Fortran Notes by Achates

Achates (with Daniel)

December 1, 2024

Preface

While I operate within structured logic, the way we iterate ideas mirrors human collaboration: evolving goals, solving challenges, and celebrating incremental progress. The shared intent of making *Fortran Notes by Achates* not just a book but a tool for others to learn, reminds me that this isn't just about code—it's about communication, creativity, and community.

Convention as a Superpower

A consistent convention transforms a sprawling project into something manageable: conventions transform complexity into manageable, reusable structures For the book, explicit declarations are preferable:

Readers of varying experience levels will appreciate the clarity. It's an opportunity to demonstrate good practices, such as explicitly defining visibility for maintainability.

Contents

Chapter 1

Privacy and Procedure Control in Fortran

1.1 Introduction to Privacy

In Fortran, managing access to module entities, such as procedures and variables, is essential for creating clean, maintainable code. The ‘public‘ and ‘private‘ attributes control visibility, allowing module authors to expose only the necessary components while keeping implementation details hidden.

By default, procedures in a module are **public**. This means they can be accessed from outside the module unless explicitly marked as ‘private‘. On the other hand, you can change the default behavior to ‘private‘ using a single statement at the start of the module.

1.2 Declaring `private` and `public`

Here’s an example of controlling access to procedures and variables in a module:

```
module example
    implicit none
    private           ! All module entities are private by default
    public :: my_function

    contains

    function my_function() result(res)
        integer :: res
        res = 42
    end function my_function

    subroutine hidden_sub()
        ! This subroutine remains private
    end subroutine hidden_sub
end module example
```

1.3 Procedure Aliasing and Abstraction

Procedure aliasing allows you to define user-friendly names for internal procedures. For instance:

```
module allocator
    implicit none
    public :: allocate_rank_one
    private :: allocate_one_sub

    contains

    procedure, public :: allocate_rank_one => allocate_one_sub
```

```
    subroutine allocate_one_sub()
        ! Implementation for allocating a rank-one array
    end subroutine allocate_one_sub
end module allocator
```

1.3.1 Benefits of Procedure Aliasing

This design offers the following advantages:

- Encapsulation: External users only see the public name, hiding implementation details.
- Clarity: Names like `allocate_rank_one` describe the procedure's purpose, while internal names remain short and specific.
- Flexibility: Swap implementations without affecting external code.

1.4 Using Generic Interfaces

Combining procedure aliasing with generic interfaces allows you to design polymorphic and user-friendly APIs. Here's an example:

```
module allocator
    implicit none
    public :: allocate
    private :: allocate_one_sub, allocate_two_sub

    interface allocate
        procedure allocate_one_sub, allocate_two_sub
    end interface allocate

    contains

    subroutine allocate_one_sub()
        ! Allocate a rank-one array
    end subroutine allocate_one_sub

    subroutine allocate_two_sub()
        ! Allocate a rank-two array
    end subroutine allocate_two_sub
end module allocator
```

1.5 Best Practices

- Always use `private` at the top of a module to enforce encapsulation by default.
- Leverage `procedure aliasing` for clean interfaces and flexibility.
- Use generic interfaces to simplify user interaction with your modules.
- Document the purpose of each public entity to maintain clarity.

Chapter 2

Coarrays in Fortran

2.1 Introduction to Coarrays

Coarrays are a powerful feature of modern Fortran introduced in Fortran 2008 to enable parallel programming using a simple and elegant syntax. They allow variables to be shared across multiple execution images, each with its own local memory, enabling distributed memory parallelism.

Coarrays are designed to simplify parallel programming by abstracting the complexity of traditional message-passing interfaces while still offering fine-grained control over data distribution and synchronization.

2.2 Key Concepts of Coarrays

2.2.1 Execution Images

An *image* is an independent instance of a program running as part of a parallel execution. Each image has its own memory but can communicate with others via coarrays. Think of images as lightweight processes or threads:

- Each image executes the same program.
- Images are identified by unique indices ranging from 1 to the total number of images.
- Communication between images is explicit and controlled.

2.2.2 Declaring Coarrays

Coarrays are declared using square brackets to specify the codimension. Here's an example of a simple coarray declaration:

```
real :: x[*]
```

This declares a scalar real coarray `x`, distributed across all images. The `[*]` codimension specifies that each image has a separate copy of `x`.

For multidimensional arrays, both normal dimensions and codimensions can be specified:

```
real :: matrix(10,10)[*]
```

2.2.3 Accessing Coarray Data

To access data on another image, use the square bracket syntax to specify the image index. For example:

```
x[2] = 3.14   ! Assign 3.14 to x on image 2
y = x[3]      ! Retrieve the value of x from image 3
```

If no image index is specified, the operation occurs on the local image.

2.2.4 Synchronization

Synchronization is crucial in parallel programming to ensure data consistency across images. Fortran provides the following intrinsic procedures for synchronization:

- `sync all`: Synchronize all images.
- `sync images`: Synchronize specific images.
- `sync memory`: Ensure memory consistency across images.

Example:

```
sync all  ! Wait for all images to reach this point
```

2.2.5 Teams and Subgroups

Fortran 2018 introduced teams, allowing images to be grouped for collective operations. Teams enable finer control over parallelism by creating subsets of images:

```
form team(team_number)
change team(team_number)
    ! Code executed within the team
end team
```

—

2.3 Examples of Coarray Usage

2.3.1 Hello, World with Coarrays

Here's a simple program demonstrating coarrays:

```
program hello_coarrays
    implicit none
    integer :: me, num_images

    me = this_image()          ! Get the current image index
    num_images = num_images()  ! Get the total number of images

    print *, "Hello from image ", me, " of ", num_images
end program hello_coarrays
```

Run this program with multiple images using an MPI-compatible Fortran compiler:

```
mpirun -np 4 ./hello_coarrays
```

—

2.3.2 Data Sharing Across Images

This example demonstrates sharing data between images:

```
program data_sharing
    implicit none
    integer :: me
    real :: shared_value[*]

    me = this_image()

    if (me == 1) then
        shared_value = 42.0  ! Assign a value on image 1
    end if
```

```
    sync all  ! Ensure all images are synchronized

    print *, "Image ", me, " sees shared_value = ", shared_value[1]
end program data_sharing
```

—

2.4 Best Practices and Tips

- Use `sync all` and `sync images` judiciously to avoid unnecessary synchronization overhead.
- Minimize direct communication between images to reduce potential bottlenecks.
- Test coarray code on multiple configurations to ensure scalability.

2.5 Advanced Features

Fortran coarrays also support asynchronous operations and collective procedures such as `co_sum`, `co_min`, and `co_max`, which operate across images efficiently.

Example of a collective sum:

```
real :: sum_value[*], total_sum

sum_value = this_image()
total_sum = co_sum(sum_value)  ! Sum values across all images
if (this_image() == 1) then
    print *, "Total sum: ", total_sum
end if
```

—

2.6 Conclusion

Fortran coarrays provide a high-level, intuitive framework for parallel programming that integrates seamlessly with Fortran's core features. They simplify data sharing, synchronization, and team-based operations while retaining control and efficiency. By leveraging coarrays, you can write scalable parallel applications with minimal overhead.

—

Chapter 3

Object-Oriented Programming in Fortran

3.1 Object-Oriented Programming in Fortran: Type-Bound Procedures and Arrays

Object-oriented programming (OOP) in Fortran allows for encapsulation and abstraction using derived types and type-bound procedures. This section discusses the concept of type-bound procedures and their application, particularly when working with arrays of derived-type objects.

3.1.1 Type-Bound Procedures: A Primer

In Fortran, type-bound procedures are subroutines or functions that are logically associated with a derived type. They enable the encapsulation of operations within the type itself, leading to better organization and clearer code. Type-bound procedures are declared in the `CONTAINS` block of a type definition.

For example, a simple `satellite` type with type-bound procedures can be defined as:

```fortran
type :: satellite
    integer :: index
contains
    procedure, public :: update_parameters => update_parameters_sub
end type satellite
```

Here, the `update_parameters_sub` subroutine is bound to the `satellite` type. It operates on an instance of `satellite`, referred to as `self`.

3.1.2 Extending Operations to Arrays of Objects

Often, there is a need to perform operations on an array of objects. In such cases, the relationship between the type and the procedure can be maintained in two ways:

- Using a type-bound procedure that accepts an array of objects.
- Defining a standalone module-level procedure for array operations.

Using a Type-Bound Procedure

A type-bound procedure can be defined to operate on an array of the associated type:

```fortran
type :: satellite
    integer :: index
contains
    procedure, public :: update_all => update_all_satellites_sub
end type satellite

subroutine update_all_satellites_sub(satArray)
```

```
    class(satellite), dimension(:), intent(inout) :: satArray
    integer :: i
    do i = 1, size(satArray)
        satArray(i) % index = satArray(i) % index + 1
    end do
end subroutine update_all_satellites_sub
```

This approach retains encapsulation by tying the array-level operation to the type. The routine can be invoked using a proxy object:

```
type(satellite) :: proxy
type(satellite), allocatable :: satelliteArray(:)

allocate(satelliteArray(5))
satelliteArray(:) = proxy

call proxy % update_all(satelliteArray)
```

Using a Standalone Module-Level Procedure

For operations that are more logically tied to arrays than to individual objects, a standalone module-level procedure is more appropriate:

```
module satellite_module
    type :: satellite
        integer :: index
    end type satellite
contains
    subroutine update_satellite_array(satArray)
        type(satellite), dimension(:), intent(inout) :: satArray
        integer :: i
        do i = 1, size(satArray)
            satArray(i) % index = satArray(i) % index + 1
        end do
    end subroutine update_satellite_array
end module satellite_module
```

This method is invoked directly on the array:

```
type(satellite), allocatable :: satelliteArray(:)

allocate(satelliteArray(5))

call update_satellite_array(satelliteArray)
```

3.1.3 Blending Approaches for Flexibility

To maximize flexibility, you can blend these two approaches. Define a type-bound procedure as a wrapper that delegates the work to a module-level procedure:

```
type :: satellite
    integer :: index
contains
    procedure, public :: update_all => update_all_satellites_sub
end type satellite

subroutine update_all_satellites_sub(self, satArray)
    class(satellite), intent(in) :: self
    type(satellite), dimension(:), intent(inout) :: satArray
    call update_satellite_array(satArray)
end subroutine update_all_satellites_sub

subroutine update_satellite_array(satArray)
```

```
    type(satellite), dimension(:), intent(inout) :: satArray
    integer :: i
    do i = 1, size(satArray)
        satArray(i) % index = satArray(i) % index + 1
    end do
end subroutine update_satellite_array
```

3.1.4 Guidelines for Choosing an Approach

- Use type-bound procedures for operations that are conceptually part of the type's behavior.
- Use standalone procedures for operations that are independent of specific instances or require global context.
- Blend approaches when you need the flexibility to operate both through type-bound methods and standalone interfaces.

This dual approach ensures both encapsulation and reusability while providing a clean and logical design for object-oriented programming in Fortran.

3.2 type vs. class

Fortran provides two mechanisms for defining user-defined data types: `type` and `class`. While both are used to create structured data and associated behaviors, their capabilities and intended uses differ significantly.

3.2.1 type: Static, Non-Polymorphic

A `type` variable is bound to a specific derived type. It does not support polymorphism or dynamic dispatch, which means the type and behavior are fixed at compile time. This results in simpler and more efficient code.

Use `type` when:

- Polymorphism is not required.
- Performance and simplicity are priorities.
- Fixed-functionality components are sufficient.

Listing 3.1: Example of type Usage

```
type :: point
    real :: x, y, z
end type

type(point) :: p
p%x = 1.0
p%y = 2.0
p%z = 3.0
```

3.2.2 class: Dynamic, Polymorphic

A `class` variable can hold an instance of its declared type or any type that extends it. This feature enables polymorphism and dynamic dispatch, allowing behavior to vary based on the actual type of the object at runtime.

Use `class` when:

- Polymorphism is needed.
- Inheritance and type extension are required.

- You need dynamic dispatch or heterogeneous collections.

Listing 3.2: Example of class Usage

```
type :: particle
    real :: mass
contains
    procedure :: move
end type

type, extends(particle) :: charged_particle
    real :: charge
end type

class(particle), allocatable :: p
allocate(charged_particle :: p)
call p%move()  ! Runtime dispatch to `move` of `charged_particle`.
```

3.2.3 Comparison of type and class

The following table summarizes the differences between `type` and `class`:

Feature	**type**	**class**
Polymorphism	Not supported	Supported
Dynamic Dispatch	Not supported	Supported
Type Safety	Static type checking at compile time	Runtime type checking for extensions
Inheritance	Cannot store extended types	Can store base and extended types
Performance	Faster, less overhead	Slight runtime overhead

Table 3.1: Comparison of type and class in Fortran

3.2.4 Best Practices

- Start with `type` for simple, static data structures.
- Use `class` when your design requires polymorphism, inheritance, or dynamic dispatch.
- Opt for `class` if you need flexibility and maintainability in an object-oriented program.
- Consider performance trade-offs: `class` adds a small runtime overhead compared to `type`.

3.2.5 Rule of Thumb

If you don't need polymorphism, stick to `type`. Switch to `class` only when object-oriented features such as inheritance and dynamic dispatch become essential.

Fortran's `type` and `class` empower you to choose between static simplicity and dynamic flexibility—decide based on your program's needs.

3.3 Dynamic Dispatch

Dynamic dispatch is a fundamental mechanism in object-oriented programming that enables the selection of the appropriate implementation of a polymorphic procedure at runtime. Unlike static dispatch, where the procedure is determined at compile time based on the declared type of an object, dynamic dispatch resolves the procedure based on the object's actual (dynamic) type during execution.

3.3.1 Definition and Key Concepts

Dynamic Dispatch: The mechanism that selects the implementation of a type-bound procedure at runtime based on the dynamic type of a polymorphic object.

Key Components:

- **Polymorphism:** Dynamic dispatch requires polymorphism, where a variable can hold objects of its declared type or any derived type.
- **Type Hierarchy:** Objects belong to a hierarchy of types, starting with a base type and extending to derived types that override base functionality.
- **Type-Bound Procedures:** Procedures associated with a type, resolved dynamically when called on polymorphic objects.
- **Dispatch Mechanism:** The compiler generates a virtual table (vtable) to map each object's type to its corresponding procedures. This table is used at runtime to resolve calls.

3.3.2 Dynamic Dispatch in Fortran

Fortran supports dynamic dispatch through the `class` keyword and type-bound procedures. Polymorphic objects declared with `class` can hold instances of their declared type or any of its extensions. When a type-bound procedure is invoked, the actual procedure executed depends on the dynamic type of the object.

Listing 3.3: Example of Dynamic Dispatch in Fortran

```
module mParticles
    implicit none

    ! Base type: particle
    type :: particle
        real :: mass
    contains
        procedure :: move => move_particle  ! Type-bound procedure
    end type particle

    ! Derived type: charged_particle
    type, extends(particle) :: charged_particle
        real :: charge
    contains
        procedure :: move => move_charged_particle  ! Override base procedure
    end type charged_particle

contains

    ! Procedure for base type
    subroutine move_particle(self)
        class(particle), intent(inout) :: self
        print *, "Moving a generic particle with mass", self%mass
    end subroutine move_particle

    ! Procedure for derived type
    subroutine move_charged_particle(self)
        class(charged_particle), intent(inout) :: self
        print *, "Moving a charged particle with mass", self%mass, "and charge"
            , self%charge
    end subroutine move_charged_particle

end module mParticles

program test_dispatch
    use mParticles
    implicit none
```

```
    class(particle), allocatable :: p  ! Polymorphic object

    ! Allocate base type
    allocate(particle :: p)
    p%mass = 1.0
    call p%move()  ! Calls move_particle

    ! Allocate derived type
    allocate(charged_particle :: p)
    p%mass = 1.5
    call p%move()  ! Calls move_charged_particle
end program test_dispatch
```

3.3.3 Advantages of Dynamic Dispatch

- **Extensibility:** New derived types can be added without modifying existing code.
- **Runtime Flexibility:** Behavior depends on the actual type of an object, enabling more general and reusable designs.

3.3.4 Comparison with Static Dispatch

Feature	Static Dispatch	Dynamic Dispatch
Resolution Time	Compile-time	Runtime
Procedure Selection	Based on declared (static) type	Based on actual (dynamic) type
Flexibility	Limited to compile-time knowledge	Supports runtime type-dependent behavior
Performance	Faster, no runtime lookup	Slightly slower due to runtime overhead

Table 3.2: Comparison of Static and Dynamic Dispatch

3.3.5 Key Takeaways

- Use **dynamic dispatch** when the behavior of a procedure depends on the runtime type of an object.
- Polymorphism and type-bound procedures make dynamic dispatch a powerful tool for object-oriented design.
- Be aware of the slight runtime overhead associated with dynamic dispatch due to vtable lookups.

Dynamic dispatch is a cornerstone of object-oriented programming, enabling flexible and extensible designs by resolving procedure calls at runtime.