University of Zurich[UZH]

# OOP Fortran: About constructors and practical applications of patterns

Tiziano Müller
tiziano.mueller@chem.uzh.ch

4. July 2020

Dept. of Chemistry, UZH

# Construction of Derived types

# There are no constructors in Fortran[1]

```
type Mytype
   integer :: a
   logical :: init = .false.
end type

[...]

Mytype :: t

! t%init is set to .false
! the value of t%a is undefined
```

- No way to inject logic (other than default initializers) to run at *declaration* time.
- Strict separation between data declaration and code maintained.
- Cf. to C/C++ where you can run code before `int main()`.

[1] at least not in the same sense as in C++ or Python

# What defining a Derived Type gets you

```
type Mytype
   integer :: a
   logical :: init = .false.
end type
```

...automatically gives you:

- an initializer (right-side)
- a copy function
- a vtable

```
function Mytype(a, init) result(self)
   type(Mytype) :: self
   integer, optional :: a
   integer, optional :: init

   if (present(a)) &
      self%a = a

   if (present(init)) then
      self%init = init
   else
      self%init = .false.
   end if
end type
```

# What defining a Derived Type gets you, the lower level

```
__mymod_MOD___copy_mymod_Mytype:
        mov     rax, QWORD PTR [rdi]
        mov     QWORD PTR [rsi], rax
        ret
__mymod_MOD___vtab_mymod_Mytype:
        .long   28885743
        .zero   4
        .quad   8
        .quad   0
        .quad   __mymod_MOD___def_init_mymod_Mytype
        .quad   __mymod_MOD___copy_mymod_Mytype
        .quad   0
        .quad   0
__mymod_MOD___def_init_mymod_Mytype:
        .zero   8
```

https://gcc.godbolt.org/
(using `gfortran-9.3 -O2`)

- But where are the conditionals?
- And how does it look when I call it?

# What calling an initializer does

```
program main
    use mymod
    implicit none
    type(Mytype) :: t
    t = Mytype(42)
    print *, t
end program
```

Build with:

```
gfortran -O2 -S
-fverbose-asm
main.f90
```

```
1  MAIN__:
2  .LFB0:
3      .cfi_startproc
4      subq    $488, %rsp  #,
5      .cfi_def_cfa_offset 496
6  # main.f90:6:     print *, t
7      movabsq $25769803904, %rax  #, tmp93
8  # main.f90:5:     t = Mytype(42)
9      movq    $42, t.3505(%rip)   #, MEM[(struct mytype *)&t]
10 # main.f90:6:     print *, t
11     movq    %rsp, %rdi  #,
12     movq    %rax, (%rsp)    # tmp93, MEM[(integer(kind=4) *)&dt_parm.1]
13     movq    $.LC0, 8(%rsp)  #, dt_parm.1.common.filename
14     movl    $6, 16(%rsp)    #, dt_parm.1.common.line
15     call    _gfortran_st_write  #
16 ...
```

→No function call to be found!

# Does this apply to custom initializers?

```
module testmod
implicit none

type Mytype
    integer :: a
    logical :: init
end type

interface Mytype
    module procedure :: &
        Mytype_constructor
end interface

contains
function Mytype_constructor(a, init) &
    result(self)
! as shown above
end function
end module
```

```
1   AIN__:
2   .LFB0:
3           .cfi_startproc
4           subq        $504, %rsp        #,
5           .cfi_def_cfa_offset 512
6   # main.f90:5:     t = Mytype(42)
7           xorl        %esi, %esi        #
8           movl        $.LC0, %edi        #,
9           call        __mymod_MOD_mytype_constructor        #
10  # main.f90:6:     print *, t
11          leaq        16(%rsp), %rdi        #, tmp98
12  # main.f90:5:     t = Mytype(42)
13          movq        %rax, 8(%rsp)        # tmp87, t
14  # main.f90:6:     print *, t
15          movabsq        $25769803904, %rax        #, tmp97
16  ...
```

→Unfortunately not!

# Summary

- For *data classes* (all attributes public): use *default* initializers. Everything will be inlined.
- You need custom "constructors" if attributes are not public.
- Do not call custom "constructors" (*interface*) from performance-critical sections. You get a function call and a copy.
- Using a type-bound procedure for initialization will give function call (and use the vtable), but no copy.
- Using Link-time-optimization (LTO, `-flto` with GCC) inlines custom initialization functions.
- Compiler behavior may vary.

# Design Patterns in Practice

# What are Design and Software Architecture Patterns?

Salginatobel Bridge, by Rama, CC BY-SA 2.0 fr

*Software Design Patterns* are general, reusable solutions to commonly occurring problems within software engineering.
*Software Architecture Patterns* are common design patterns occurring in the organisation and interaction of software components.

**Literature**

"GoF" (1995): Design Patterns: Elements of Reusable OO Software.
Fowler (2002): Patterns of Enterprise Application Architecture.

# What do you gain by knowing about Design Patterns

- Scientific programming often straight forward and levels of indirection may affect performance.
- Growing software becomes quickly unmaintainable.
- Design patterns provide uniform way of documenting common solutions to problems.
- Compilers are awesome and can optimize away the levels of indirection introduce by modularization/encapsulation.
- Do not try to put everything in patterns.
  Learn to recognize them while writing code.

**Literature**

Sutter/Alexandrescu (2004): C++ Coding Standards.
Hunt/Thomas (1999): The Pragmatic Programmer: Your Journey To Mastery.

# 2 Examples

## Iterator

Provide access to elements of an aggregate object (sequentually) without exposing the underlying representation.

- *AKA Cursor*
- Remember: For striding Fortran has array slice syntax.
- Useful to hide complex access patterns.
- Avoid code duplication in loops.

`https://github.com/cp2k/dbcsr/`

## Command

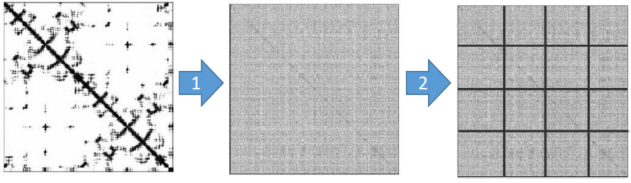
Encapsulate a possibly parametrized request (or operation) as an object. User of object doesn't have to know about the parameters.

- *AKA Action, Transaction*
- As seen in Arjen Markus' functional examples
- Alternative to bare function pointers

`https://github.com/cp2k/cp2k/`

# The DBCSR Sparse Matrix Library

- **Co-developed** with the CP2K electronic structure software package
- **DBCSR**: **D**istributed **B**lock-**C**ompressed **S**parse **R**ow format
- **Static Decomposition**: distribution over a two-dimensional grid of processes




- **Parallel Performance**: Cannon's algorithm
- **Single Node Perfomance**: Specialized libraries for handling local multiplication of small blocks ($m, n, k \leq 64$): *libxsmm* (CPU) & *libsmm_acc* (GPU).

→ **Talks** by Patrick Seewald and Maximilian Ambroise!

# Iterator pattern usage in DBCSR

```
1  TYPE(dbcsr_iterator) :: iter
2
3  ! Loop over blocks of a blocked matrix
4  CALL dbcsr_iterator_start(iter, matrix)
5
6  DO WHILE (dbcsr_iterator_blocks_left(iter))
7        CALL dbcsr_iterator_next_block(iter, row, col, blk, row_size, col_size)
8        ! block pointed by blk at (row, col) position
9        ! and (row_size x col_size) elements
10       ...
11 END DO
12
13 CALL dbcsr_iterator_stop(iter)
```

- Hides synchronization/communication.
- OpenMP-aware!
- Could use type-bound procedures for same effect.
- See src/mm/dbcsr_mm_cannon.F, src/block/dbcsr_iterator_operations.F.

# Iterator pattern exercise: Write file line reader

- Very simple use case:
  - Pass file unit to iterator *start*.
  - Do most of the logic in the *end*/*is_done* function.
  - *get*/*next* is simple getter.
- Can be extended in various directions:
  - Behavioral: Strip characters
  - Behavioral: Filter lines
  - Behavioral: Handle line continuation characters
  - Implementation: use `ADVANCE='no'`
  - Implementation: handle arbitrary line length
  - Implementation: wrap `mmap` for (almost) zero-copy.

# Command pattern usage in CP2K

```
TYPE, EXTENDS(eri_type_eri_element_func) :: eri2array
   INTEGER(C_INT), POINTER :: coords(:)
   REAL(C_DOUBLE), POINTER :: values(:)
   INTEGER                 :: idx = 1
CONTAINS
   PROCEDURE :: func => eri2array_func
END TYPE
...
CALL active_space_env%eri%eri_foreach(1, eri2array(buf_coords, buf_values))
```

```
   TYPE, EXTENDS(eri_type_eri_element_func) :: eri_fcidump_print
      INTEGER :: unit_nr
   CONTAINS
      PROCEDURE :: func => eri_fcidump_print_func
   END TYPE
...
! Print integrals: ERI
CALL active_space_env%eri%eri_foreach(1, eri_fcidump_print(iw))
```

# Command pattern usage in CP2K: The function object

```
TYPE, ABSTRACT :: eri_type_eri_element_func
CONTAINS
   PROCEDURE(eri_type_eri_element_func_interface), DEFERRED :: func
END TYPE eri_type_eri_element_func

ABSTRACT INTERFACE
   LOGICAL FUNCTION eri_type_eri_element_func_interface(this, i, j, k, l, val)
      IMPORT :: eri_type_eri_element_func, dp
      CLASS(eri_type_eri_element_func), INTENT(inout) :: this
      INTEGER, INTENT(in)                             :: i, j, k, l
      REAL(KIND=dp), INTENT(in)                       :: val
   END FUNCTION eri_type_eri_element_func_interface
END INTERFACE
```

See src/qs_active_space_types.F, src/start/libcp2k.F, src/qs_active_space_methods.F for the *foreach* and function implementations.

# What did we gain?

- Shared data preparation and data access encapsulated in *foreach*.
- *foreach* loop wrapper is completely parameter- and format-agnostic.
- Possibility of advanced data access patterns.
- But, need Link-time-optimization (LTO) to optimize calls further.

# Command pattern exercise: Create transformers

- Write a base class with a *transform* function taking a string and returning a string
- Implement (at the beginning unparametrized) *lowercase* and *uppercase* function objects, doing the respective operation of the string.
- Possible extensions of the exercise:
  - Parametrization: act only on every *n*th word or character
  - Implement a *capitalization* transformation