Object-Oriented Programming in Fortran

Jonas Lindemann, LUNARC

LUND UNIVERSITY

Object-oriented Programming

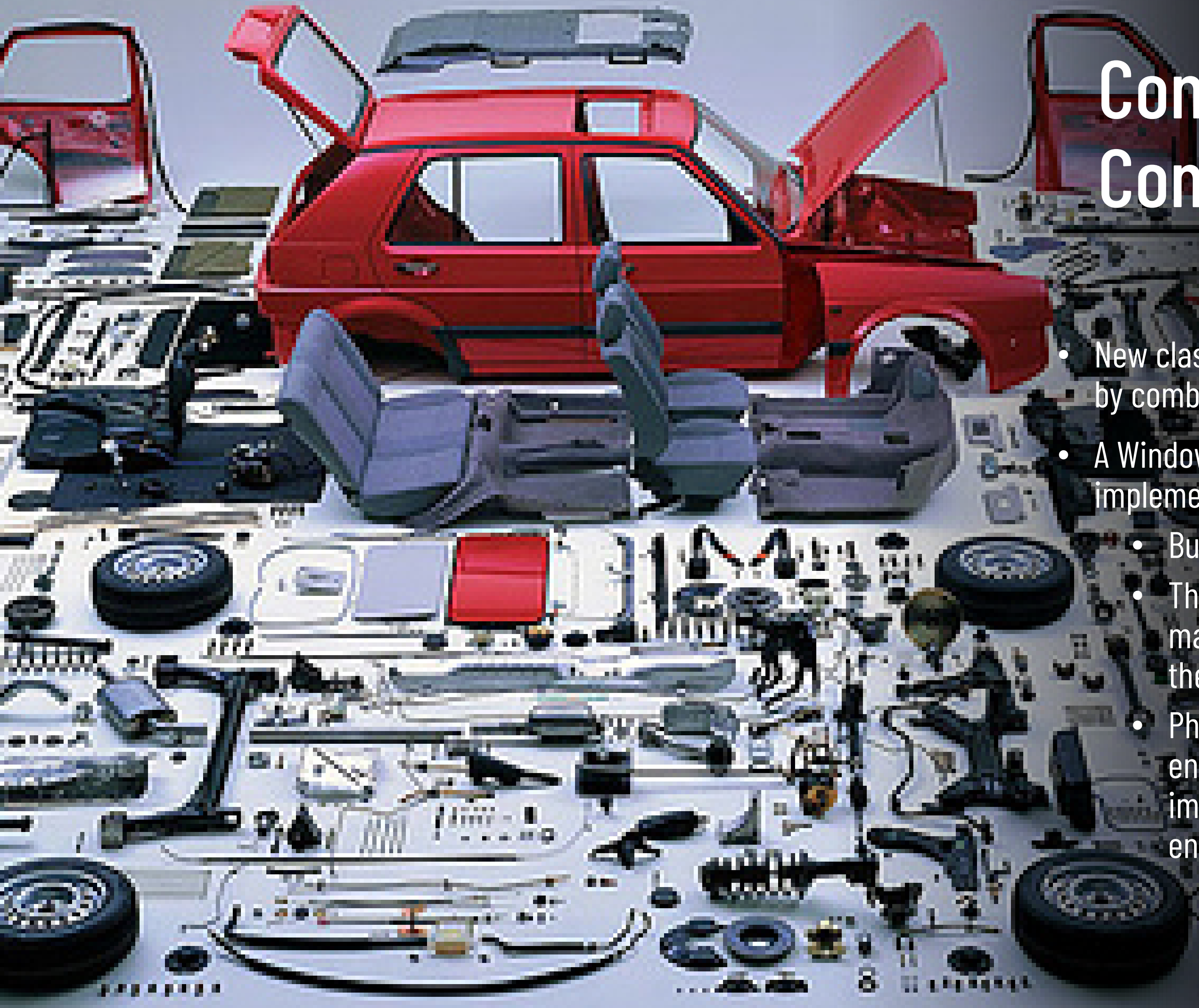
- Introduces the concepts of objects in programming
- Objects contain both data and code in a single unit.
  - Data often called attributes or properties
  - Code associated with objects is often called methods.
- The blueprint of an object is the class
  - Class describeds the data and methods of specific object type.
  - Objects/intances are created/instantiated from classes

Concepts of OOP - Encapsulation

- Hides the actual implementation from the user
- Changing an object state/data should be done through access methods
- Enables checking data validation.
- Enables changes of the implementation without breaking existing codes.
- Creates more robust code by hiding implementation details.

Concepts of OOP - Inheritance

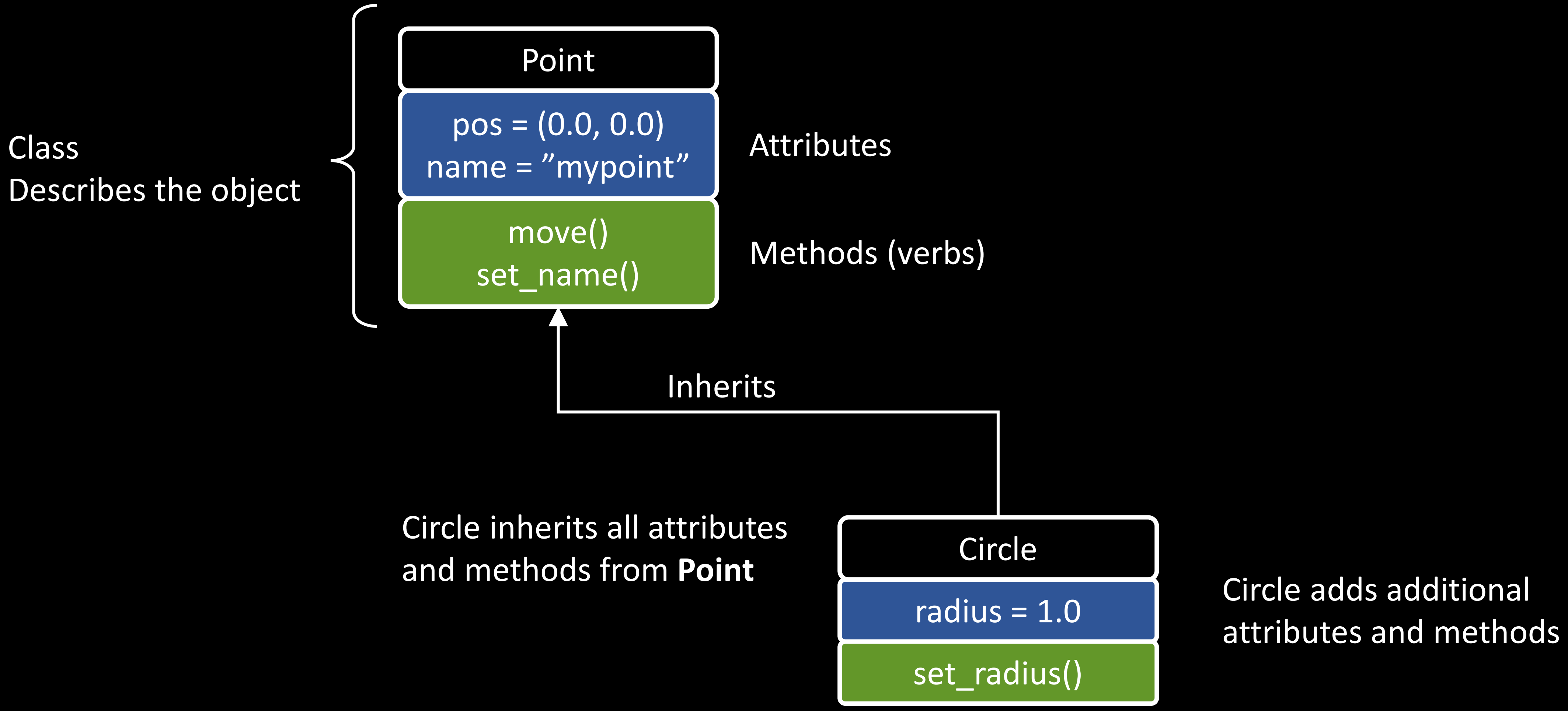
- Existing classes can be specialised and extended to create similar classes with updated functionality
- Enable code reuse by using functionality in existing classes.
- Make large libraries easier to use by providing similar methods and attributes for different related classes.



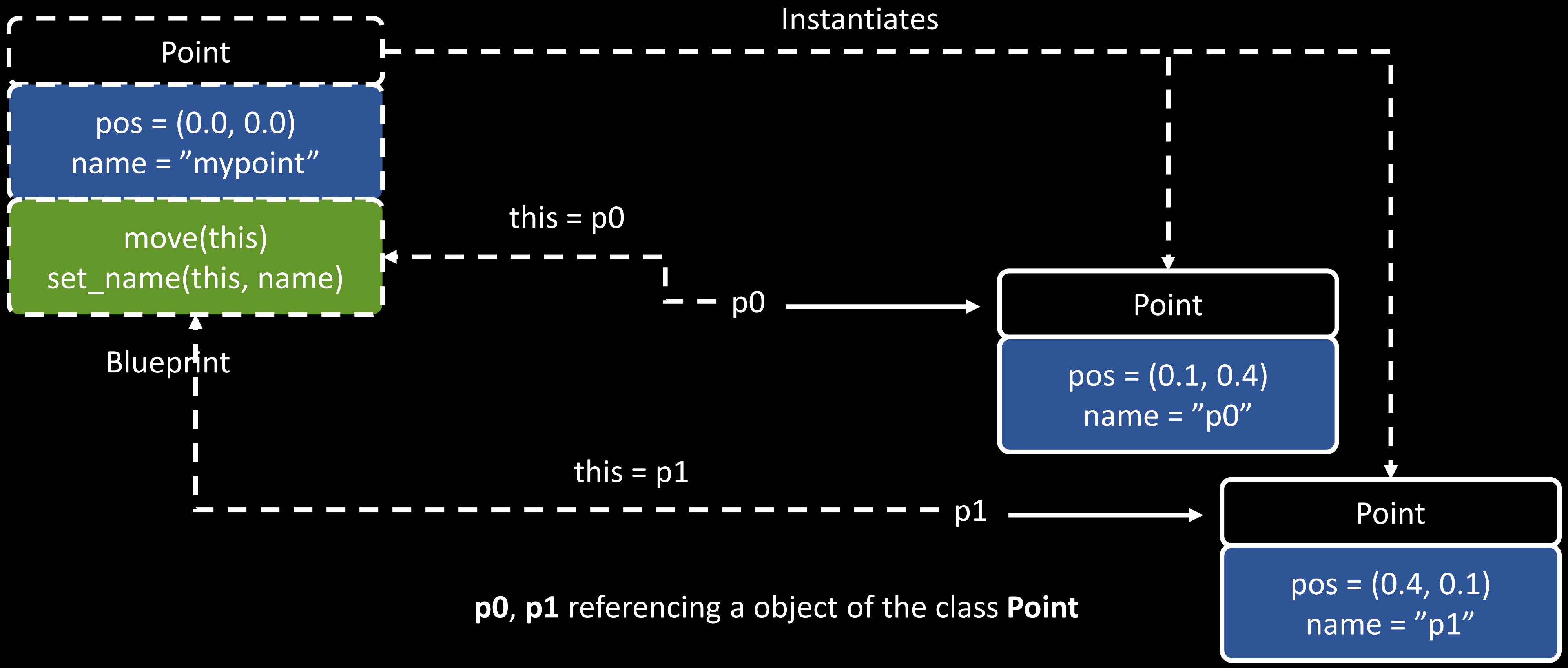
Concepts of OOP - Composition

- New classes can also be created that builds functionality by combining different object instances in a single class.
- A Window class can own multiple different objects to implement window functionality
  - Buttons, Controls, Canvas and more...
  - These classes do not inherit functionality for the main Window class, instead they add funtionality to the main window
  - Physical example: Car consists of 4 wheels and an engine. The car is the composite class implementing its functionality using wheels and the engine

Objects and inheritance

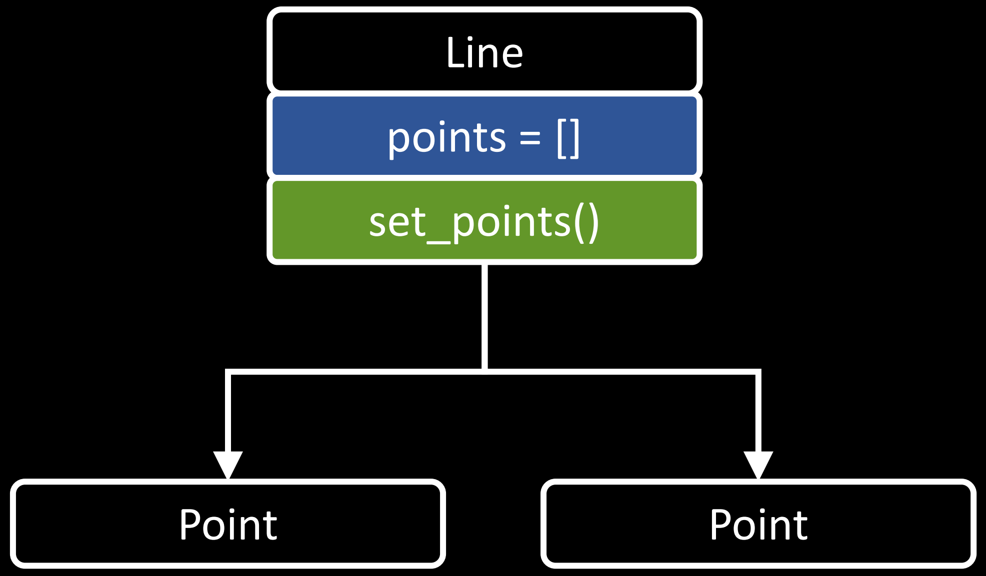


Instanciating objects



p0, **p1** referencing a object of the class **Point**

Composition



Objects in Fortran

Derived data types in Fortran

- Derived datatypes in Fortran is the base for OOP
- Custom data types can be defined consisting of existing Fortran datatypes
- Derived types are declared using the type([derived type]) keyword.
- A derived member variable is accessed using the % operator

```fortran
type Particle
    real :: pos(3)
    real :: vel(3)
    real :: mass
end type

! Instantiating a Particle object

type(Particle) :: p

! Accessing attributes

print*, p % pos(0)
print*, p % vel(1)
```

Classes in Fortran

- Classes in Fortran extends derived types with a contains section
- In the contains section the access- and initialisation methods can be defined
- The procedure keyword is used to map a method name to an implementing subroutine

```
module shape_classes

implicit none

type Shape
    real :: pos(2)
contains
    procedure :: init => shape_init
    ...
end type

contains

subroutine shape_init(this)

...

end subroutine

...
```

Class methods

- Class method in Fortran starts with a this-parameter
- this is a reference to the actual object / instance
- this is used to access the class attributes. % operator is used.
- this is declared as class([class name]) :: this

```
subroutine shape_init(this)

    class(Shape) :: this

    print*, 'shape_init()'
    this % pos = (/ 0.0, 0.0 /)

end subroutine

subroutine shape_set_pos(this, x, y)

    class(Shape) :: this
    real, intent(in) :: x
    real, intent(in) :: y

    this % pos(1) = x
    this % pos(2) = y

end subroutine
```

Class constructors

- When an object / instance is created its attributes has to initialised
- In Fortran there is no constructor concept for initialising an object upon creation.
- Instead an init-routine can be defined.

```fortran
type Shape
    private
    real :: pos(2)
contains
    procedure :: init => shape_init
    procedure :: set_pos => shape_set_pos
    procedure :: print => shape_print
end type

...

type(Shape) :: s

call s % init()
```

Class constructors cont...

- It is possible to create a special function for returning an initialised object / instance
- Not strictly required

```fortran
interface Shape
    module procedure shape_constructor
end interface

...

function shape_constructor() result(shape_inst)

    type(Shape) :: shape_inst
    call shape_init(shape_inst)

end function

...

type(Shape) :: s

    s = Shape() ! Shape constructor
!   call s % init()
    call s % print()
    call s % set_pos(2.0, 2.0)
    call s % print()
```

Class destructors – Cleaning up

- When an object is destroyed or deallocated a special routine, destructor can be called
- Should be used if the object has allocated arrays or other objects

```
type Shape
    ...
Contains
    ...
    final :: shape_destructor
end type

...

subroutine shape_destructor(this)

    type(Shape) :: this

    print*, 'shape_destructor()'

end subroutine
```

Allocatable objects

- Objects / instances can also be allocated on the heap
- Here the destrutor will be called when deallocating the object.

```
type(Shape), allocatable :: s2

...

allocate(s2)

call s2 % init()
call s2 % print()

deallocate(s2)
```

```
shape_init()
shape_print()
x =     0.00000000
y =     0.00000000
shape_destructor()
```

Arrays of objects

- Objects can also be stored in arrays.
- Constructor must be called for each all objects in the array

```
type(Shape) :: static_shapes(20)

do i=1,20
    call static_shapes(i) % init()
    call static_shapes(i) % print()
end do
```

```
shape_init()
shape_print()
x =     0.00000000
y =     0.00000000
shape_init()
shape_print()
x =     0.00000000
...
```

Arrays of objects

- Arrays of objects can also be allocated dynamically
- Important to deallocate array after use.

```
type(Shape) :: static_shapes(20)

do i=1,20
    call static_shapes(i) % init()
    call static_shapes(i) % print()
end do
```

```
shape_init()
shape_print()
x =     0.00000000
y =     0.00000000
shape_init()
shape_print()
x =     0.00000000
...
```

Arrays of dynamically allocated objects

- A bit more tricky ;)
- Need to use pointer directives
- Need allocate object assign pointer to array of pointers.
- Need to deallocate both individual object pointers as well as array.

```
type SimpleShapeArray
    type(Shape), pointer :: element
end type

type(Shape), pointer :: p_simple_shape

...

allocate(simple_shapes(20))

do i=1,20
    allocate(p_simple_shape)
    call p_simple_shape % init()
    simple_shapes(i) % element => p_simple_shape
end do

do i=1,20
    p_simple_shape => simple_shapes(i) % element
    call p_simple_shape % print()
end do

do i=1,20
    p_simple_shape => simple_shapes(i) % element
    deallocate(p_simple_shape)
end do

deallocate(simple_shapes)
```

Encapsulation

Hiding the inner workings of objects

Encapsulation

- In a Fortran class the attributes can be hidden from direct access using the private directive.
- private can be applied to all attributes by placing it directly after the type definition
- Just like module variables specific variables can be hidden by specifying private directive in the variable declaration

```fortran
type Shape
    private
    real :: pos(2)
contains
    procedure :: init => shape_init
    procedure :: set_pos => shape_set_pos
    procedure :: print => shape_print
    final :: shape_destructor
end type
```

Accessing attributes

- To enable flexibility of implementation the attributes should only be accessed using functions / subroutines in the class.

```
subroutine shape_set_pos(this, x, y)

    class(Shape) :: this
    real, intent(in) :: x
    real, intent(in) :: y

    this % pos(1) = x
    this % pos(2) = y

end subroutine
```

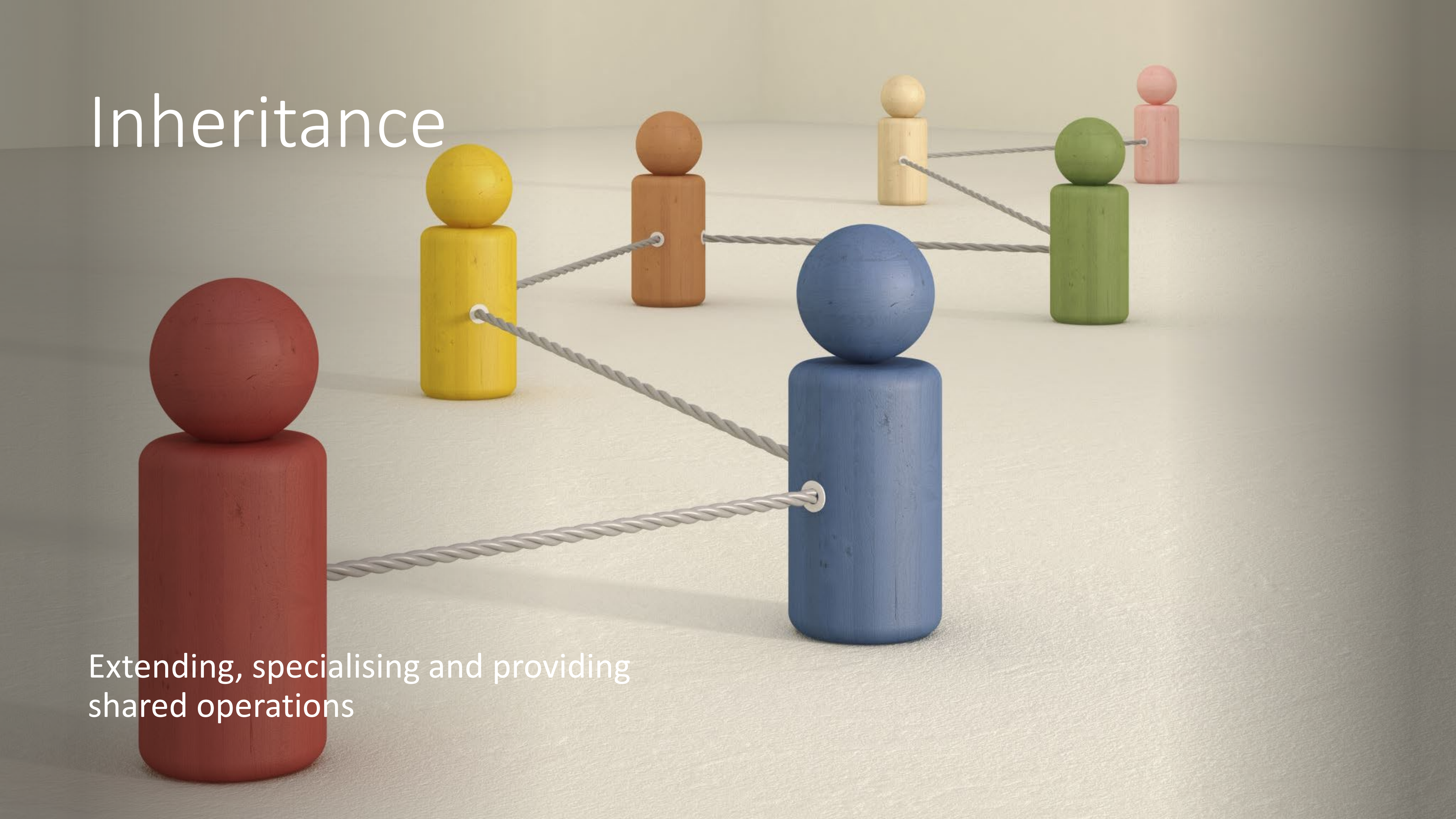
Accessing internal arrays through access methods

- Accessing internal arrays in a class has to be done using pointers

```
real(dp), pointer :: pos(:,:)
real(dp), pointer :: vel(:,:)
real(dp), pointer :: r(:)

pos => psys % positions()
vel => psys % velocities()
r => psys % radius()
```

```
type ParticleSystem
Private
    ...
    real(dp), pointer :: m_pos(:,:)
    real(dp), pointer :: m_vel(:,:)
    real(dp), pointer :: m_r(:)
    ...

function particle_system_positions(this) result(arr)

    class(ParticleSystem) :: this
    real(dp), pointer :: arr(:,:)

    arr => this % m_pos

end function
```

Inheritance



Extending, specialising and providing shared operations

Classes can inherit functionality from existing classes

- In Fortran this is done by using the extends keyword in the type definition
- Note that we provide the same operations, but different implementations
- Square will inherit all attributes / methods from Shape
- We need to provide updated methods for init / print

```
type, extends(Shape) :: Square
    private
    real :: side
contains
    procedure :: init => square_init
    procedure :: print => square_print
end type
```

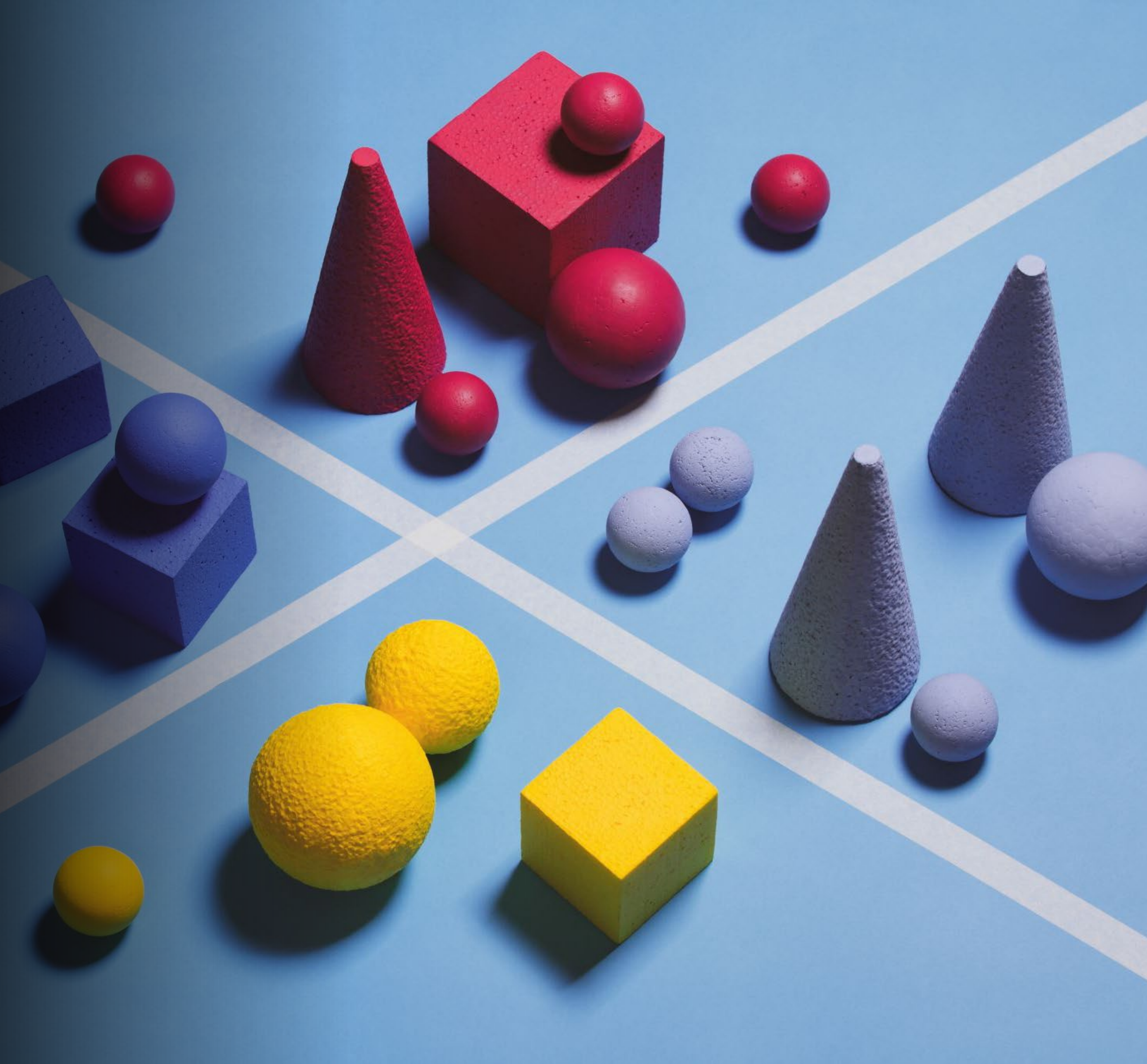
Constructing inherited classes

- A inherited class constructor must also initialise the base class
- Could be done by calling the base class init-method and then initialising derived attributes

```
subroutine square_init(this)

    class(Square) :: this

    print*, 'square_init()'

    call shape_init(this)

    this % side = 1.0

end subroutine
```

Polymorphism

The ability of an object to take many forms



Polymorphism

- An object that can pass at least 2 is-a tests
  - The Square class is also a Shape
  - The Circle class is also a Shape
- Static polymorphism
  - Compile time polymorphism
  - An object has 2 methods with the same name but different parameters
  - Method overloading
- Runtime polymorphism
  - A list consists of references to Shape derived types. Calling a method on a reference to a Shape in this list will call the correct method on the underlying object.

Dynamic polymorphism in Fortran

- Requires heap allocated objects.
- Must be declared with
  - class([classname]), pointer

```
class(Square), pointer :: p_square
class(Circle), pointer :: p_circle
class(Shape), pointer :: p_shape

allocate(p_square)
allocate(p_circle)

call p_square % init()
call p_circle % init()

print*, '-----'
p_shape => p_square
call p_shape % print()

p_shape => p_circle
call p_shape % print()
print*, '-----'
```

```
square_print()
side =    1.00000000
shape_print()
x =    0.00000000
y =    0.00000000
cirlce_print()
radius =    1.00000000
shape_print()
x =    0.00000000
y =    0.00000000
```

An array of Shape derived classes

- We defined a ShapeArray of Shape derived class pointers
- We allocate an array shapes with ShapeArray elements
- We can now assign Shape derived object to this array
- Calling print() on the Shape reference will call the right method in the corresponding object

```
type ShapeArray
    class(Shape), pointer :: element
end type

type(ShapeArray), allocatable :: shapes(:)

allocate(shapes(20))

do i = 1, 20
    allocate(p_square)
    shapes(i) % element => p_square
end do

do i = 1, 20
    p_shape => shapes(i) % element
    call p_shape % print()
end do
```

Composite classes

Building a car

Points and a Triangle

- We define a composite triangle consisting of 3 Point objects
- The triangle class has an array of pointer to Point-instances

```
type Point
private
    real :: m_x, m_y
contains
    procedure :: init => point_init
    procedure :: set_pos => point_set_pos
    procedure :: print => point_print
end type

type PointElement
    class(Point), pointer :: p_element
end type

type Triangle
private
    type(PointElement) :: m_points(3)
contains
    procedure :: init => triangle_init
    procedure :: get_point => triangle_get_point
    procedure :: print => triangle_print
end type
```

Points and a Triangle

- The Triangle allocates 3 Point objects
- The get_point() method is used to access the internal Point instances.
- We can access the point instances p_pX just like any other object

```fortran
class(Triangle), pointer :: p_triangle
class(Point), pointer :: p_p0
class(Point), pointer :: p_p1
class(Point), pointer :: p_p2

allocate(p_triangle)
call p_triangle % init()

p_p0 => p_triangle % get_point(1)
p_p1 => p_triangle % get_point(2)
p_p2 => p_triangle % get_point(3)

call p_p0 % set_pos(0.0, 0.0)
call p_p1 % set_pos(1.0, 0.0)
call p_p2 % set_pos(1.0, 1.0)

call p_triangle % print()
```

```
Triangle
Point
  1.00000000       1.00000000
Point
  1.00000000       0.00000000
Point
  1.00000000       1.00000000
```

Recommendations on using OOP in Fortran

- Don’t code directly
  - Draw a sketch on paper
  - Identify object candidates – Element, Solver, PointSet, ElementSet, ElementMaterial, BoundaryCondition
  - Identify object relationships - Element has Nodes - A PointSet has a set of Nodes
  - Eliminate redundant objects
  - Define object attributes
- Don’t create classes every object you can find
- Use classes to encapsulate larger objects
  - A PointSet instead of a Point
- Object-oriented design should make things easier to use – Not more complicated.

Mixed language programming

Combining C++ and Fortran

Mixed language programming

- Fortran is not suited for all programming tasks
- User interface is often implemented in C++
- Instead of spawning the Fortran code as separate process it is possible to link C++ with Fortran code
- Before modern Fortran it was complicated to mix C++ and Fortran due to different calling conventions and ambigous data type mappins
- Fortran 2003 introduced the iso_c_binding module
  - Provides data type mappings for C and Fortran
- Fortran 2003 also introduces the value attribute for passing paramaters by value which is the standard for scalars in C

Calling C++ from Fortran

General flow

- Create an interface header file (.h) and a C++ implementation file (.cpp)
- In the header declare the functions that should be called using the extern "C" clause
- Defined the functions in Fortran using interfaces and the iso_c_binding module

Step 1 – C++ header and implementation sources

```cpp
#ifndef MYCPP_H
#define MYCPP_H

extern "C" {
    void simple();
}

#endif // MYCPP_H
```

mycpp.h

```cpp
#include "mycpp.h"

#include <iostream>
#include <cmath>

void simple()
{
    std::cout << "Hello form C++\n";
}
```

mycpp.cpp

Step 2 – Declare the functions in Fortran

```fortran
use iso_c_binding

implicit none

interface
    subroutine simple() bind(C, name="simple")
    end subroutine simple
end interface
```

Tells the compiler to use C calling conventions

Must be the same name as i the C++ sources

Step 3 – Call the C++ function from Fortran

```fortran
call simple()
```

```
Hello from C++
```

Passing scalar values to C++ function

```cpp
void easy(int a, int b)
{
    std::cout << "a = " << a << " b = " << b << "\n";
}
```

```fortran
interface
    ...
    subroutine easy(a, b) bind(C, name="easy")
        use iso_c_binding
        integer(c_int), value :: a
        integer(c_int), value :: b
    end subroutine
    ...
End interface
```

```fortran
call easy(2, 4)
```

```
a = 2 b = 4
```

Scalar return parameters

```cpp
void no_problem(float a, float b, float* c)
{
    std::cout << "a = " << a << " b = " << b << "\n";
    *c = a + b;
    std::cout << "c = " << *c << "\n";
}
```

```fortran
interface
    ...
    subroutine no_problem(a, b, c) bind(C, name="no_problem")
        use iso_c_binding
        real(c_float), value :: a
        real(c_float), value :: b
        real(c_float), intent(out) :: c
    end subroutine
    ...
End interface
```

```fortran
call no_problem(2.0, 4.0, c1)
```

```
a = 2 b = 4
c = 6
   6.00000000
```

Scalar return parameters

```cpp
void no_problemas(float a, float b, float& c)
{
    std::cout << "a = " << a << " b = " << b << "\n";
    c = a + b;
    std::cout << "c = " << c << "\n";
}
```

```fortran
interface
    ...
    subroutine no_problemas(a, b, c) bind(C, name="no_problemas")
        use iso_c_binding
        real(c_float), value :: a
        real(c_float), value :: b
        real(c_float), intent(out) :: c
    end subroutine
    ...
End interface
```

```fortran
call no_problemas(2.0, 4.0, c2)
```

```
a = 2 b = 4
c = 6
   6.00000000
```

Passing arrays

```
void many_numbers(float* a, float* b, float* c, int n)
{
    for (auto i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
```

```
interface
    ...
    subroutine many_numbers(a, b, c, n) bind(C, name="many_numbers")
        use iso_c_binding
        real(c_float) :: a(*)
        real(c_float) :: b(*)
        real(c_float) :: c(*)
        integer(c_int), value :: n
    end subroutine
    ...
End interface
```

```
real(c_float), allocatable,
dimension(:) :: a, b, c

allocate(a(20), b(20), c(20))

call many_numbers(a, b, c, 20)
```

Functions

```c
double myfunc(double x)
{
    return sin(x);
}
```

```fortran
interface
    ...
    function myfunc(x) result(y) bind(C, name="myfunc")
        use iso_c_binding
        real(c_double), value :: x
        real(c_double) :: y
    end function
    ...
End interface
```

```fortran
print*, myfunc(1.0_c_double)
```