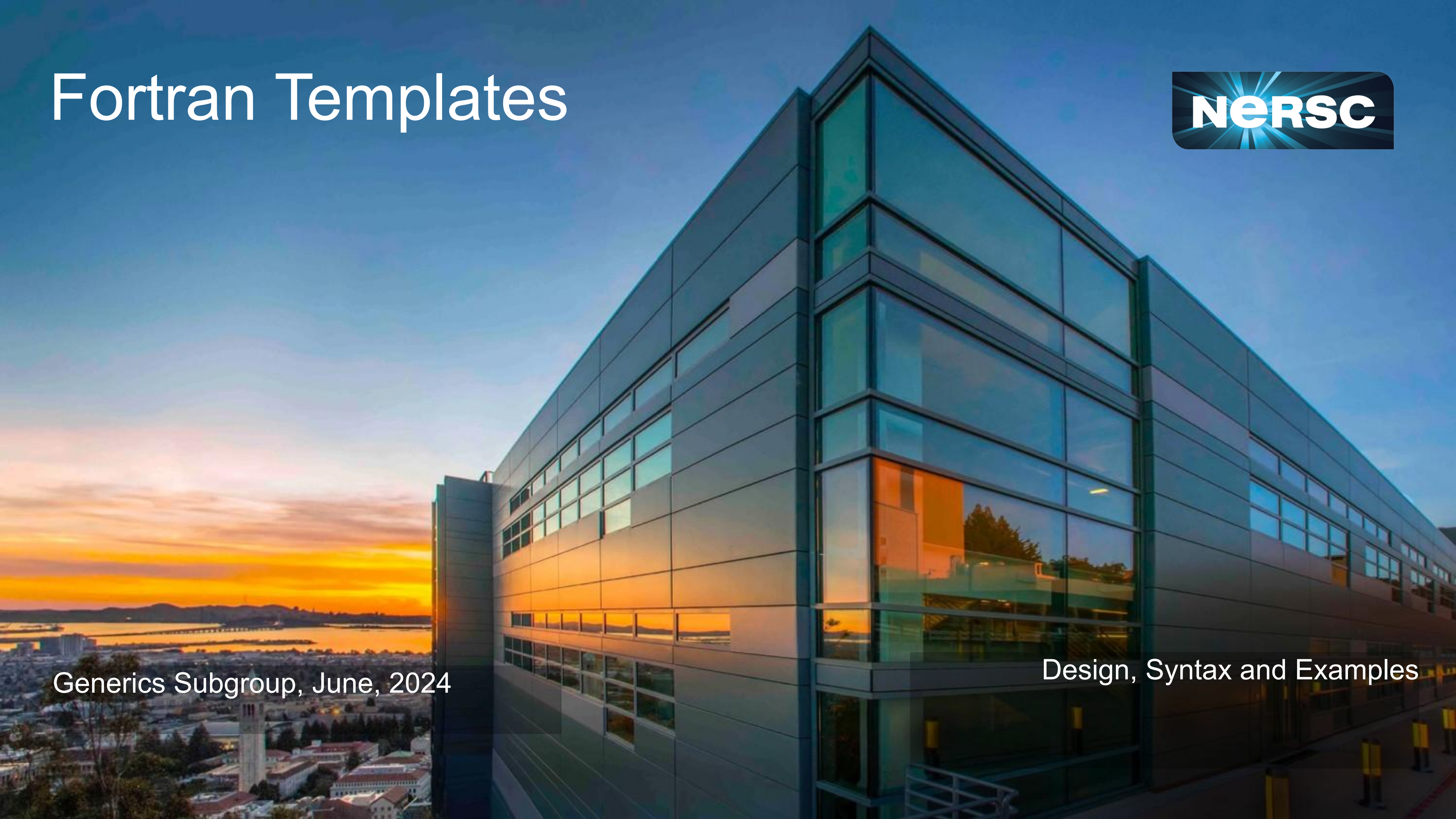
Fortran Templates



Design, Syntax and Examples

Generics Subgroup, June, 2024

Use Cases and Requirements

- Use Cases
  - Algorithms
    - Swap
    - Intrinsics, E.g. findloc, maxval, etc.
    - Sorting and Searching
    - Numeric Algorithms, E.g. Matrix Solver
  - Containers
    - Vector
    - Set
    - Associative Array/Map/Dictionary
- Requirements
  - Named Templates
  - Full type safety (i.e. “strong concepts”)
  - Named “requirements”
  - “Duplicate” types are the same type

Template Design Goals

- Compiler ensures the template is:
  - Self consistent
  - Specifies what makes a valid combination of arguments
  - Easy to write templates that work with derived and intrinsic types
- Template doesn’t dictate spelling of derived type components or type-bound procedures

Some Example Syntax

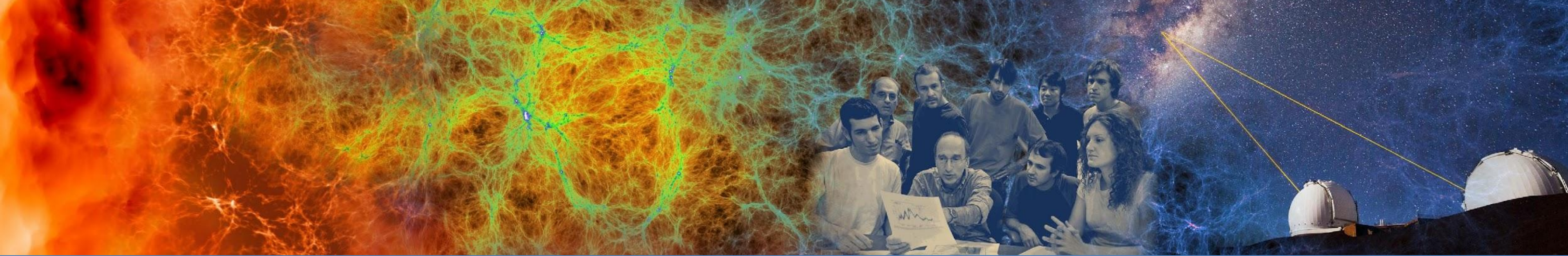
```
requirement fizz{t, buzz}
    type, deferred :: t
    interface
        function buzz(…)
            …
        end function
end requirement

template tmpl{u, f, …}
    requires fizz{u, f}
    …
end template

instantiate tmpl{integer, operator(+), …}, only: a, b => c, …
```

```
function tmpl_func{a, b}(x, y)
  type, deferred :: a
  integer, constant :: b
  type(a), intent(in) :: x, y
  type(a) :: tmpl_func
  ...
end function

print *, tmpl_func{real, 4}(1., 2.)
```



A “Simple” Example

AXPY

```
simple function axpy(a, x, y)
  real, intent(in) :: a
  real, contiguous, intent(in) :: x(:)
  real, intent(in) :: y(size(x))
  real :: axpy(size(x))

  axpy = a*x + y
end function
```

Don’t want to have to duplicate this

Kind Agnostic AXPY

```
simple function axpy{k}(a, x, y)
  integer, constant :: k
  real(k), intent(in) :: a
  real(k), contiguous, intent(in) :: x(:)
  real(k), intent(in) :: y(size(x))
  real(k) :: axpy(size(x))

  axpy = a*x + y
end function
```

```
integer, parameter :: sp = kind(1.0)
integer, parameter :: dp = kind(1.d0)
real(sp) :: a, x(10), y(10)
real(dp) :: da, dx(10), dy(10)
…
print *, axpy{sp}(a, x, y)
print *, axpy{dp}(da, dx, dy)
```

Type Agnostic AXPY

```
requirement bin_op{T, op}
  type, deferred :: T
  interface
    simple elemental function op(x, y)
      type(T), intent(in) :: x, y
      type(T) :: op
    end function
  end interface
end requirement

simple function axpy &
    {T, plus, times} &
    (a, x, y)
  requires bin_op{T, plus}
  requires bin_op{T, times}
  type(T), intent(in) :: a
  type(T), contiguous, intent(in) :: x(:)
  type(T), intent(in) :: y(size(x))
  type(T) :: axpy(size(x))

  axpy = plus(times(a, x), y)
end function
```

```
integer, parameter :: sp = kind(1.0)
integer, parameter :: dp = kind(1.d0)
real(sp) :: a, x(10), y(10)
real(dp) :: da, dx(10), dy(10)
integer :: ia, ix(10), iy(10)
…
print *, axpy &
    {real(sp), operator(+), operator(*)} &
    (a, x, y)
print *, axpy &
    {real(dp), operator(+), operator(*)} &
    (da, dx, dy)
print *, axpy &
    {integer, operator(+), operator(*)} &
    (ia, ix, iy)
```

Mixed Type AXPY

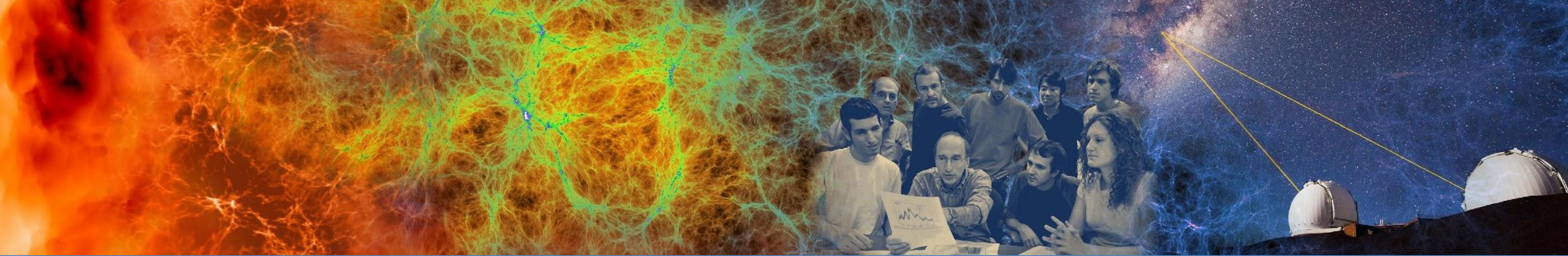
```
requirement bin_op{T, U, V, op}
  type, deferred :: T, U, V
  interface
    simple elemental function op(x, y)
      type(T), intent(in) :: x
      type(U), intent(in) :: y
      type(V) :: op
    end function
  end interface
end requirement
```

```
simple function axpy &
    { a_type, x_type, y_type, &
      times_result_type, plus_result_type, &
      plus, times} &
    (a, x, y)
  requires bin_op{ &
      a_type, x_type, times_result_type, times}
  requires bin_op{ &
      times_result_type, y_type, &
      plus_result_type, plus}
  type(a_type), intent(in) :: a
  type(x_type), contiguous, intent(in) :: x(:)
  type(y_type), intent(in) :: y(size(x))
  type(plus_result_type) :: axpy(size(x))

  axpy = plus(times(a, x), y)
end function
```

```
integer, parameter :: sp = kind(1.0)
integer, parameter :: dp = kind(1.d0)
real(sp) :: a
integer :: x(10)
real(dp) :: y(10)
instantiate axpy{ &
        a_type = real(sp), &
        x_type = integer, &
        y_type = real(dp), &
        times_result_type = real(sp), &
        plus_result_type = real(dp), &
        plus = operator(+), &
        times = operator(*)}

print *, axpy(a, x, y)
```



Examples with “Containers”

Print the things in a container

```
subroutine print_things_in_container( container )
  type(container_type), intent(in) :: container

  type(iterator_type) :: i, iteration_end

  iteration_end = end(container)
  i = begin(container)
  do while (.not. equal(i, iteration_end))
    call print(item(container, i))
    call next(container, i)
  end do
end subroutine
```

Add the necessary deferred-args

```
subroutine print_things_in_container &
    { item_type, container_type, iterator_type, &
      begin, end, equal, next, item, print } &
    ( container )
   …
end subroutine
```

And their declarations

```
type, deferred :: &
    item_type, &
    container_type, &
    iterator_type
interface
  function begin(container)
    type(container_type), intent(in) :: &
        container
    type(iterator_type) :: begin
  end function
  function end(container)
    type(container_type), intent(in) :: &
        container
    type(iterator_type) :: end
  end function
  function equal(lhs, rhs)
    type(iterator_type), intent(in) :: &
        lhs, rhs
    logical :: equal
  end function
subroutine next(container, iterator)
  type(container_type), intent(in) :: &
      container
  type(iterator_type), intent(inout) :: &
      iterator
end subroutine
function item(container, iterator)
  type(container_type), intent(in) :: &
      container
  type(iterator_type), intent(in) :: &
      iterator
  type(item_type) :: item
end function
subroutine print(item)
  type(item_type), intent(in) :: item
end function
end interface
```

And define a vector

```
template vector_tmpl(T)
  type, deferred :: T
  type :: vector
    type(T), allocatable :: items(:)
  end type
contains
  function begin(v)
    …
  end function
  function end(v)
    …
  end function
  subroutine next(v, iterator)
    …
  end subroutine
  function item(v, index)
    …
  end function
end template
```

And then we can use them

```
instantiate vector_tmpl{integer}, only: &
    integer_vector => vector, &
    integer_vector_begin => begin, &
    integer_vector_end => end, &
    integer_vector_next => next, &
    integer_vector_item => item

call print_things_in_container{ &
    item_type = integer, &
    container_type = integer_vector, &
    iterator_type = integer, &
    begin = integer_vector_begin, &
    end = integer_vector_end, &
    equal = operator(==), &
    next = integer_vector_next, &
    item = integer_vector_item, &
    print = print_integer} &
    (integer_vector([1, 2, 3]))
```

```
subroutine print_integer(i)
  integer, intent(in) :: i
  print *, i
end subroutine
```

Or with reals

```
instantiate vector_tmpl{real}, only: &
    real_vector => vector, &
    real_vector_begin => begin, &
    real_vector_end => end, &
    real_vector_next => next, &
    real_vector_item => item

call print_things_in_container{ &
    item_type = real, &
    container_type = real_vector, &
    iterator_type = integer, &
    begin = real_vector_begin, &
    end = real_vector_end, &
    equal = operator(==), &
    next = real_vector_next, &
    item = real_vector_item, &
    print = print_real} &
    (real_vector([1., 2., 3.]))
```

```
subroutine print_real(r)
  real, intent(in) :: r
  print *, r
end subroutine
```

Or with a little bit more work

```
template polymorphic_wrapper_tmpl{T}
    type, abstract, deferred :: T
    type :: wrapper_type
        class(T), allocatable :: item
    end type
end template
```

```
type, abstract :: shape_type
contains
  procedure(to_string_i), deferred :: to_string
end type

abstract interface
  function to_string_i(shape)
    import :: shape_type
    class(shape_type), intent(in) :: shape
    character(len=:), allocatable :: to_string_i
  end function
end interface

instantiate
polymorphic_wrapper_tmpl{shape_type}, only: &
    shape_wrapper_type => wrapper_type
```

It works with polymorphic, heterogeneous lists

```
instantiate vector_tmpl{shape_wrapper_type}, only: &
    shape_vector => vector, &
    shape_vector_begin => begin, &
    shape_vector_end => end, &
    shape_vector_next => next, &
    shape_vector_item => item

call print_things_in_container{ &
    item_type = shape_wrapper_type, &
    container_type = shape_vector, &
    iterator_type = integer, &
    begin = shape_vector_begin, &
    end = shape_vector_end, &
    equal = operator(==), &
    next = shape_vector_next, &
    item = shape_vector_item, &
    print = print_shape_wrapper} &
    (shape_vector([shape_wrapper_type(...), shape_wrapper_type(...), ...]))
```

```
subroutine print_shape_wrapper(shape_wrapper)
  type(shape_wrapper_type), intent(in) :: &
      shape_wrapper
  print *, shape_wrapper%item%to_string()
end subroutine
```