Object–Oriented Features in Fortran 2003

Reinhold Bader

Leibniz Supercomputing Centre
Munich
December 2007
Characterization of object-oriented (OO) programming (1)

Following the Wikipedia entry
the following properties are relevant:

1. **Class**: Unit of definition of data and behaviour (=methods) of some-kind-of-thing; the basis of modularity and structure in an object oriented program.

**Fortran 95 support:**
- type definitions
- contained subroutines
- within a module (re-use)

**Fortran 2003 support:**
- by compatibility
- **class** keyword
  - refers to inheritance/polymorphism
  - improves abstraction
2. **Object**: An instance of a class, an object is a run-time manifestation of an exemplar of a class. Each object has its own (separate) data, which characterize the state of the object.

**Fortran 95 support:**
- `type(...) :: object` declaration
  - default initialization
- dynamically via `pointer` attribute
- array support
  - dynamic arrays via `allocatable` attribute
  - intrinsics for array manipulation
- `structure constructor`

**Fortran 2003 support:**
- by compatibility
- `allocate` is much more powerful
3. **Encapsulation**: protection of the internal structure of objects against manipulation, unless via the objects' exposed interface.

**Fortran 95 support:**
- module concept
  - module (not class) is unit of encapsulation
- impose access limits: *public* and *private* attributes
  - type definitions, type components
  - global variables and contained subroutines

**Fortran 2003 support:**
- by compatibility
- more fine-grained
  - required because of type extensibility
- **new attribute** *protected*
  - global objects only
4. **Message passing (Interfacing):** the process by which an object sends data to another object or asks the other object to invoke a method.

**Fortran 95 support:**
- not on type/object level
- indirectly via explicit and named (“generic”) interfaces
  - check object TKR

**Fortran 2003 support:**
- by compatibility
- type-bound procedures
  - bind method to a type definition
- procedure pointer components
  - bind subroutine call to an object
- abstract interfaces
Characterization of object-oriented programming (5)

5. **Inheritance**: mechanism for creating sub-classes, by specialization (subtyping, extending) another class. All data and functions of the superclass(es) are acquired, but data/methods may be added/changed. “Is-a” relationship, as opposed to “has-a” relationship induced by composition.

**Fortran 95 support:**
- no
  - emulate by combining composition, delegation and generic interfaces

**Fortran 2003 support:**
- type extension
  - inherit type components (including procedure pointer components)
  - inherit type bound procedures, or override them
- multiple inheritance is unsupported
6. **Abstraction**: Ability of a program to ignore the details of an objects' (sub)class and work at a more generic level when appropriate.

**Fortran 95 support:**
- derived data types, type composition
- operator overloading, self-defined operators
- generic interfaces

**Fortran 2003 support:**
- by compatibility
- generic type-bound procedures
7. **Polymorphism**: Behaviour of methods that varies depending on the class membership of objects worked upon.

- **static** polymorphism: all method calls are fixed at compilation time.
- **dynamic** polymorphism: delay method calling determination to runtime.
- **parametric** polymorphism: parameterize functions and data structures over arbitrary values.

**Fortran 95 support:**

- **static polymorphism via generic interfaces**
  - (limited) emulation of dynamic polymorphism

**Fortran 2003 support:**

- **dynamic polymorphism**
  - dynamic objects
  - subroutine interface
- **parametric polymorphism**
  - very limited: kind and length parameters allowed
Problems with terminology
- terms and their meaning vary between languages
  - danger of misunderstandings
- will use Fortran-specific jargon
  - but will also compare with C++ from time to time

Aims of OO paradigm:
- improvements in
  - re-usability
  - abstraction
  - moving from procedural to data-centric programming
  - reducing software development effort, improving productivity
- indiscriminate usage of OO however may be (very) counterproductive
  - identify abstract "software patterns" which have proven useful
Scope of OO within Fortran

- Fortran 95 supports **object-based** programming
- Fortran 2003 supports **object-oriented** programming

Specific intentions of Fortran object model:
- backward compatibility with Fortran 95
  - broken essentially only with respect to semantic change in treatment of **allocatable** variables
- allow extensive correctness and consistency checking by the compiler
- module remains the unit of encapsulation
- design more reminiscent of Ada than C++
Part I

Type Extension and Polymorphism
Defining inheritance: Type extension (1)

Type definitions
- date, datetime

Fortran concept:
- type extension
  - single inheritance only

```
type :: date
  private
  integer :: yr, mon, day
end type
:
type, extends(date) :: datetime
  private
  integer :: hr, min, sec
end type
:
type(datetime) :: o_dt
```

- re-use date definition
  - datetime a specialization (or subclass) of date
  - date more general than datetime

- instantiation of objects
  - can be performed as with “standard” derived types
  - other possibilities discussed later
Defining inheritance: Type extension (2)

- **Accessing component data**
  - **inherited** components
    - `o_dt%day, o_dt%mon, o_dt%yr`
  - **additional** components
    - `o_dt/hr, o_dt%min, o_dt%sec`
  - **parent** component(s)
    - object of parent type
    - recursive references possible
    - is itself inherited
  - `o_dt%date`

- **Parent type can be empty**
  - will discuss abstract types later

- **Can have zero additional components**
  - use only for type differentiation
  - or additional **type bound procedures** not available to parent type

- **Type parameters are also inherited**

- **Which types can be extended?**
  - must be derived type
  - may not have the `bind` or `sequence` attribute
Extending component accessibility

- Fortran 2003 allows setting accessibility for each type component individually

```fortran
  type :: date
    private
    integer :: yr, mon, day
    character(len=5), public :: tag = 'none'
  end type
  type, extends(date) :: datetime
    private
    integer :: hr, min, sec
  end type
```

- all accessibility attributes are also inherited
- extended type in different module cannot access private components of parent type

   this is the same as in Fortran 95
   need accessor methods
Polymorphism (1)

Polymorphic objects:

- **class(date), ... :: o_poly_dt**
  - **declared** type is **date**
  - **dynamic** type may vary at run time
    - may be **declared** type and all its (known) extensions
  - **type compatibility**
  - direct access only possible to components of **declared** type
    - compiler lacks knowledge

Data item can be

- pointer or allocatable variable
- dummy data object

Example:

```fortran
class(date), pointer :: p_d
type(date), target :: o_d
type(datetime), target :: o_dt :
p_d => o_dt
... = p_d%hr     ! illegal
p_d => o_d
... = p_d%hr     ! worse than illegal
... = p_d%mon    ! OK
```

- dynamic type changes at run time
Polymorphism (2):
Dynamic creation of polymorphic entities

- **Typed allocation**
  
  ```fortran
  class(x), pointer :: p
  class(x), allocatable :: q :
  allocate(xx :: p, q)
  ```

  - `xx` of type `x` or an extension of `x`
  - note that allocatable scalars are allowed in
  - usual difference between pointer and allocatable

- **Sourced allocation**
  
  ```fortran
  class(xx) :: src
  class(x), allocatable :: cpy :
  ! define src
  allocate(cpy, source = src)
  ```

  - `xx` of type `x` or an extension of `x`
  - produces a clone of `src`
  - deep copy for allocatable components
  - shallow copy for pointer components

**For disassociated pointer/unallocated allocatable objects:**
- dynamic type is equal to declared type

©2006-8 LRZ
Polymorphism (3):
Dynamic type/class resolution

- **select type** construct
  - provides access to dynamic parts
  - executes alternative code depending on dynamic type

**Execution sequence:**
- at most one block is executed
- selection of block:
  1. find **type guard** exactly matching the dynamic type
  2. if none exists, select **class guard** which most closely matches dynamic type and is still type compatible
     - at most one exists
  3. if none exists, execute block of **class default** (if it exists)

**Access to components**
- in accordance with resolved type (or class)

```plaintext
class(date), ... :: dt :
select type(dt)
type is (date) :
type is (datetime)
  ... = dt%hr  ! this is OK
class is (...) :
class default :
end select
```
Polymorphism (4): Additional remarks on dynamic type resolution

Can introduce an **associate name** to abbreviate referencing:

```fortran
select type(o => x%dt)
  type is (date)
  : 
  type is (datetime)
  ... = o%hr
  : 
end select
```

- assumption: subobject `dt` of `x` is of **class date**

Type selection allows both
- run time type identification
- run time class identification

It is necessary to ensure **type safety** in the Fortran object model

**Recommendations:**
- test each guard for each new subclass separately
- use **class default** to check for incompletely covered inheritance DAG
Polymorphism (5):
Type inquiry intrinsics

Compare dynamic types:

```plaintext
extends_type_of(a, mold)
same_type_as(a, b)
```

- functions returning a logical value
- require extensible types as arguments
- arguments can be polymorphic or non-polymorphic
Polymorphism (6): Dummy arguments

**Example:**

```
subroutine inc_day(dt, inc)
  class(date), intent(inout) :: dt
  integer, intent(in) :: inc
  ! implementation omitted
end subroutine
```

- increment **date** object by a given number of days

**Inheritance mechanism**
- actual argument can be
  - polymorphic or non-polymorphic, of declared type of dummy or an extension
  - type compatibility
- **dynamic** type of actual argument is assumed by dummy

**Example continued:**

```
subroutine inc_sec(dt, inc)
  class(datetime), &
  intent(inout) :: dt
  integer, intent(in) :: inc
  ! implementation omitted
end subroutine
```

- increment **datetime** object by a given number of seconds
- cannot take objects of type **date** as actual argument
Polymorphism (7): Unlimited polymorphic (UP) objects

- An object capable of being of any of
  1. intrinsic
  2. extensible
  3. non-extensible
  type is called unlimited polymorphic

Example:

```
class(*), pointer :: poly_pt
```

- no declared type
- type compatible with all entities

- An UP pointer can point to anything:

```
type(datetime), target :: o_dt
real, pointer :: rval :
poly_pt => o_dt
allocate(rval) ; rval = 3.0
poly_pt => rval
```

- Properties:
  - type information is only maintained for 1.+2.
  - case 3: types with the same structure are considered the same type

- Dereferencing is illegal

```
poly_pt => o_dt
write(6, *) poly_pt%yr
! will not compile
```
Polymorphism (8): Dereferencing an unlimited polymorphic object

**Need to perform dynamic type resolution**
- Use of intrinsic type guard allowed in this situation
- Allocation of UP object:
  - must use typed or sourced allocation
  - and specify type parameters if applicable

**Non-extensible target**
- can de-reference

```
type(datetime), pointer :: pt
select type (poly_pt)
type is (datetime)
  write(6, *) poly_pt%yr
pt => poly_pt
type is (real)
  write(6, '(f12.5)') poly_pt
end select
```

```
type, bind(c) :: cvec
  real(c_float) :: x(3)
end type

type :: ivec
  sequence ; integer :: i(3)
end type

type(cvec), target :: ov
type(cvec), pointer :: pv
type(ivec), pointer :: iv
ov = cvec( (/1.1,2.2,3.3/) )
poly_pt => ov
pv => poly_pt
iv => poly_pt
```

**but is not type-safe**
- also allowed for lhs:
  - (another) unlimited polymorphic pointer
Polymorphism (9)
UP object as subroutine argument

UP dummy argument

```fortran
subroutine upt(this, ...)  
  class(*) :: this  
end subroutine
```

- actual argument can be of any type

UP pointer or allocatable dummy argument

```fortran
subroutine upt_pt(this, ...)  
  class(*), pointer :: this  
end subroutine
```

- type compatibility implies that the actual argument must also be UP and have same attribute

subroutine can be used to
- establish pointer association, or allocate pointer (depend on intent), or
- (de)allocate allocatable entity
- resolve dynamic type via select type
- hand on to another subroutine

```
class(*), pointer :: pp  
!: client usage  
call upt_pt(pp, ...)
```

Not allowed:

- hand UP actual argument to non-UP dummy argument
Part II

Types and Procedures
Type bound procedures (1):
Binding a subroutine to a type

- **added to type definition**

  ```
  type :: date
      private
      integer :: yr, mon, day
  contains
      procedure :: inc => inc_day
      procedure :: set_date
  end type
  ```

  and used by client as

  ```
  type(date) :: dt :
  call dt%set_date(2006,12,12)
  call dt%inc(12) ! Christmas
  ```

- **Properties:**
  - **name mapping**
    - to existing subroutine, or
    - implement subroutine of given name
  - **passed object**
    - type compatible
    - first argument by default (this can be changed)
    - must be scalar, non-pointer, non-allocatable
  - **Semantic intent for inc**
    - want smallest granularity
  - **private clause does not extend to TBPs**
    - unless separately specified in `contains` part of type definition

- **works, but not as intended**

  ```
  type(datetime) :: dtt :
  call dtt%inc(120) ! by seconds?
  ```
Diagrammatic representation of type bound procedures

Fortran 95

```
mod_date

date
yr, mon, day

inc_day()
```

- implementation assumed accessible

Fortran 2003

```
mod_date

date
yr, mon, day

%inc()

inc_day()
```

- “%” indicates TBP
- implementation may not be accessible
Type bound procedures (2):
Overriding TBPs in subclasses

Not every subclass needs to define an override

- inheritance mechanism goes upward in DAG until a TBP is found
  - match is with dynamic type
  - module procedure does not allow this
- uniquely defined

If an overriding TBP is defined

- each must have same interface as the original
  - even same argument keywords!
- except
  - passed object dummy, which must be declared class (extension)

With the overriding TBP

- code (previous slide) will now work as intended

Note:

- uniform call mechanism
- no name conflicts with unrelated class

```
type :: date
  : ! as before
contains
  procedure :: inc => inc_day
  procedure :: set_date
end type
type datetime
  : ! as before
contains
  procedure :: inc => inc_sec
end type
```
Diagrammatic representation for overriding TBP

**mod_date**

- **date**:
  - yr, mon, day
  - `%set_date()`
  - `%inc()`
  - `inc_day()`

- **datetime**:
  - hr, min, sec
  - `%inc()`
  - `inc_sec()`

**non-overridden procedures:**

- need not replicate since inherited
Type bound procedures (3):
Enforcing base type inheritance

- **Passive (by client)**
  - invoke method on parent type

  ```fortran
  type(datetime) :: dtt
  : call dtt%date%inc(120) ! by days
  ```

- **Active (enforced by compiler)**
  - set_date as example:

  ```fortran
  type :: date
  : ! as before
  contains
    procedure :: inc => inc_day
    procedure, non_overridable :: &
      set_date
  end type

  subroutine set_date(this, &
    yr, mon, day, hr, min, sec)
  class(date), intent(out) :: &
    this
  integer(ik), intent(in) :: &
    yr, mon, day
  integer(ik), intent(in), &
    optional :: hr, min, sec :
  select type(this)
  type is (date)
    : ! further type guards
  end select
  end subroutine
  ```

  - already treats all cases

- **Implementation:**

- **Other rationales are possible**
Type bound procedures (4):
Variations on passed object dummy

1. Have **method**
   subroutine old_meth(a,b,..., &
       this, ...)
   class(xx) :: this
   :
       want to modernize:
   type :: xx
       ! unchanged
   contains
       procedure, pass(this) :: &
           m_tbp => old_meth
   end type xx
   
   **only 1 change to** old_meth
   
   unless additional component
   references needed for extensions
   
   **need explicit interface**
   
   **omit this** on TBP call

2. Module procedure is bound to multiple types
   
   need to explicitly specify
   non-default **pass** in at least
   one of the type definitions

3. Method does not involve
   object itself at all
   
   **possible reason:**
   manipulating module globals
   
   type :: yy
       ! whatever
   contains
       procedure, nopass :: tbp
   end type yy
Type bound procedures (5): Generic TBPs

- Lift restriction of only varying passed object
- Example:
  - add increment by date

```fortran
module type
  type :: date
    : ! as before
  contains
    private
      procedure :: inc_day, inc_date
      generic, public :: inc =>
        inc_day, inc_date
  :
end type
```

- Interface of `inc_date`:

```fortran
subroutine inc_date(this, diffd)
  class(date) :: this
  type(date), intent(in) :: diffd
```

- Only generic name available in example
- Resolution of generic TBP's
  - For all non-passed dummy arguments
    - type incompatible
    - TKR based on `declared` type (!)
    - number can vary
  - passed object dummy
    - as for TBP (deferred to run-time)

- Standard generic resolution
  - requires type incompatibility
    - for at least one non-optional arg.
  - subclassing only cannot be resolved
    - (non)generic TBPs are therefore qualitatively different!
Type bound procedures (6):
Generic overriding and overloading

Subclasses may
- need to override a specific
  to get correct semantics,
  or
- need to add a specific

Operator overloading:
- operator, assignment, derived
type I/O specification
- can be defined as unnamed
generic TBP
- not blocked by `use, only`
- usual resolution rules apply

```plaintext
type :: datetime
  : ! as before
contains
  private
  ! override for granularity
procedure :: inc_day => inc_sec
  ! add new method to generic set
procedure :: inc_datetime
generic, public :: inc => &
  inc_datetime
! beware TKR resolution
end type
```

```plaintext
type foo
  :
contains
  procedure :: plus_1
  procedure :: plus_2
generic, operator(+) => &
  plus_1, plus_2
end type
```

- specific TBP may not have the `nopass` attribute
Diagrammatic representation of generic TBPs

use italics to indicate generic-ness

- provide list of specific TBPs as usual
- overriding in subclasses can then be indicated as previously shown
Type-bound procedures (7): Finalization (aka Destruction)

- Have a class or object associated with additional state
  - open files
  - unfinished non-blocking network (MPI) calls
  - allocated pointer components

- Imagine object goes out of scope
  - unrecoverable I/O unit
  - communication breakdown
  - memory leak

- Solution: auto-destruct
  - provide type with a method which is called as soon as object of type goes out of scope,
  - is deallocated,
  - is passed to an intent(out) dummy argument, or
  - is the left hand side of an intrinsic assignment
Type-bound procedures (8): Defining a final TBP

**Type definition**

```fortran
type tp
  real, pointer :: r(:)
contains
final :: cleanup
end type
```

**Finalizer implementation**

```fortran
subroutine cleanup(this)
  type(tp) :: this
  if (associated(this%r)) then
! assume target was
! dynamically allocated
    deallocate(this)
  end if
end subroutine
```

**Differences to “normal” TPBs:**

- **not normally invoked by user**
  - automatically executed as described on previous slide
- **must have a single dummy argument**
  - of type to be finalized
  - non-polymorphic
  - non-pointer, non-allocatable
  - all length type parameters assumed
- **generic set of finalizers is possible:**
  - rank
  - kind parameter values
  - multiple execution order processor-dependent
Diagrammatic representation of Finalizers

Finalizer is **not** inherited by extensions
- reflected in nonpolymorphic argument
- exact type match

Layering of finalizers

If an object of type `tp` goes out of scope
- first `cleanup()` is called
- then `destroy()`
  - unless contd is a pointer component, which needs to be accounted for in `cleanup()`
If \( tp \) is a subclass of \( base \), and an object of type \( tp \) goes out of scope

- first \( \text{cleanup()} \) is called
- then \( \text{destroy()} \)

this applies recursively in the case of more than one inheritance level.
Pointers to procedures

Up to F95:
- pointer can only point at data object with target attribute

New in F03:
- associate pointer with a procedure
- explicit or implicit interface
- target may be a function or subroutine
- association with target is analogous to dummy procedure
  - no generic or elemental interface possible

Procedure pointer variables:

```fortran
interface
  subroutine subr(x)
    real, intent(inout) :: x
  end subroutine
end interface subr

procedure(subr), pointer :: pr
!
! implicit interface pointers:
procedure(), pointer :: pr_2
external, pointer :: pr_3 => null()
real :: y
!
! client use:
pr => subr; call pr(y)
```

target is a procedure known by explicit interface
- or implicit interface
  - avoid if possible

implementation of subr() only needed if dereferenced (?)
Procedure pointers as type components: object-bound procedures

```fortran
type xx
  :
  procedure(foo), pointer :: &
    p => null()
contains
  : ! TBPs come here!
end type
```

- **Properties as for variables**
  - for explicit interface, component can point to any procedure with the same interface

- **Important difference:**
  - by default, the calling object is passed as 1st argument
  - need to have interface like (see foo above right)

```fortran
subroutine foo(this, ...)
  class(xx) :: this
  ! must be polymorphic
  :
end subroutine
```

- or alternatively use `nopass` attribute in type definition
- `nopass` must be specified if interface is implicit

**Client use in this example:**

```fortran
type(xx) :: o_xx
procedure(foo) :: bar ::
  :
o_xx%p => bar
call o_xx%p(...)
```
Part III

Interfaces
## Dummy argument association for (non-) polymorphic objects

| Actual argument | Dummy argument type (…) | class (…) | class (…), & [pointer | allocatable] |
|-----------------|--------------------------|-----------|---------------------------------|
| type (…)        | Fortran 95 type matching rules apply | Actual argument must have same declared type as dummy or be an extension | No |
| class(...)      | Actual argument must have same declared type as dummy | (type compatibility) | Actual argument must have same declared type + attribute as dummy. [passed object dummy: No] |

- **Passed object:** auto-selects suitable TBP at run time
Function results and polymorphism

Remember – polymorphic object must be
- either a dummy argument
  - not the case for a function result
- or have the pointer or allocatable attributes

Hence, a function result can only be polymorphic if it additionally has either the pointer or allocatable attributes

However, default assignment of function result to a polymorphic object not allowed

Hence,
- assignment must occur to a non-polymorphic object, with respect to which the function results' declared type is the same or an extension, or
- sourced allocation must be used to transfer the result to a polymorphic object
Extensions to the interface concept (1): The `import` statement

- Interfaces in module specification section
  - `module mod_foo`
    - `type :: foo` : 
      - `end type`
    - `interface`
      - `subroutine m_foo(this, ...)`
        - `type(foo) :: this` :
          - `! illegal in ` F95
    - `end subroutine`
    - `end interface`
  - `end module`

  - access to host entities is not possible from interfaces
  - need to specify interface in separate module
    - access by `use` association
    - break encapsulation

- For F03 this was fixed:
  - `module mod_foo`
    - `type :: foo` :
      - `end type`
    - `interface`
      - `subroutine m_foo(this, ...)`
        - `import :: foo`
        - `type(foo) :: this` ! OK :
    - `end subroutine`
    - `end interface`
  - `end module`

  - can import any module entity
    - no argument: all entities available
  - also applicable to interfaces for dummy procedure arguments
    - of contained subroutines
Extensions to the interface concept (2): Abstract interfaces

Scenario:
- subroutines with same function as dummy argument

Solution:
- provide an explicit interface for which no actual implementation must exist

```fortran
module mod_interf
    abstract interface
        real function fun(x)
            real, intent(in) :: x
        end function
    end interface
end module

subroutine foo_23(x, f, y)
    use mod_interf
    procedure(fun) :: f
end subroutine foo_23
```

- requires replication of interface
Extensions to the interface concept (3): Interface classes

**Abstract type**
- no object of such a type can exist
- can have components or not
- can have TBPs
  - may enforce client override in subclass
  - typically specified via an abstract interface if no implementation available

```fortran
abstract interface
  subroutine open_handle(this,...)
    import :: handle
    class(handle) :: this
  :
  end subroutine
end interface
```

**Declaration fixes interface**
- Polymorphic variable can have abstract type as declared type
  - but not as dynamic type
- **deferred** attribute only allowed in abstract types

```fortran
type, abstract :: handle
 :
contains
  procedure(open_handle), &
  deferred :: open :
end type handle
```
Extensions to the interface concept (4): 
Subclassing an interface class

module mystuff
  use mod_handle
  type, extends(handle) :: &
    fhandle
contains
  procedure :: open => fopen
! will not compile without above
end type fhandle
contains
  subroutine fopen(this, ...
    class(fhandle) :: this
    : ! further details omitted
  end subroutine
end module mystuff

program client
  use mystuff, only : fhandle
! nothing else is needed
  implicit none
  type(fhandle) :: my_fhandle
  :
  call my_fhandle%open(...)
! object is passed as 1st dummy
  :
end program client

Extension module  Client usage
Diagrammatic representation of an interface class and its realization

Will typically use (at least) two separate modules
- e.g., binary version of abstract type vendor-provided
- abstract class and abstract interface indicated by italics
  - non-overridable part → “invariant method”
Extensions to the interface concept (5): Generalizing generic interface blocks

```
interface foo_generic
   module procedure foo_1
   module procedure foo_2
end interface
```

can be replaced by

```
interface foo_generic
   procedure foo_1
   procedure foo_2
end interface
```

with generalized functionality:
referenced procedures can be
- external procedures
- dummy procedures
- procedure pointers

Example:

```
interface foo_gen
   ! provide explicit interface
   ! for external procedure
   subroutine foo(x,n)
      real, intent(out) :: x
      integer, intent(in) :: n
   end subroutine foo
end interface
```

```
interface bar_gen
   procedure foo
end interface
```

- is legal in
- is illegal if
  ```
  module procedure
  is used
  ```
Problems with modules

- tendency towards monster modules for large projects
  - type component privatization also prevents being able to break up modules
- recompilation cascade effect
  - changes to module procedures would not actually require recompilation
  - workarounds are available, but somewhat clunky

Solution:

- split off implementations (module procedures) into separate files
- these files are called submodules
  - need only to recompile these and their descendants for changes to an implementation
- need to reference parent modules in submodule
  - access to module specifications is by host association
- need to spell out explicit interface in module specification section

Note: no compiler support today (Feb 2008)
Extensions to the interface concept (7): Submodules (cont'd)

Example

first, the module:

```fortran
module mod_date
  type :: date
    ! as previously
  end type
interface
  module subroutine &
    inc_day(dt, inc)
  import :: date
  class(date), &
    intent(inout) :: dt
  integer, intent(in) :: inc
end subroutine
end interface
end module
```

next, the submodule:

```fortran
submodule (mod_date) date_methods
  : ! specification part
contains
  module procedure inc_day
  ! interface taken from mod_date
  : ! implementation
end procedure inc_day
end submodule date_methods
```

note keyword indicating separate module procedure
default public attribute

can omit interface or specify exactly identical to module

specifications in submodule specification section
- only accessible within submodule or its children
- access contents via pointers / TBPs

similar for “normal” subroutines within submodule
Diagrammatic representation of submodules

“l” (for “implementation”) within circle indicates that a submodule is referred to
Final remarks on submodules: handling use dependencies

Direct or indirect use association of parent module is disallowed (pure F95 apart submodule use)

Independent use access is OK (Use e.g., polymorphism to satisfy type dependencies)
Part IV

Generic Programming
Generic container classes

**Example: linked list**

```fortran
type :: list
  type(<anything>) :: stuff
  type(list), pointer :: &
    next => null()
contains
  procedure :: add_item
  procedure :: delete_list
end type
```

- would like to put anything into a list
  - cf. C++ class template
- per-list constraints might be needed
- the above code fragment is not Fortran

**Container in general:**
- abstract data type containing collections of other objects
- methods provided to manage the object substructure

**Further classification:**
- value containers
  - store copies of objects
- reference containers
  - store references to objects
  - objects externally managed
  - must be persistent during lifetime of container
Exploiting the renaming feature (1)

Write container methods once

```
type :: list
type(dummy) :: stuff
type(list), pointer :: &
   next => null()
end type
interface add_item
   module procedure insert
end interface
contains
subroutine insert(this, stuff)
   type(list) :: this
   type(dummy), intent(in) :: &
   stuff
end subroutine
```

Type definitions here: within a single module

```
module all_types
   type :: t1
   :
   end type
type :: t2
   :
   end type
contains
:
end module all_types
```
Exploiting the renaming feature (2)

Create full set of generics:

```fortran
module mod_l_t1
  use all_types, dummy => t1
  include 'list.inc'
end module
module mod_l_t2
  use all_types, dummy => t2
  include 'list.inc'
end module
```

Client use:
- script-generated from full list of required types
- also requires use of renaming feature
- otherwise type definition of `list` is non-unique within client

```fortran
use all_types
use mod_l_t1, l_t1 => list
use mod_l_t2, l_t2 => list
type(l_t1) :: mylist_t1
type(l_t2) :: mylist_t2 :
call add_item(mylist_t1, o_t1)
call add_item(mylist_t2, o_t2) :
```

Issues:
- while only one implementation, somewhat clunky to use
  - generic only on implementation level, not on usage level
- for globals in implementation rename also needed
- some compilers have problems with generic resolution (bugs)
Alternative 1: fpp/cpp Preprocessing:
not standard-conforming, but simple

Executing preprocessor
- usually automatic for *.F files
- specify option otherwise
- example Intel Fortran:

```
ifort -c mod_type1.f90
ifort -c -fpp -DGENTYPE=type1 \ 
   -o list_type1.o list.f90
```

Apart from not needing explicit type renaming –
- no substantial improvement over previous method
- still cannot perform generic naming on client
Alternative 2: Using fully polymorphic objects

Features

- one implementation
- one module
  - namespace pressure reduced on client
- enforce type constraint
- can also implement value container:

```fortran
module mod_list
  type :: list
    class(*), pointer :: stuff
  type(list), pointer :: &
    next => null()
contains
  procedure :: add_item
contains
  subroutine add_item(this, stuff)
    class(list) :: this
    class(*), intent(in), &
      target :: stuff
:
    if (same_type_as(stuff,...)) &
      then
      this%stuff => stuff
    ! reference container
      allocate(this%next)
    else ; ... ; endif
  end subroutine
end module
```

Issues:

- dereferencing contained objects
  - requires select type
  - (partial) offload to client?
Parametrization of types (1)

Recall

Generalize this concept

Flavours allowed for parameters:

- **length type parameter**
  - actual value need not be known at compile time ("deferred")
  - must be determined at run time

- **kind type parameter**
  - must be known at compile time
  - however need not necessarily always refer to `kind` numbers

Intrinsic types:

- with exception of character only kind type parameters are allowed
- allows variable length strings (finally!)

```fortran
character(len=20,kind=c_char) :: line

character(len=:), pointer :: p
character(len=:), allocatable :: v2
character(len=122), target :: v1 :
p => v1  ! p has now len 122
allocate(v2(len=64))
```

Note: no compiler support today (Feb 2008)
Parametrization of types (2)

Derived types

Exactly analogous to intrinsic types

type :: matrix(rk, n, m)
  integer, kind :: rk
  integer, len :: n, m
  real(rk) :: entry(n, m)
end type

declaration of an object

integer, parameter :: dk = &
  selected_real_kind(15)
type(matrix(dk, 20, 30)) :: om
type(matrix(dk, :, :), &
  allocatable :: am
 :
allocate(am(n=15,m=20))
! by order or keyword

Type parameters are inherited

type, extends(matrix) :: mv(l)
  integer, len :: l
  real(rk) :: vector(l)
end type
:
type(mv(dk, :, :, :)), &
  allocatable :: o_mv
 :
allocate(o_mv(n=15,m=20,l=20))

can also omit entry in keyword list if default values for length/kind
type parameters are specified
Parametrization of types (3)

- Assumed type parameters can be used for:
  - dummy argument object, or
  - `select type` selector, or
  - `allocate` statement

  - only length type parameters

```fortran
subroutine foo(xm, ...)  
  type(matrix(dk, *, *)) :: xm  
```

  - values from `actual` arguments are taken over at subroutine call

Methods/Subroutines:

- must still implement separately for each kind

Type parameter enquiry:

- applies to intrinsic and derived types
- inquiry by type parameter name

```fortran
type(matrix(...)) o_m  
:  
print *, o_m%rk  
print *, o_m%n  
print *, o_m%m  
```

- also works for assumed type parameters
- can be used for scalars and arrays in same manner
Conclusion

- Support for generic programming in Fortran
  - is weak for Fortran 90/95
  - only slightly improved for Fortran 2003:
    - fully polymorphic objects / interface classes help
    - but do not get you all the way there (dereferencing!)
    - analogue to template metaprogramming not available
  - no further improvements planned for Fortran 2008

- The hope is that more features will be offered in the post-2008 iteration of the standard
Part V

Enhancements of I/O functionality
I/O for derived data types

Non-trivial derived data type

type :: list
   character(len=,:), &
   allocatable :: name
   integer :: age
   type(list), pointer :: next
end type

can perform I/O using suitable module procedures or TBPs

Disadvantages:
  recursive I/O disallowed
  I/O transfer not easily integrable into an I/O stream
     defined by edit descriptor for intrinsic types and arrays
     or sequence of binary I/O statement

F03 :

  enables binding a subroutine to an edit descriptor

  type(list) :: o_list
  o_list

  ! set up o_list
  write(unit, fmt='(dt ...)', ...) & o_list

  example shows formatted output
     bound subroutine called automatically when dt encountered

  other variants are enabled by using generic TBPs or generic interfaces

  can use recursion for hierarchical types
Binding I/O subroutines to derived types

- Interface of subroutines is fixed
  - with exception of the passed object dummy

- Define as special generic type bound procedure

```fortran
type :: foo
  :
contains
  :
  generic :: read(formatted) => rf1, rf2
  generic :: read(unformatted) => ru1, ru2
  generic :: write(formatted) => wf1, wf2
  generic :: write(unformatted) => wu1, wu2
end type
```

- generic-ness refers to rank, kind parameters of passed object

- Define via interface block

```fortran
interface read(formatted)
  module procedure rf1, rf2
end interface
```
DTIO module procedure interface
(dummy parameter list determined)

subroutine rf1(dtv,unit,iotype,v_list,iostat,iomsg)
subroutine wu1(dtv,unit,iostat,iomsg)

- dtv: scalar of derived type
  may be polymorphic
  suitable intent

- unit: integer, intent(in) – describes I/O unit or negative for internal I/O

- iotype (formatted only): character, intent(in)
  'LISTDIRECTED', 'NAMELIST'
  or 'DT'//string
  see dt edit descriptor

- v_list (formatted only): integer, intent(in) – assumed shape array
  see dt edit descriptor

- iostat: integer, intent(out) – scalar, describes error condition
  iostat_end / iostat_eor
  zero if all OK

- iomsg: character(*) – explanation if iostat nonzero
Limitations for DTIO subroutines

- I/O transfers to other units than unit are disallowed
  - I/O direction also fixed
  - internal I/O is OK (and commonly needed)
- Use of the statements
  - open, close, rewind
  - backspace, endfile
  is disallowed

- File positioning:
  - entry is left tab limit
  - no record termination on return
  - positioning with
    - rec=... (direct access) or
    - pos=... (stream access)
  is disallowed
Example:

```fortran
write(20, '(dt 'MyDT' (2, 10) )') o_mydt
```

both `iotype` and `v_list` are available to the programmer of the I/O subroutine
- determine further parameters of I/O as programmer sees fit
Example: Formatted DTIO on a linked list

module mod_list
  type :: list
    integer :: age
    character(20) :: name
    type(list), pointer :: next
  contains
    generic :: &
    write(formatted) => wl
end type list
contains

recursive subroutine wl( &
  this,unit,iotype, &
  vlist,iostat,iomsg)
  class(list), intent(in) :: this
  integer, intent(in) :: unit
  integer, intent(in) :: vlist(:)
  character(len=*), &
    intent(in) :: iotype
  integer, intent(out) :: iostat
  character(len=*), intent(in) :: iomsg
! .. locals
  character(len=12) :: pfmt
! continued next slide
Example (cont'd):
DTIO subroutine implementation

! cont'd from previous slide
    if (iotype != 'DTList') return
    if (size(vlist) < 2) return
! perform internal IO to generate format descriptor
    write(pfmt, '(a,i0,a,i0)') &
        '(i',vlist(1),',a',vlist(2),')'
    write(unit, fmt=pfmt, iostat=iostat) this%age,this%name
    if (iostat /= 0 ) return
    if (associated(this%next)) then
! recursive call
        call wl(this%next,unit,iotype(3:),vlist,iostat,iomsg)
    end if
end subroutine
end module
Example (cont’d):
Client use

```fortran
  type(list), pointer :: mylist
  ! set up mylist
  ! open formatted file to unit
  write(unit, fmt='(dt 'List' (4,20))', iostat=is) mylist
  ! close unit and destroy list
```

- Final remarks: Unformatted DTIO
  - bound subroutine with shorter argument list
  - is automatically invoked upon execution of write statement

```fortran
  type(mydt) :: o_mydt
  ! unformatted writing (also) bound to mydt
  ! open unformatted file to unit 21
  write(21[, rec=...]) o_mydt
```

- additional parameters (e.g. record number) only specifiable in parent data transfer statement