: sh
  class(*), allocatable :: als
  allocate(als, source=sh) ! als becomes a clone of sh
  :
end subroutine  init
```

Note if the `source=` argument is an unlimited polymorphic object (i.e., declared `class(*)`), the allocate argument (e.g., `als`) must also be an unlimited polymorphic object. When the allocate argument is declared `class(*)`, the declared type in the `source=` argument can be any type including `class(*)`, any derived type, or any intrinsic type. For example,

```
class(*), allocatable :: als
integer i
i = 1234
source(als, source=i)
```

The above code demonstrates sourced allocation with an unlimited polymorphic allocatable argument and an intrinsic typed `source=` argument.

4. Case Study: Data Polymorphic Linked List

As mentioned in the previous section, one of the advantages to unlimited polymorphic objects is that we can create data structures that can operate on all data types (intrinsic and derived) in F2003. To demonstrate, we will create data structures that can be used to create a heterogeneous list of objects. Traditionally, data stored in a linked list all have the same data type. However, with unlimited polymorphic objects, we can easily create a list that contains a variety of data types and values.

We start out by creating a derived type that will represent each link in our linked list. For example,

```
type link
   class(*), pointer :: value => null()
   type(link), pointer :: next => null()
end type link
```

The basic link derived type above contains an unlimited polymorphic pointer that points to the value of the link followed by a pointer to the next link in the list. Recall that information hiding allows others to use an object without understanding its implementation details. Therefore, we will place this derived type into its own module, add a constructor, and add some type-bound procedures to access the value(s). For example,

```
module link_mod
  private    ! information hiding
  public :: link

  type link
     private   ! information hiding
     class(*), pointer :: value => null()
     type(link), pointer :: next => null()
     contains
     procedure :: getValue    ! get value in this link
     procedure :: nextLink    ! get the link after this link
     procedure :: setNextLink ! set the link after this link
  end type link

  interface link
    procedure constructor
  end interface

contains

  function nextLink(this)
    class(link) :: this
    class(link), pointer :: nextLink
    nextLink => this%next
  end function nextLink

  subroutine setNextLink(this,next)
    class(link) :: this
    class(link), pointer :: next
    this%next => next
  end subroutine setNextLink

  function getValue(this)
    class(link) :: this
    class(*), pointer :: getValue
    getValue => this%value
  end function getValue

  function constructor(value, next)
    class(link),pointer :: constructor
    class(*) :: value
    class(link), pointer :: next
    allocate(constructor)
    constructor%next => next
    allocate(constructor%value, source=value)
  end function constructor

end module link_mod
```

Because we added the private keywords above, the user of the object must use the `getValue()` function to access the values of each link in our list, the `nextLink()` procedure to access the next link in the list, and `setNextLink()` to add a link after a link. Note that the `getValue()` function returns a pointer to a `class(*)`, meaning it can return an object of any type.

We employ type overloading for the constructor function. Recall from part one that type overloading allows us to create a generic interface with the same name as a derived type. This allows us to create a constructor function and hide it behind the name of the type. Using our example above, we can construct a link in the following manner:

```
class(link),pointer :: linkList
integer v
linkList => link(v, linkList%next)
```

Although one could easily create a linked list with just the link object above, the real power of OOP lies in its ability to create flexible and reusable components. Unfortunately, the user is still required to understand how the list is constructed with the link object (e.g., the link constructor assigns its result to the `linkList` pointer above). Therefore, we can create another object called list that acts as the "Application Program Interface" or API to the link object. Below is an example of a list object that acts as an API to our linked list data structure:.

```
type list
  class(link),pointer :: firstLink => null() ! first link in list
  class(link),pointer :: lastLink => null()  ! last link in list
contains
  procedure :: addInteger ! add integer to list
  procedure :: addChar    ! add character to list
  procedure :: addReal    ! add real to list
  procedure :: addValue   ! add class(*) to list
  generic :: add => addInteger, addChar, addReal
end type list
 ```
The list derived type has two data components, ``firstlink``, which points to the
first link in its list and ``lastLink`` which points to the last link in the list.
The ``lastLink`` pointer allows us to easily add values to the end of the list.
Next, we have three type-bound procedures called ``addInteger()``, ``addChar()``,
and ``addReal()``. As one may guess, the first three procedures are used to add an
``integer``, a ``character``, and a ``real`` to the linked list respectively. The
``addValue()`` procedure adds ``class(*)`` values to the list and is the main add
routine. The ``addInteger()``, ``addChar()``, and ``addReal()`` procedures are
```
```

~~~
actually just wrappers to the ``addValue()`` procedure. Below is the
``addInteger()`` procedure. The only difference between ``addInteger()``,
``addChar()``, and ``addReal()`` is the data type dummy argument, value.

```fortran
subroutine addInteger(this, value)
  class(list) :: this
  integer value
  class(*), allocatable :: v

  allocate(v,source=value)
  call this%addValue(v)
end subroutine addInteger
~~~

The `addInteger()` procedure takes an `integer` value and allocates a `class(*)` with that value using sourced allocation. The value is then passed to the `addValue()` procedure shown below:

```
subroutine addValue(this, value)
  class(list) :: this
  class(*), value
  class(link), pointer :: newLink

  if (.not. associated(this%firstLink)) then
     this%firstLink => link(value, this%firstLink)
     this%lastLink => this%firstLink
  else
     newLink => link(value, this%lastLink%nextLink())
     call this%lastLink%setNextLink(newLink)
     this%lastLink => newLink
  end if

end subroutine addValue
```

The `addValue()` procedure takes two arguments; a `list` and a `class(*)`. If the `list`'s`firstLink` is not associated (i.e., points to `null()`), then we will add our value to the start of the list by assigning it to the `list`'s`firstLink` pointer. Otherwise, we add it after the `list`'s`lastLink` pointer.

Returning to the list type definition above, you will see the following statement:

```
generic :: add => addInteger, addChar, addReal
```

This statement uses an F2003 feature known as a generic-type bound procedure. Generic-type bound procedures act very much like generic interfaces, except they are specified in the derived-type and only type-bound procedures are permitted in the generic-set.

In our example above, we can invoke the add type-bound procedure and either the `addInteger()`, `addChar()`, or `addReal()` implementations will get called. The compiler will determine which procedure to invoke based on the data type of the actual arguments. If we pass an integer to `add()` 's value argument, `addInteger()` will get invoked, a character value will invoke `addChar()`, and a real value will invoke `addReal()`.

Below is a simple program that adds values to a list and prints out the values. You can download the complete list_mod and link_mod modules, which encapsulate the list and link objects respectively.

```
program main
  use list_mod
  implicit none
  integer i
  type(list) :: my_list

  do i=1, 10
     call my_list%add(i)
  enddo

  call my_list%add(1.23)
  call my_list%add('A')
  call my_list%add('B')
  call my_list%add('C')
  call my_list%printvalues()
end program main
```

~~~
% pgfortran -c list.f90
% pgfortran -c link.f90
% pgfortran -V main.f90 list.o link.o
pgfortran 15.1-0 64-bit target on x86-64 Linux -tp sandybridge
The Portland Group - PGI Compilers and Tools
Copyright 2015, NVIDIA CORPORATION.  All Rights Reserved.
% a.out
            1
            2
            3
            4
            5
            6
            7
            8
            9
           10
    1.230000
 A
 B
 C
 ```
## 5. Abstract Types and Deferred Bindings
In our case study, we had a ``list`` derived type that acted as the API for our
linked list. Rather than employ one implementation for the ``list`` derived type,
we could choose to define some of the components and type-bound procedures for a
``list`` object and require the user to define the rest. This can be accomplished
through an **abstract** type. An **abstract** type is a derived type that cannot be
instantiated. Instead, it is extended and further defined by another type. The type
extension too can be declared abstract, but ultimately it must be extended by a
non-abstract type if it ever is to be instantiated in a program. Below is an
~~~

~~~
example of list type declared abstract:

```fortran
module abstract_list_mod
:
type, abstract :: list
   private
   class(link),pointer :: firstLink => null() ! first link in list
   class(link),pointer :: lastLink => null()  ! last link in list
   class(link),pointer :: currLink => null()  ! list iterator
 contains
   procedure, non_overridable :: addValue    ! add value to list
   procedure, non_overridable :: firstValue  ! get first value in list
   procedure, non_overridable :: reset       ! reset list iterator
   procedure, non_overridable :: next        !iterate to next value in list
   procedure, non_overridable :: currentValue! get current value in list
   procedure, non_overridable :: moreValues  ! more values to iterate?
   generic :: add => addValue
   procedure(printValues), deferred :: printList  ! print contents of list
end type list

abstract interface
  subroutine printValues(this)
    import list
    class(list) :: this
  end subroutine
end interface
:
end module abstract_list_mod
 ```

The abstract list type above uses the link type discussed in the previous section
as its underlying data structure. We have three data components, firstLink,
lastLink, and currLink. The firstLink and lastLink components point to the first
and last link in the list respectively. The currLink component points to the
"current" link that we are processing in the list. In other words, currLink acts as
a list iterator that allows us to traverse the list using inquiry functions. If we
did not provide a list iterator, the user of our list type would need to understand
the underlying link data structure. Instead, we take advantage of information
hiding by providing a list iterator.

Our list type is declared abstract. Therefore, the following declaration and
allocate statements are invalid for list:

```fortran
type(list) :: my_list    ! invalid because list is abstract
allocate(list::x)        ! invalid because list is abstract
~~~

On the other hand, the abstract type can be used in a class declaration since its dynamic type can be a non-abstract type extension. For example,

```

subroutine list_stuff(my_list)
  class(list) :: my_list
  class(list), pointer :: p
  class(list), allocatable :: a
```

```
  select type (my_list)
  class is (list)
  :
  end select
end subroutine
```

The above usage of list is valid because we are not declaring or allocating anything with type list. Instead, we are asserting that each variable is some type extension of list.

In our list type definition above, we added the **deferred** binding to the `printValues` type-bound procedure. The **deferred** binding can be added to type-bound procedures that are not defined in the abstract type but must be defined in all of its non-abstract type extensions. Deferred bindings allow the author of the abstract type to dictate what procedures must be implemented by the user of the abstract type and what may or may not be overridden. F2003 also requires that a deferred binding have an interface (or an abstract interface) associated with it. The following is the syntax for deferred bindings:

```
procedure (interface-name), deferred :: procedure-name
```

Because deferred bindings have an interface associated with them, there is no `=>` followed by an implementation-name allowed in the syntax (e.g., `procedure, deferred :: foo => bar` is not allowed).

Below is an example of a type that extends our list type above:

```
module integer_list_mod
:
type, extends(list) :: integerList
contains
   procedure :: addInteger
   procedure :: printList => printIntegerList
   generic :: add => addInteger
end type integerList
:
end module integer_list_mod
```

The `integerList` extends our abstract type, list. Note that `printList()` is defined as required by the deferred binding in list. Below is the implementation for the `printList()` type-bound procedure:

```
subroutine printIntegerList(this)
  class(integerList) :: this
  class(*), pointer :: curr

  call this%reset()              ! reset list iterator

  do while(this%moreValues())    ! loop while there are values to print
```

```
    curr => this%currentValue() ! get current value
    select type(curr)
    type is (integer)
      print *, curr            ! print the integer
    end select
    call this%nextValue()       ! increment the list iterator
  end do

  call this%reset()              ! reset list iterator

end subroutine printIntegerList
```

In `printIntegerList()`, we print out the integers in our list. We first call the list `reset()` procedure to make sure that the list iterator is at the beginning of the list. Next, we loop through the list, calling the list's `moreValues()` function to determine if our list iterator has reached the end of the list. We call the list's `currentValue()` function to get the value that the list iterator is pointing to. We then use a **select type** to access the integer value and print it. Finally, we call the list's `nextValue()` procedure to increment the list iterator so we can access the next value.

Below is a sample program that uses our abstract list and integerList types. The sample program adds the integers one through ten to our list and then calls the `integerList`'s `printList()` procedure. Next, the program traverses the list, places the integers into an array, and then prints out the array. You can download the complete abstract_list_mod and integer_list_mod modules from the PGI website.

```
program main
  use integer_list_mod
  implicit none
  integer i
  type(integerList) :: my_list
  integer values(10)

  do i=1, 10
     call my_list%add(i)
  enddo

  call my_list%printList()
  print *

  call my_list%reset()
  i = 1
  do while(my_list%moreValues())
     values(i) = my_list%current()
     call my_list%next()
     i = i + 1
  end do

  print *, values
end program main
```

Below is a sample compile and run of the above program:

```
% pgfortran -c link.f90
% pgfortran -c abstract_list.f90
% pgfortran -c integerList.f90
% pgfortran -V main.f90 link.o abstract_list.o integerList.o

pgfortran 11.6-0 64-bit target on x86-64 Linux -tp penryn
Copyright 1989-2000, The Portland Group, Inc.  All Rights Reserved.
Copyright 2000-2011, STMicroelectronics, Inc.  All Rights Reserved.
% a.out
           1
           2
           3
           4
           5
           6
           7
           8
           9
          10

           1            2            3            4            5
6          7            8            9           10
```

6. Conclusion

Polymorphism permits code reusability in the Object-Oriented Programming paradigm because the programmer can write procedures and data structures that can operate on a variety of data types and values. The programmer does not have to reinvent the wheel for every data type a procedure or a data structure will encounter.

There are two types of polymorphism: procedure polymorphism and data polymorphism. Procedure polymorphism occurs when a procedure can operate on a variety of data types and values. Data polymorphism occurs when we have a pointer or allocatable variable that can operate on a variety of data types and values.

The dynamic type of these variables change when we assign a target to a polymorphic pointer variable or when we use typed or sourced allocation with a polymorphic allocatable variable. Next, we looked at unlimited polymorphic objects and presented a case study where these objects can be used to create heterogeneous data structures (e.g., a list object that links together a variety of data types). Finally, we looked at how abstract types can be used to dictate requirements for type extensions and how they interact with polymorphic variables.

7. Acknowledgement

The author would like to thank Paul Richard Thomas and Philip Mason for submitting corrections to the code examples in earlier versions of this article.