MAMMASMIAS

LAMMPS Architecture in FORTRAN

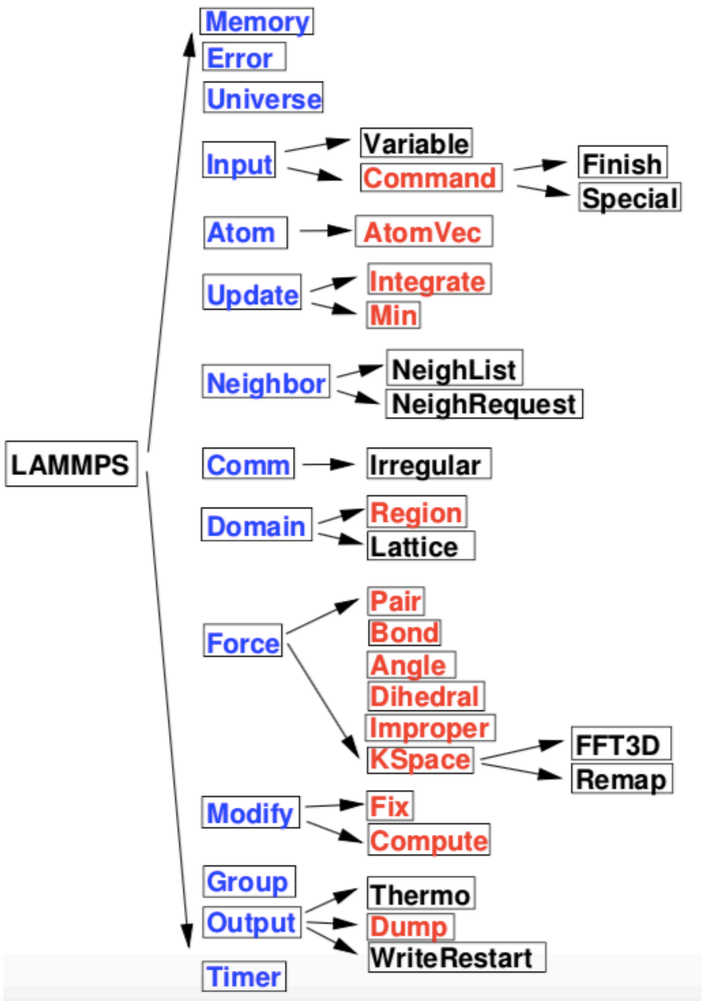
Recipe for a FLAMMPS

Nicolas Salles, CNR-IOM Democritos

LAMMPS Architecture in Fortran

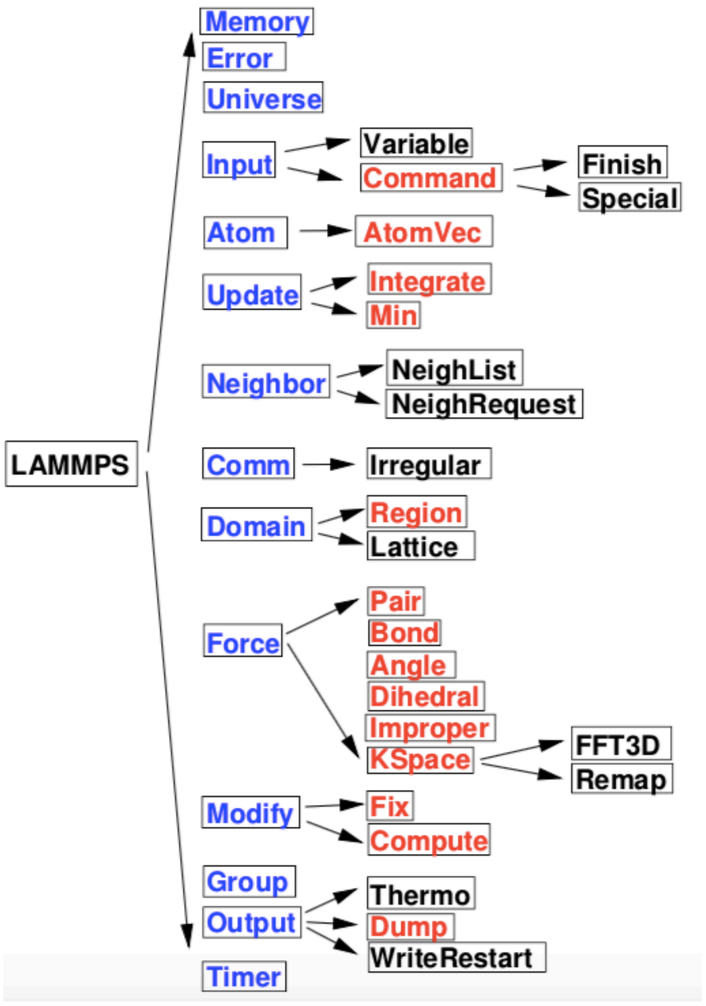
- Interest of **LAMMPS Architecture**
  - An exemple of modularity
- How to build the architecture of LAMMPS in **Object Oriented Fortran**
  - **Horizontal connection** through the `class`
- **User-Friendly** Aspect: Include automatically new derived classes at compilation stage
  - The user never touch other parts of the code than his own

LAMMPS Architecture



- Each element/part of the simulation, Atomic information, neighbor list, memory management, force computation, … has its own `class`
- The simulation is organised by extensions of `Integrate` or `Min class`

LAMMPS Architecture



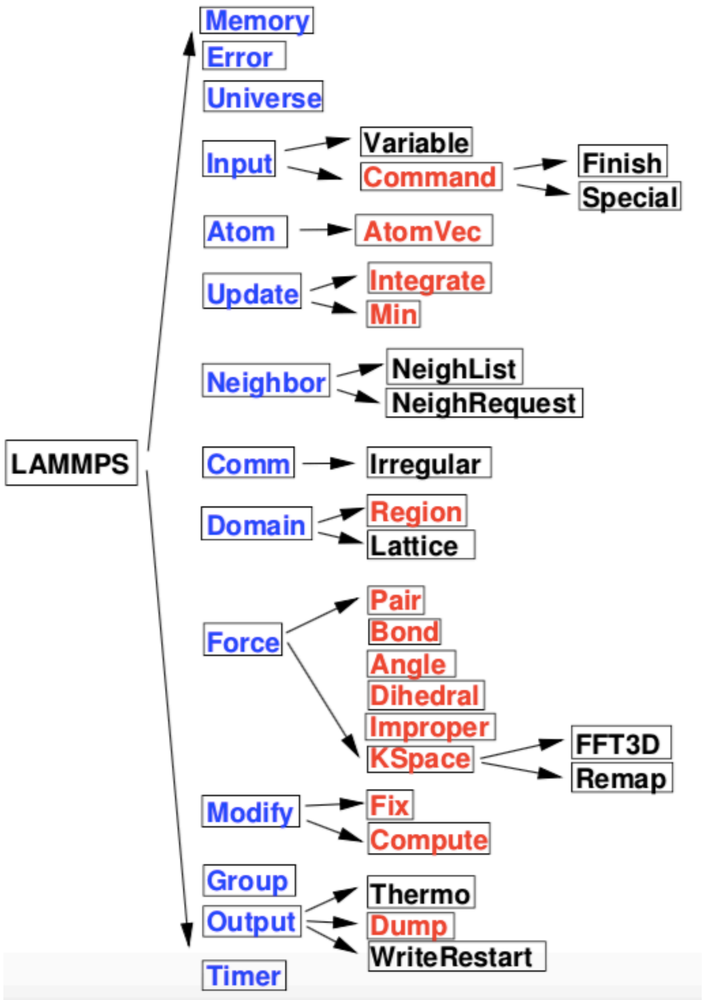
- Each element/part of the simulation, Atomic information, neighbor list, memory management, force computation, … has its own `class`

- The simulation is organised by extensions of `Integrate` or `Min class`

  → `Class Verlet : public Integrate`

```
Void Verlet::run( int n ){
  […]
  for (int i = 0; i < n; i++) {
    […]
    ntimestep = ++update->ntimestep;
    // initial time integration
    modify->initial_integrate(vflag);
    modify->post_integrate();
    // neighbor list rebuild
    if( neighbor->decide() ){
      modify->pre_neighbor();
      neighbor->build(1);
      modify->post_neighbor();
    }
    // force computations
    modify->pre_force(vflag);
    force->pair->compute(eflag,vflag);
    modify->post_force(vflag);
    modify->final_integrate();
    modify->end_of_step();

    // all output
    if (ntimestep == output->next)output->write(ntimestep);
  }
}
```

LAMMPS Architecture



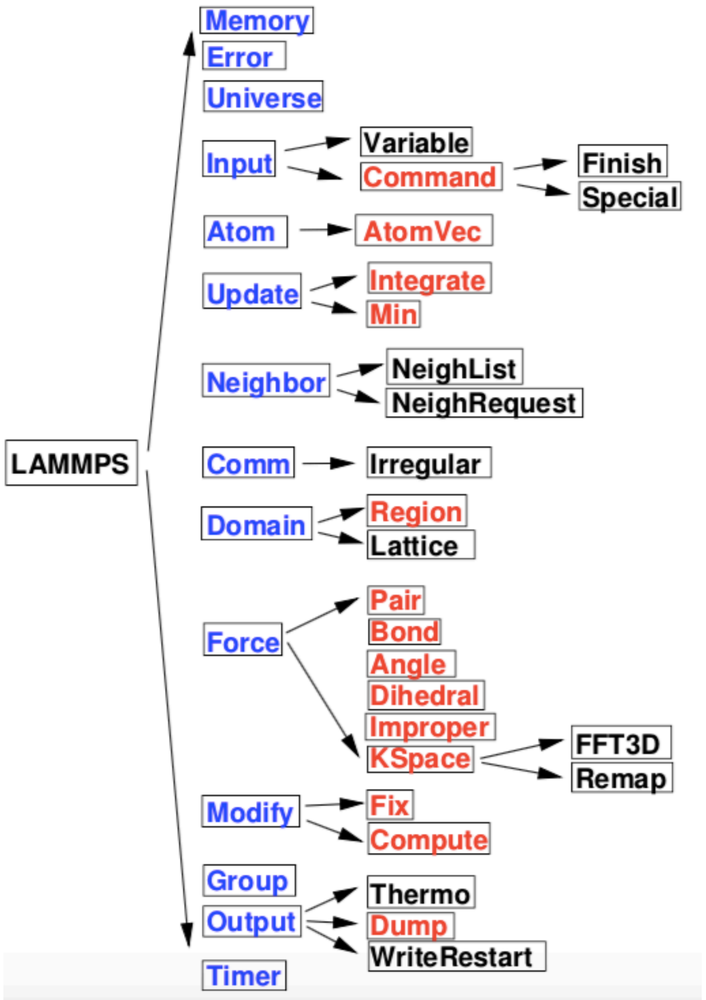
- Each element/part of the simulation, Atomic information, neighbor list, memory management, force computation, … has its own `class`
- The simulation is organised by extensions of `Integrate` or `Min class`
  - `Class Verlet : public Integrate`

```
Void Verlet::run( int n ){
  […]
  for (int i = 0; i < n; i++) {
    […]
    ntimestep = ++update->ntimestep;
    // initial time integration
    modify->initial_integrate(vflag);
    modify->post_integrate();
    // neighbor list rebuild
    if( neighbor->decide() ){
      modify->pre_neighbor();
      neighbor->build(1);
      modify->post_neighbor();
    }
    // force computations
    modify->pre_force(vflag);
    force->pair->compute(eflag,vflag);
    modify->post_force(vflag);
    modify->final_integrate();
    modify->end_of_step();

    // all output
    if (ntimestep == output->next)output->write(ntimestep);
  }
}
```

Forces computed by `Force->pair class`

2 step integration of velocity-verlet algorithm carried out by `Modify->Fix Class`

LAMMPS Architecture



- Each element/part of the simulation, Atomic information, neighbor list, memory management, force computation, … has its own `class`
- The simulation is organised by extensions of `Integrate` or `Min class`
  - `Class Verlet : public Integrate`

Verlet has access to all classes

```
Void Verlet::run( int n ){
  […]
  for (int i = 0; i < n; i++) {
    […]
    ntimestep = ++update->ntimestep;
    // initial time integration
    modify->initial_integrate(vflag);
    modify->post_integrate();
    // neighbor list rebuild
    if( neighbor->decide() ){
      modify->pre_neighbor();
      neighbor->build(1);
      modify->post_neighbor();
    }
    // force computations
    modify->pre_force(vflag);
    force->pair->compute(eflag,vflag);
    modify->post_force(vflag);
    modify->final_integrate();
    modify->end_of_step();

    // all output
    if (ntimestep == output->next)output->write(ntimestep);
  }
}
```

Call to `Modify` routines for the intervention from various `Modify->Fix` or `Modify->Compute`

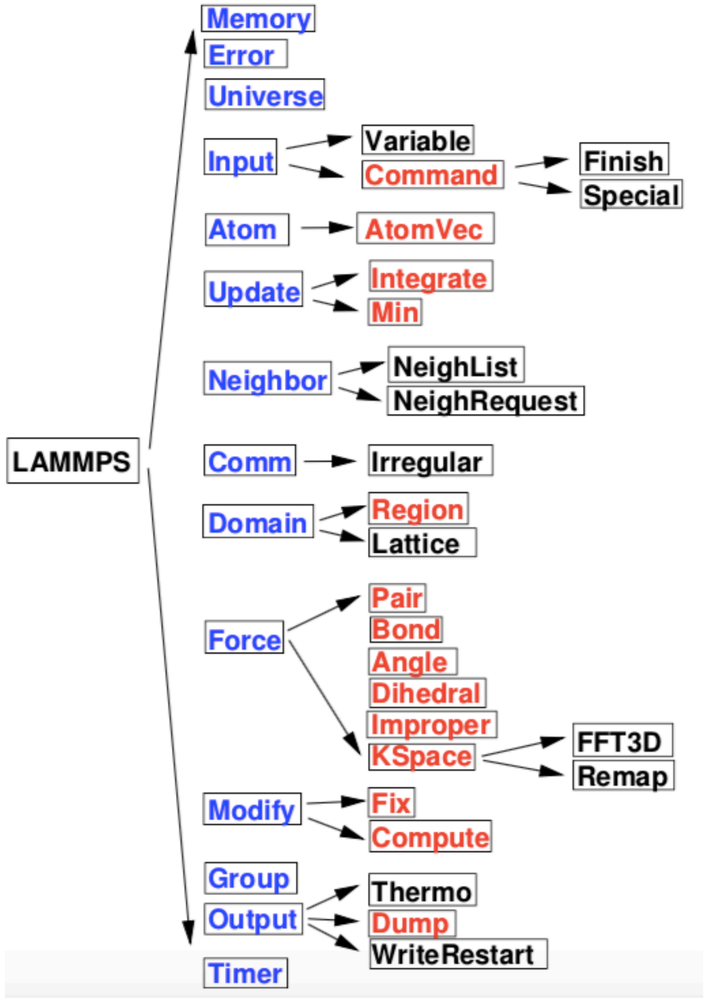
Forces computed by `Force->pair class`

2 step integration of velocity-verlet algorithm carried out by `Modify->Fix Class`

LAMMPS Architecture

- LAMMPS: Molecular Dynamic Software written in C++
  - Each element of the the simulation, Atomic information, neighbor list, Memory management, force computation, … has its own `class`
  - The `class` LAMMPS contains all **main classes**
  - **main classes** might contain other `classes` (**action classes**)
  - Each `class` has access to all **main classes**

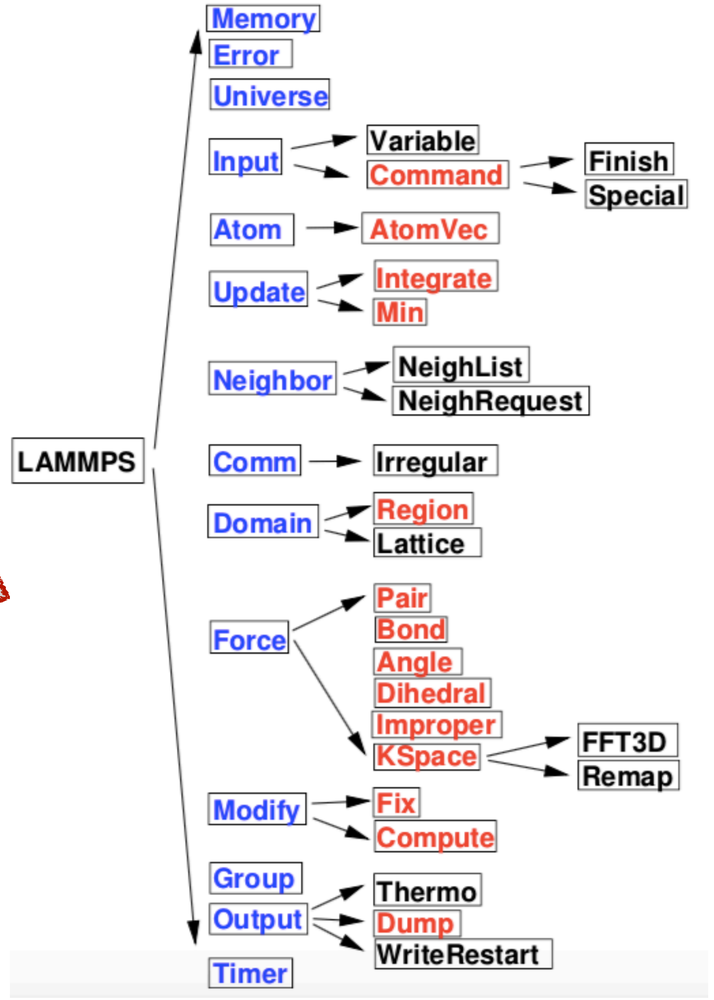
- **action classes** are built/activated by the read input script
- All **action classes** are extendable by the user
- The user-defined `class` is automatically recognised at compilation stage



LAMMPS Architecture

- LAMMPS: Molecular Dynamic Software written in C++
  - Each element of the the simulation, Atomic information, neighbor list, Memory management, force computation, … has its own `class`
  - The `class` LAMMPS contains all **main classes**
  - **main classes** might contain other `classes` (**action classes**)
  - Each `class` has access to all **main classes**

Architecture



- **action classes** are built/activated by the read input script
- All **action classes** are extendable by the user
- The user-defined `class` is automatically recognised at compilation stage

LAMMPS Architecture

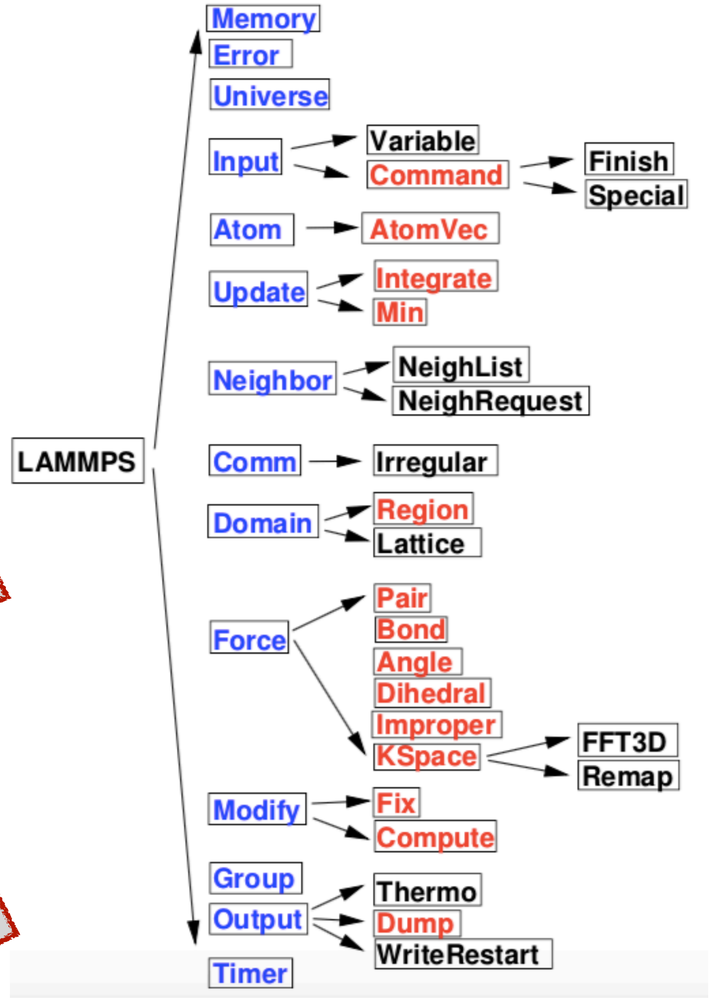
- LAMMPS: Molecular Dynamic Software written in C++

  - Each element of the the simulation, Atomic information, neighbor list, Memory management, force computation, … has its own `class`
  - The `class` LAMMPS contains all **main classes**
  - **main classes** might contain other `classes` (**action classes**)
  - Each `class` has access to all **main classes**

Architecture

- **action classes** are built/activated by the read input script
- All **action classes** are extendable by the user
- The user-defined `class` is automatically recognised at compilation stage

User Friendly



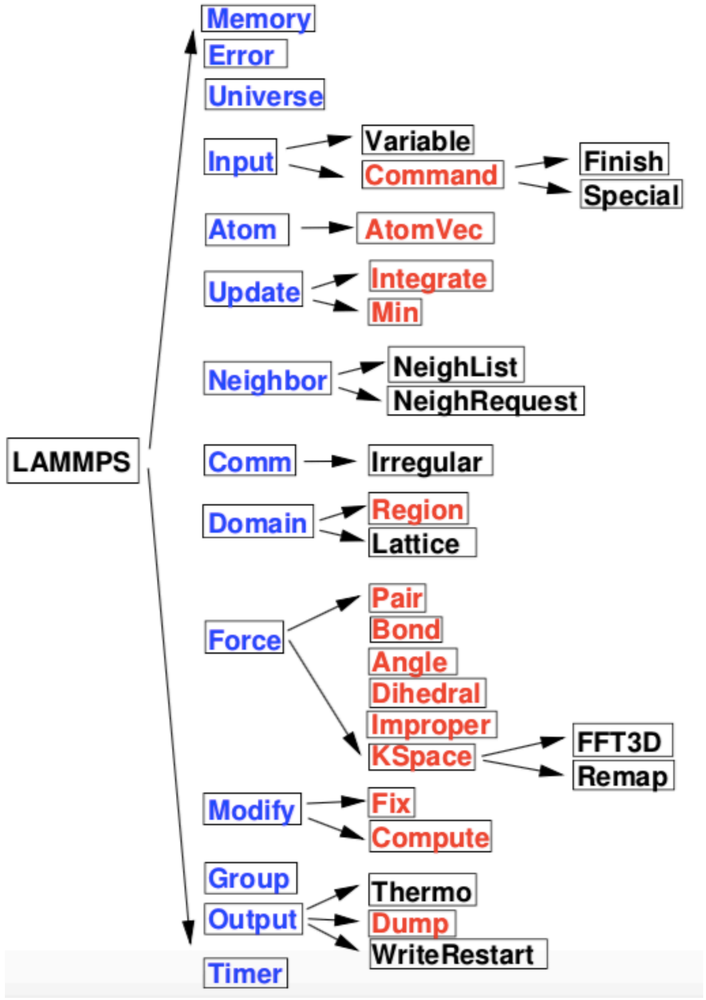
Architecture

LAMMPS execution

```
int main(int argc, char **argv)
{
  MPI_Init(&argc,&argv);

  […]
  LAMMPS *lammps = new LAMMPS( argc, argv, MPI_COMM_WORLD );
  lammps->input->file();
  delete lammps;

  […]
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Allocate **main classes**

Allocate **action classes**

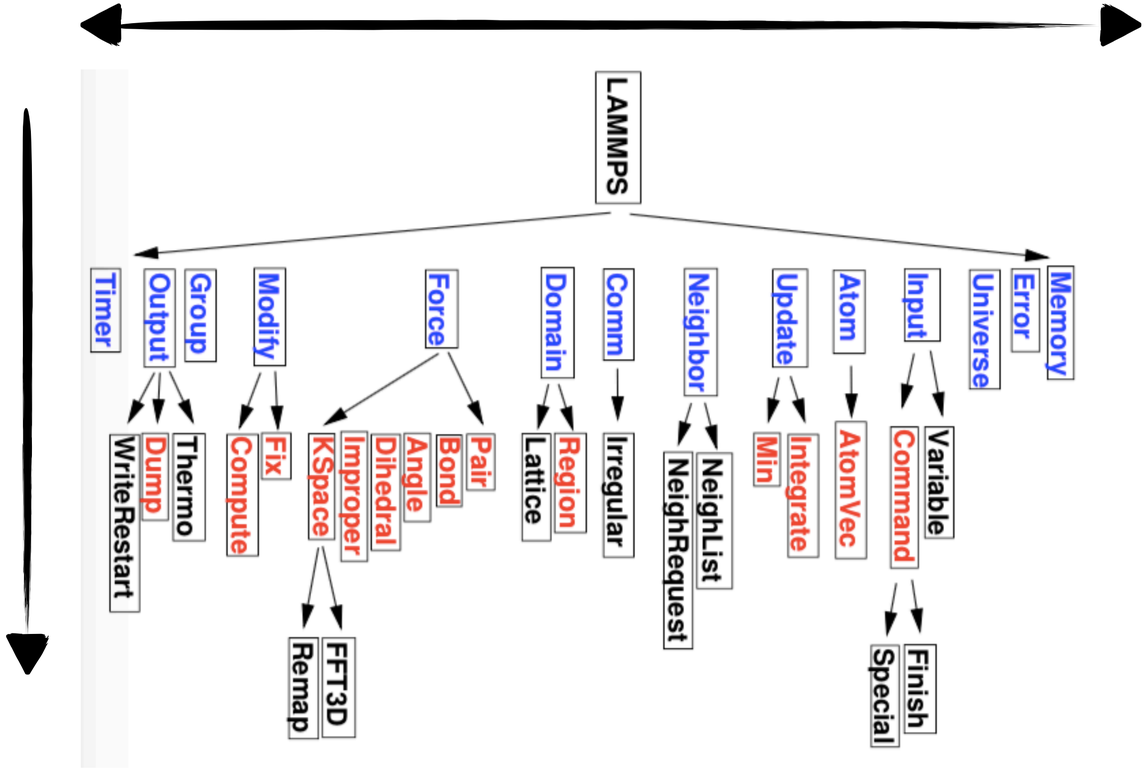


- The simulation happen inside the `class lammps`
- The simulation is customised by the input script through `action classes`

LAMMPS: Horizontal Connection

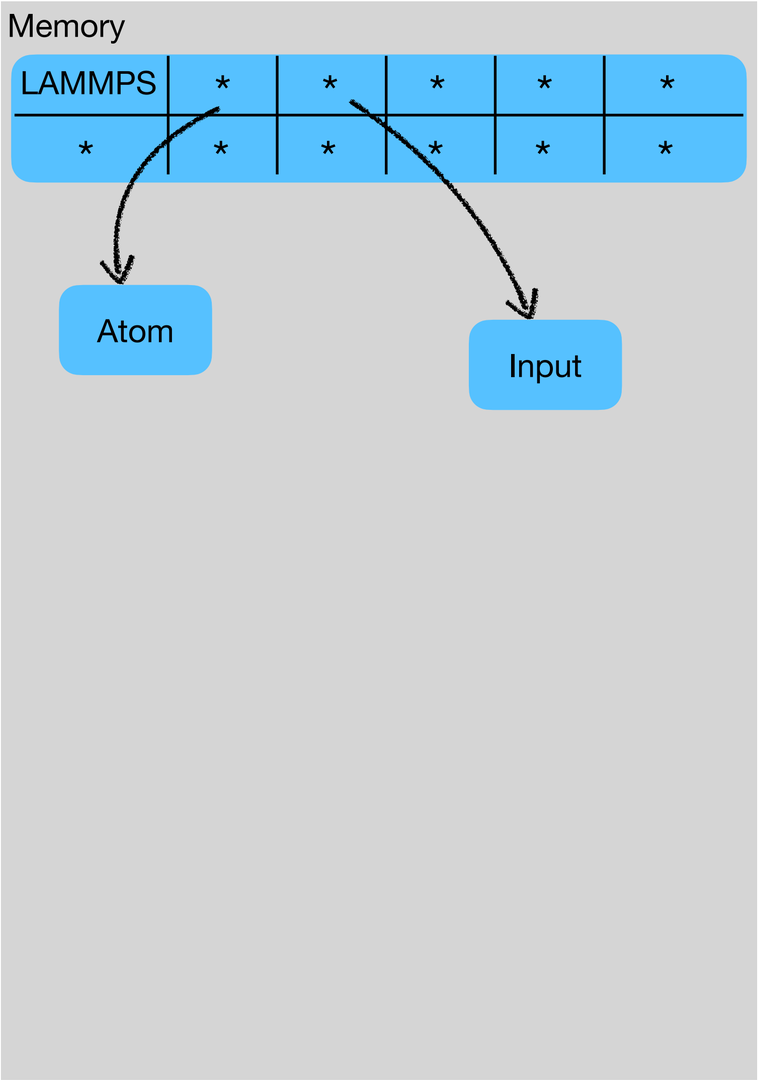
Horizontal Connection:
all `classes` have access
to all **`main classes`**

Encapsulation



LAMMPS: Horizontal Connection

```
class LAMMPS {
 public:

  class Memory *memory;
  class Error *error;
  class Universe *universe;
  class Input *input;

  class Atom *atom;
  class Update *update;
  class Neighbor *neighbor;
  class Comm *comm;
  class Domain *domain;

  class Modify *modify;
  class Group *group;
  class Output *output;
  class Timer *timer;
  […]
}
```



- When the `class input` read the input script, how it can communicate the command to other `class`

LAMMPS: Horizontal Connection

- Idea: Define a twin pointer of class LAMMPS

```
class LAMMPS {
 public:

  class Memory *memory;
  class Error *error;
  class Universe *universe;
  class Input *input;

  class Atom *atom;
  class Update *update;
  class Neighbor *neighbor;
  class Comm *comm;
  class Domain *domain;

  class Modify *modify;
  class Group *group;
  class Output *output;
  class Timer *timer;
  […]
}
```

```
class Pointers {
  […]
 protected:
  LAMMPS *lmp;
  Memory *&memory;
  Error *&error;
  Universe *&universe;
  Input *&input;

  Atom *&atom;
  Update *&update;
  Neighbor *&neighbor;
  Comm *&comm;
  Domain *&domain;
  Force *&force;
  Modify *&modify;
  Group *&group;
  Output *&output;
  Timer *&timer;
  […]
}
```

The twin pointer of LAMMPS

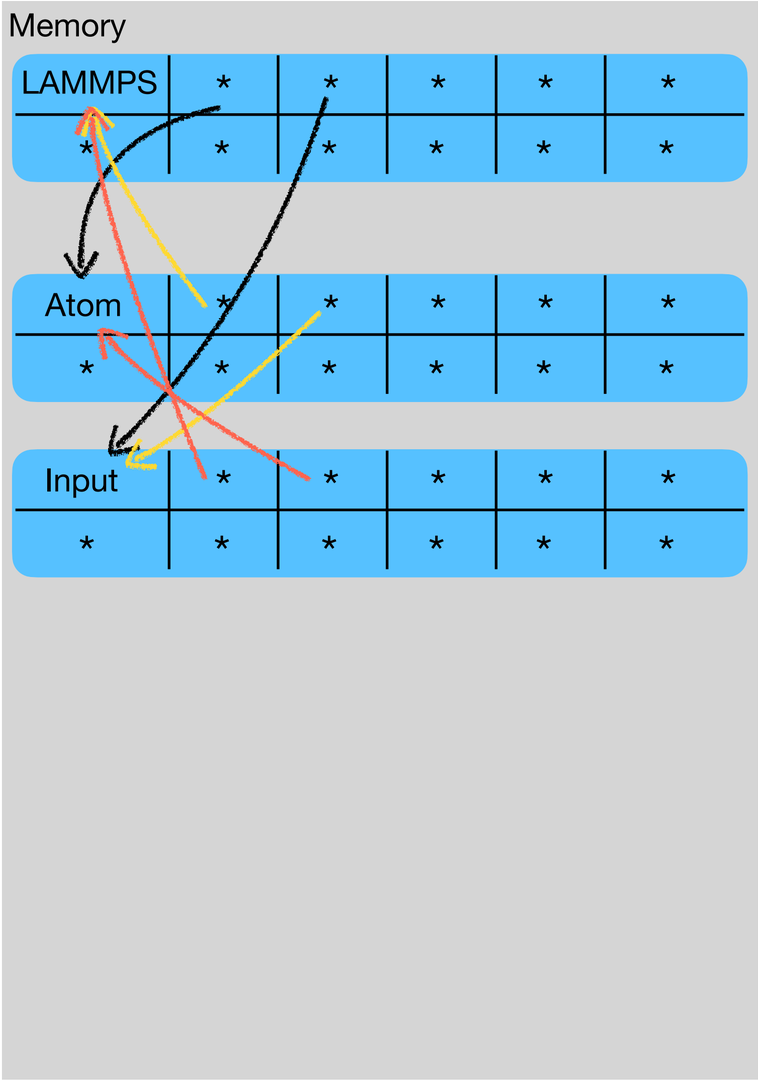
- All `main_class` are defined has extension of `class pointers`

```
class Atom : protected Pointers {
  […]
}
```

- Link the `pointers class` at the constructor level

```
Atom::Atom( LAMMPS *lmp ): Pointers( lmp ){
  […]
}
```

LAMMPS: Horizontal Connection



LAMMPS <—–> Atom <——> Input <——>....

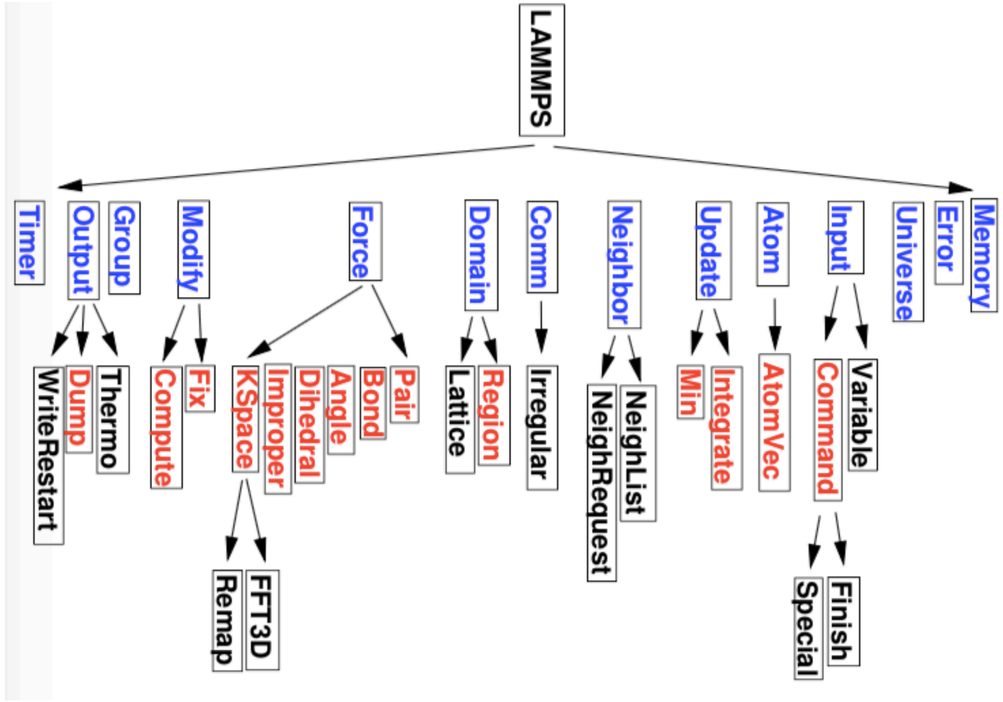
- Builds a multi-linked list between the `main classes`

LAMMPS architecture in Fortran?

- Class encapsulation in Fortran is straightforward
- The horizontal connection is less

Horizontal Connection:
all `classes` have access
to all **`main classes`**

Encapsulation



OOP Tools in Fortran

Some essential tools for the OOP in Fortran

- **Derived type**: equivalent to the `class` in C++
- **Module**: equivalent to the header file (.h) in C++
- **Submodule**: equivalent to the file .cpp in C++

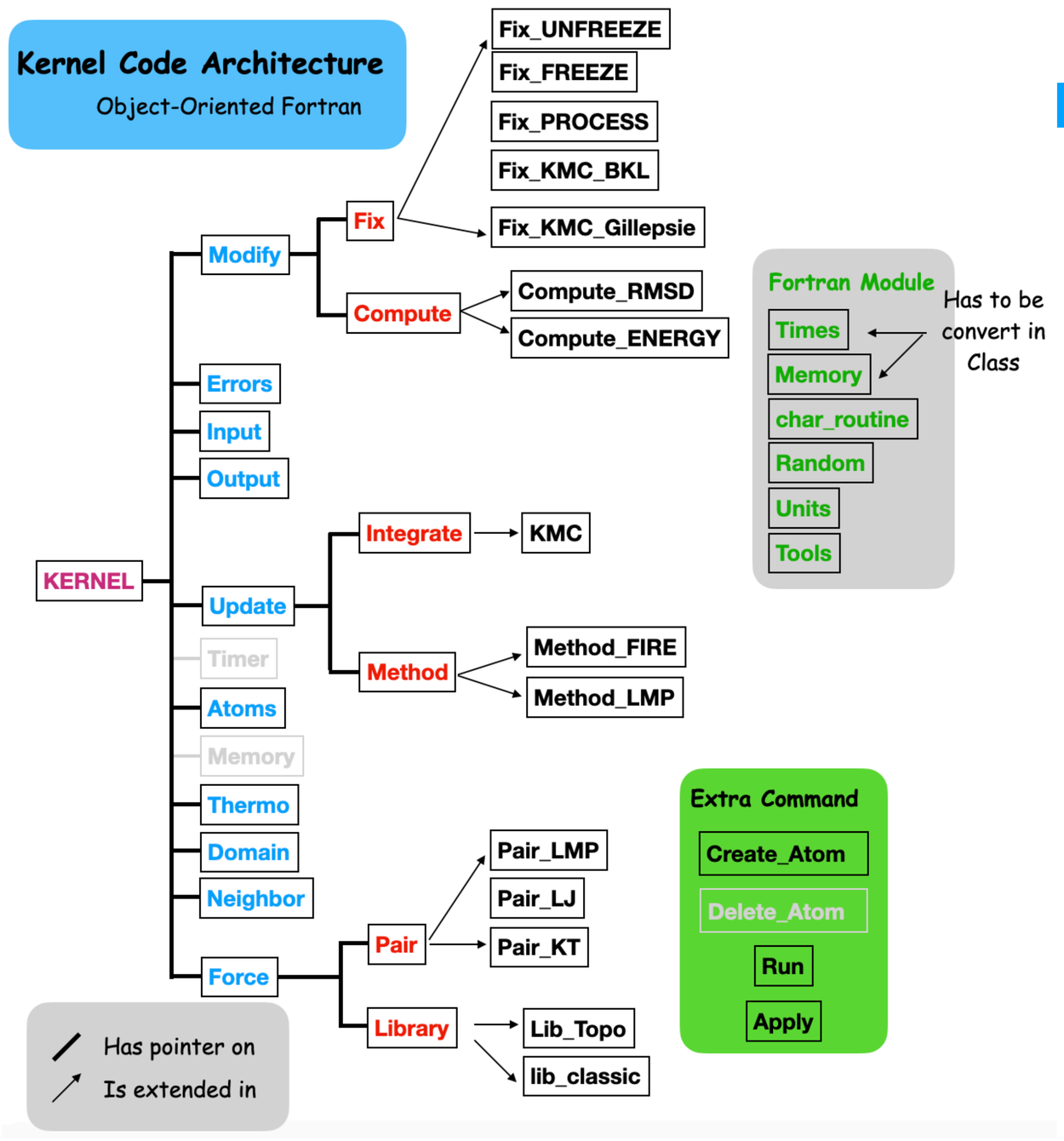
```
Module header_atom
  Type atom
    variable declaration
   Contains
    Procedure :: routines, …
  End type

  Interface atom
    Module procedure :: atom_constructor
  End interface

  Interface
    Module subroutine <prototype>
    Module function <prototype>
  End interface
End Module header_atom
```

Some rules in Fortran

- `header modules` contain the declaration of all the **elements** of the type
  - → Have to be compiled before their use
- `Submodule` contains the functions and subroutines declared in the header
  - → are compiled after the compilation of all header modules
  - → Dependencies to other `headers` are declared (use) in `submodule`

```
submodule( header_atom ) cpp_atom
  Use other_header
 contains
  module subroutine <defintion>
  module function <definition>
End submodule cpp_atom
```

LAMMPS in Fortran = Kernel




```fortran
Type :: kernel
  type( atom ),     pointer :: atoms
  type( domain ),   pointer :: domains
  type( error ),    pointer :: errors
  type( force ),    pointer :: forces
  type( input ),    pointer :: inputs
  type( modify ),   pointer :: modifys
  type( neighbor ), pointer :: neighbors
  type( output ),   pointer :: outputs
  type( thermo ),   pointer :: thermos
  type( update ),   pointer :: updates
End type kernel
```

Fortran: Twin Pointer Class

```fortran
Type :: kernel
  type( atom ),     pointer :: atoms
  type( domain ),   pointer :: domains
  type( error ),    pointer :: errors
  type( force ),    pointer :: forces
  type( input ),    pointer :: inputs
  type( modify ),   pointer :: modifys
  type( neighbor ), pointer :: neighbors
  type( output ),   pointer :: outputs
  type( thermo ),   pointer :: thermos
  type( update ),   pointer :: updates
End type kernel
```

The twin pointer of kernel

```fortran
type :: pointers
  class(*), pointer :: atoms
  class(*), pointer :: domains
  class(*), pointer :: errors
  class(*), pointer :: forces
  class(*), pointer :: inputs
  class(*), pointer :: modifys
  class(*), pointer :: neighbors
  class(*), pointer :: outputs
  class(*), pointer :: thermos
  class(*), pointer :: updates

  class(*), pointer :: kernels
  […]
end type pointers
```

The type `pointers` contains polymorph pointers `class(*)`

- The target can be anything
- Need to confirm the type of the target at each use

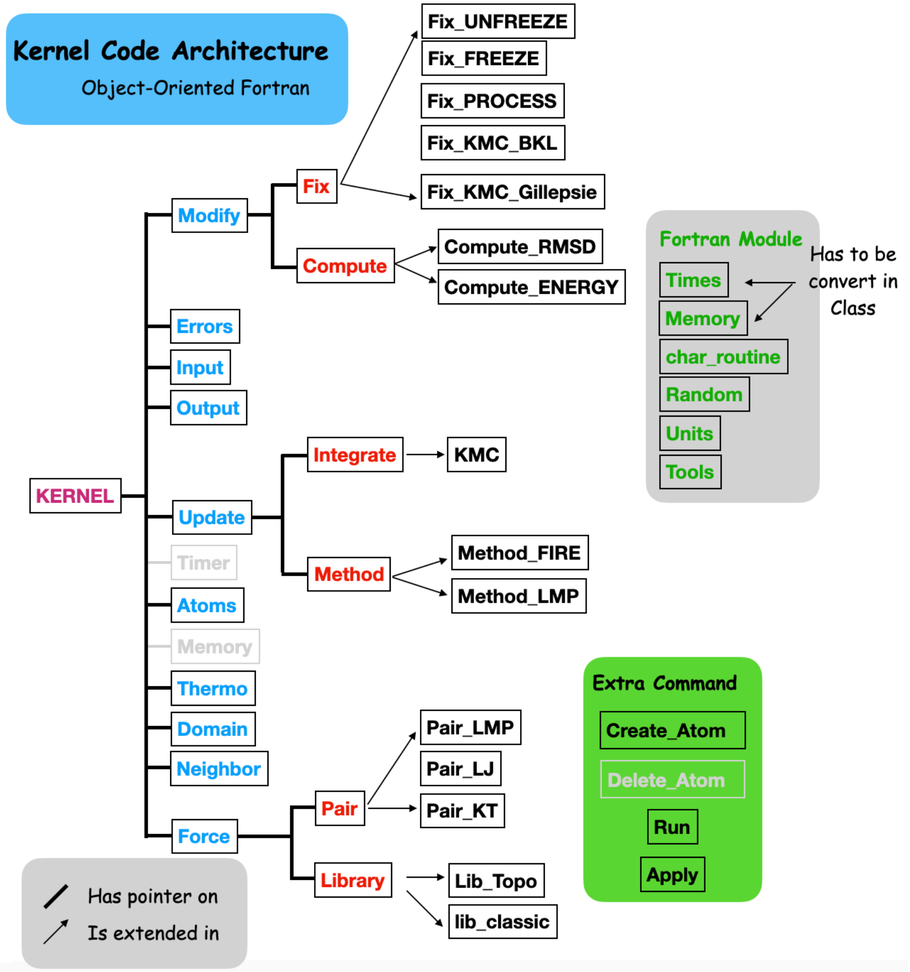
```fortran
Select type( target )
  Type is( mytype )  ! Is the type
  Class is( mytype ) ! Can be the type or a derived type
End select
```

```fortran
module function pointers_constructor( ker )result( ptr )
  class(*), intent( in ) :: ker
  type( pointers ) :: ptr

  select type( ker )
    type is( kernel )
      ptr% inputs    => ker% inputs
      ptr% updates   => ker% updates
      ptr% outputs   => ker% outputs
      ptr% modifys   => ker% modifys
      ptr% errors    => ker% errors
      ptr% forces    => ker% forces
      ptr% atoms     => ker% atoms
      ptr% domains   => ker% domains
      ptr% neighbors => ker% neighbors
      ptr% thermos   => ker% thermos
      ptr% kernels   => ker
  end select
end function pointer_constructor
```

Kernel constructor

- `FOOP,` one step as in C++?

```fortran
module function kernel_constructor()result( this )
  Implicit none
  type( kernel ), pointer :: this
  [...]
  allocate( kernel::this )

  ! create main_type of kernel
  this% atoms   => atom( this )
  this% updates => update( this )
  this% errors  => error( this )
  [...]
end function kernel_constrcutor
```



In fortran the link alone do not result in horizontal connection because the other `main classes` are not build yet

Kernel constructor

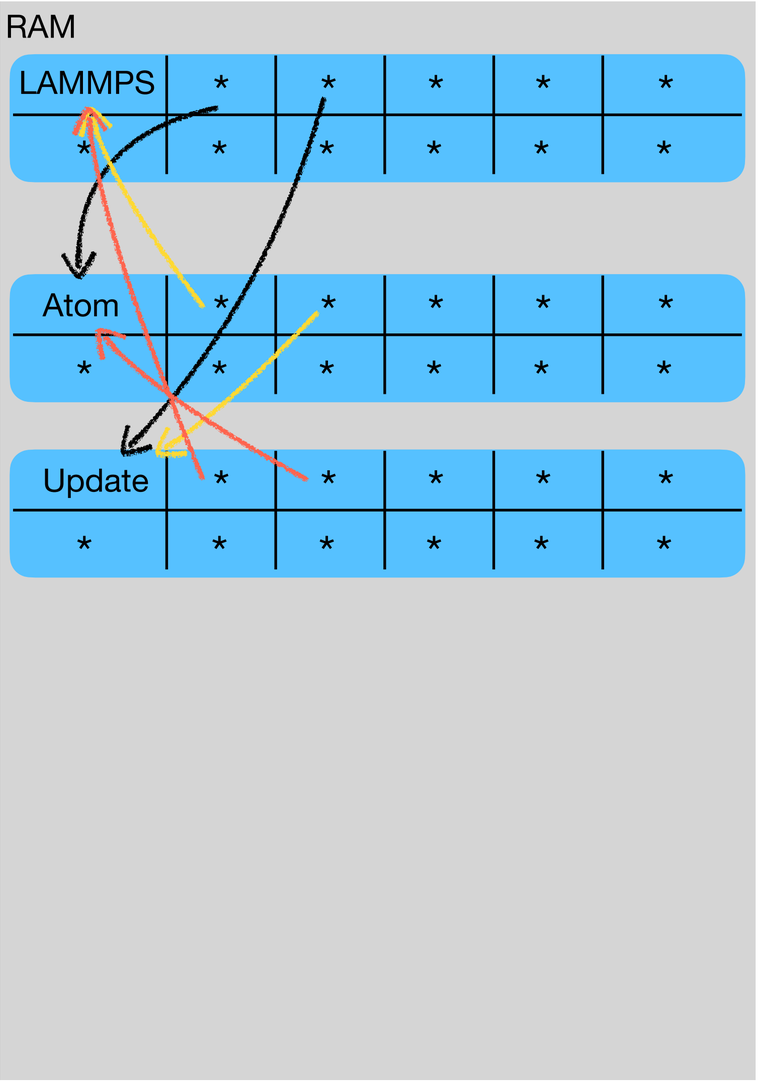
- `FOOP /= C++`: The link to class pointers requires two steps

```
module function kernel_constructor()result( this )
  use header_pointers
  Implicit none
  type( kernel ), pointer :: this
  [...]
  allocate( kernel::this )

  ! create main_type of kernel
  this% atoms   => atom()
  this% updates => update()
  this% errors  => error()
  [...]
  ! Link each main_type with each other
  This% atoms% pointers   = pointers( this )
  This% updates% pointers = pointers( this )
  This% errors% pointers  = pointers( this )
  [...]
end function kernel_constructor
```

Main class building

Horizontal connection

The link of pointers type has to be done after the building of main classes



Fortran: Class Definition

```
Type :: kernel
  type( atom ),      pointer :: atoms
  type( domain ),    pointer :: domains
  type( error ),     pointer :: errors
  type( force ),     pointer :: forces
  type( input ),     pointer :: inputs
  type( modify ),    pointer :: modifys
  type( neighbor ), pointer :: neighbors
  type( output ),    pointer :: outputs
  type( thermo ),    pointer :: thermos
  type( update ),    pointer :: updates
End type kernel
```

```
type :: pointers
  class(*), pointer :: atoms
  class(*), pointer :: domains
  class(*), pointer :: errors
  class(*), pointer :: forces
  class(*), pointer :: inputs
  class(*), pointer :: modifys
  class(*), pointer :: neighbors
  class(*), pointer :: outputs
  class(*), pointer :: thermos
  class(*), pointer :: updates

  class(*), pointer :: kernels
  […]
end type pointers
```

Type declaration in header module

```
Module header_atom
  […]
  type, extend( pointers ) :: atom
    […]
  end type atom
  Interface atom
    Procedure :: atom_constructor
  End interface
  Interface
    module function atom_constructor()result( this )
      type( atom ), pointer :: this
    end function atom_constructor
  End interface
end module header_atom
```

Type definition in submodule

```
Submodule( header_atom ) cpp_atom
 contains
  module function atom_constructor()result( this )
    type( atom ), pointer :: this

    allocate( atom::this )
    […]
    ! Initialize the variable
  end function atom_constructor
  […]
end submodule cpp_atom
```

Horizontal connection: Access to other class

- Now all class has `class(*), pointer` on each **`main class`**
  - To use these polymorph pointers, Fortran imposes to identify them

```fortran
type :: pointers
  class(*), pointer :: atoms
  class(*), pointer :: domains
  class(*), pointer :: errors
  class(*), pointer :: forces
  class(*), pointer :: inputs
  class(*), pointer :: modifys
  class(*), pointer :: neighbors
  class(*), pointer :: outputs
  class(*), pointer :: thermos
  class(*), pointer :: updates

  class(*), pointer :: kernels
  […]
end type pointers
```

- Identification Procedure in submodule:

```fortran
module subroutine something_on_atom( self )
  Use header_error
  class( atom ), intent( inout ) :: self

  type( error ), pointer :: err

  select type( self% errors )
    type is( error ); err => self% errors
  end select

  Call err% error_("There is problem!!")

end subroutine something_on_atom
```

Horizontal connection: Access to other class

- Now all class has `class(*), pointer` on each **`main class`**
  - To use these polymorph pointers, Fortran imposes to identify them

```fortran
type :: pointers
  class(*), pointer :: atoms
  class(*), pointer :: domains
  class(*), pointer :: errors
  class(*), pointer :: forces
  class(*), pointer :: inputs
  class(*), pointer :: modifys
  class(*), pointer :: neighbors
  class(*), pointer :: outputs
  class(*), pointer :: thermos
  class(*), pointer :: updates

  class(*), pointer :: kernels
  […]
end type pointers
```

- Identification Procedure in submodule:

```fortran
module subroutine something_on_atom( self )
  Use header_error
  class( atom ), intent( inout ) :: self

  type( error ), pointer :: err

  select type( self% errors )
    type is( error ); err => self% errors
  end select

  Call err% error_("There is problem!!")

end subroutine something_on_atom
```

Each time you want to access to another `main class`

Horizontal connection: Access to other class

- Now all class has `class(*), pointer` on each **`main class`**
  - To use these polymorph pointers, Fortran imposes to identify them

```fortran
type :: pointers
  class(*), pointer :: atoms
  class(*), pointer :: domains
  class(*), pointer :: errors
  class(*), pointer :: forces
  class(*), pointer :: inputs
  class(*), pointer :: modifys
  class(*), pointer :: neighbors
  class(*), pointer :: outputs
  class(*), pointer :: thermos
  class(*), pointer :: updates

  class(*), pointer :: kernels
  […]
end type pointers
```

- Define an identification procedure for each **`main class`**

```fortran
module subroutine link_error( this, selector )
  use header_pointers
  class( pointers ),     intent( in ) :: this
  type( error ), pointer, intent( inout ) :: selector
  select type( this% errors )
    type is( error ); selector => this% errors
  end select
end subroutine link_error
```

  - Define generic name

```fortran
interface link_class
  module procedure :: link_atom
  module procedure :: link_input
  module procedure :: link_error
  […]
end interface
```

Subroutine and interface `link_*` can be defined in header module and submodule `kernel`

Horizontal connection: Access to other class

- Now all class has `class(*), pointer` on each **`main class`**
  - To use these polymorph pointers, Fortran imposes to identify them

```fortran
type :: pointers
  class(*), pointer :: atoms
  class(*), pointer :: domains
  class(*), pointer :: errors
  class(*), pointer :: forces
  class(*), pointer :: inputs
  class(*), pointer :: modifys
  class(*), pointer :: neighbors
  class(*), pointer :: outputs
  class(*), pointer :: thermos
  class(*), pointer :: updates

  class(*), pointer :: kernels
  […]
end type pointers
```

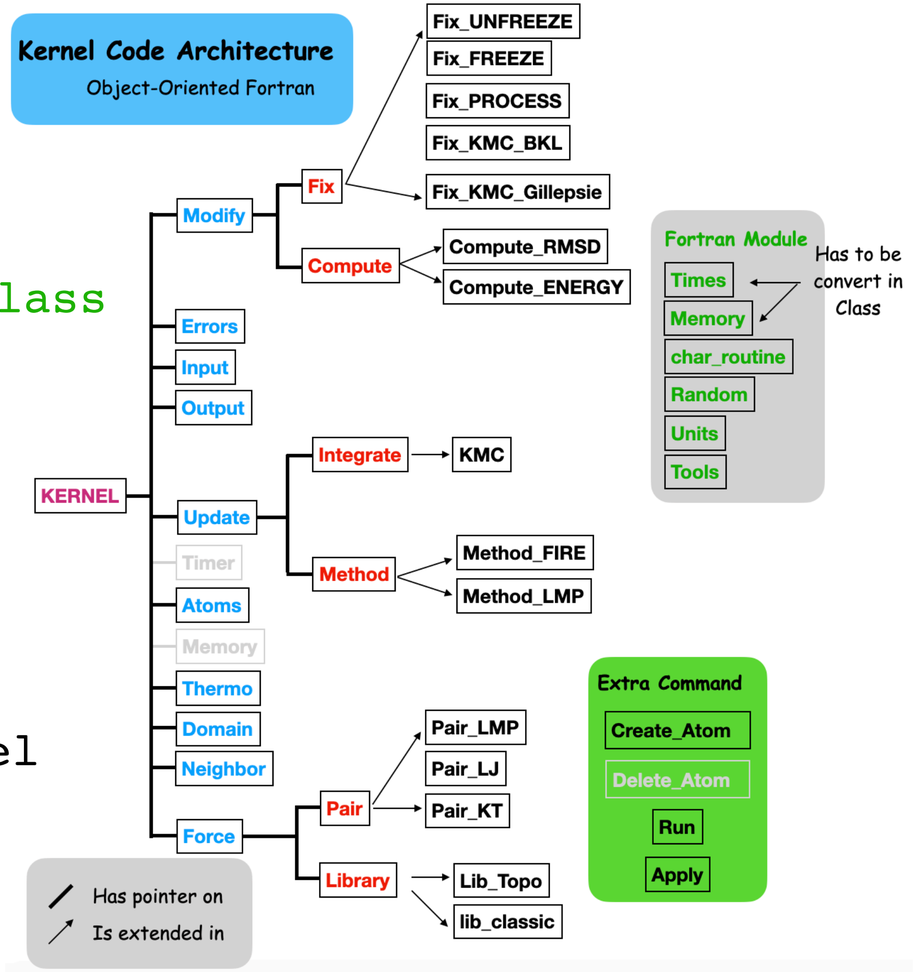
- Identification Procedure in submodule:

```fortran
module subroutine read_script( self )
  use header_kernel
  class( input ), intent( inout ) :: self

  type( error ), pointer :: err

  select type( self% errors )
    type is( error ); err => self% errors
  end select
  Call link_class( self, err )

  Call err% error_("There is problem!!")

end subroutine read_script
```

OOPFortran: Horizontal Connection

- In fortran the **horizontal connection** between the classes is obtained
  - Define `type pointers` which is a twin pointer of the `kernel class`
    - Each element is `class(*), pointer` on `main class`
  - All `main classes` are extensions of the `type pointers`
  - The building of `type` kernel requires 2 steps:
    - `main_class` constructor/allocation
    - Link `main_class%pointers` to element of `class kernel`
  - Inside `types`, the access to other `main classes` requires an identification and `link procedure:`

```fortran
Select type( target )
  Type is( mytype )  ! Is the type
  Class is( mytype ) ! Can be the type or a derived type
End select
```

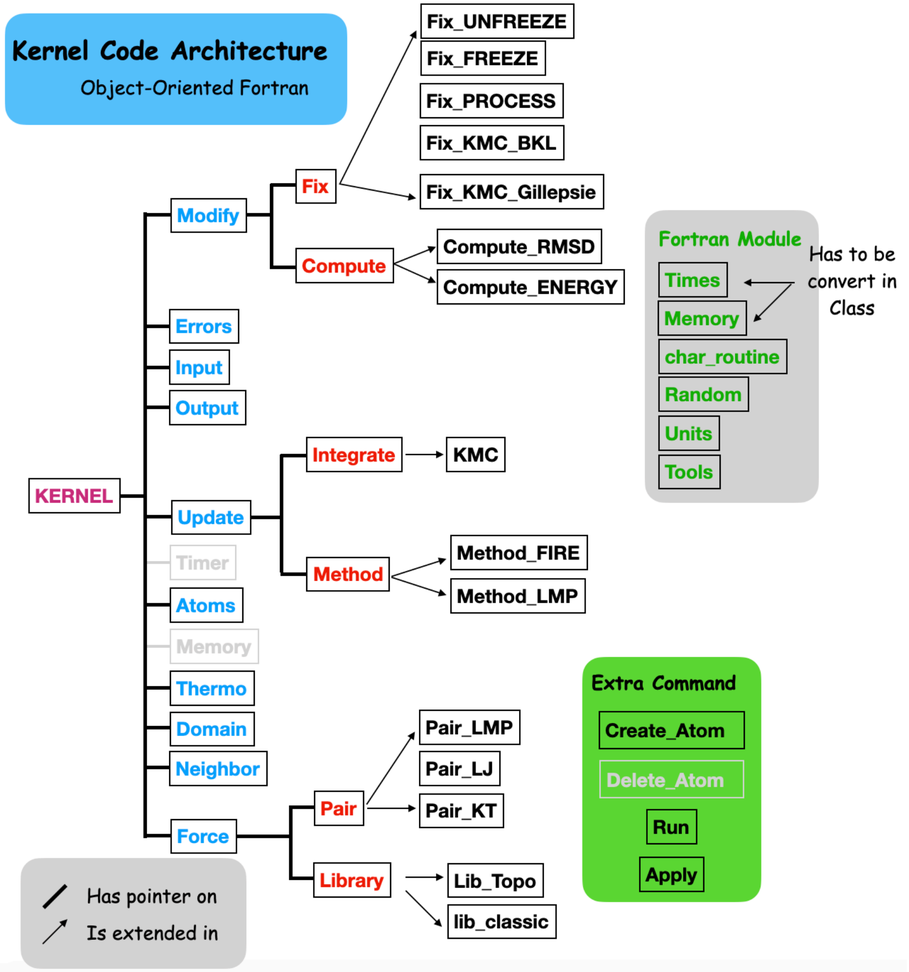


KERNEL execution

```fortran
program main
  Use header_kernel
  Implicit none

  type( kernel ), pointer :: simulation

  Simulation => kernel()
  Call simulation% inputs% read_input()

  if( associated(simulation) ) &
    deallocate(simulation)
End program main
```

Allocate **main classes**

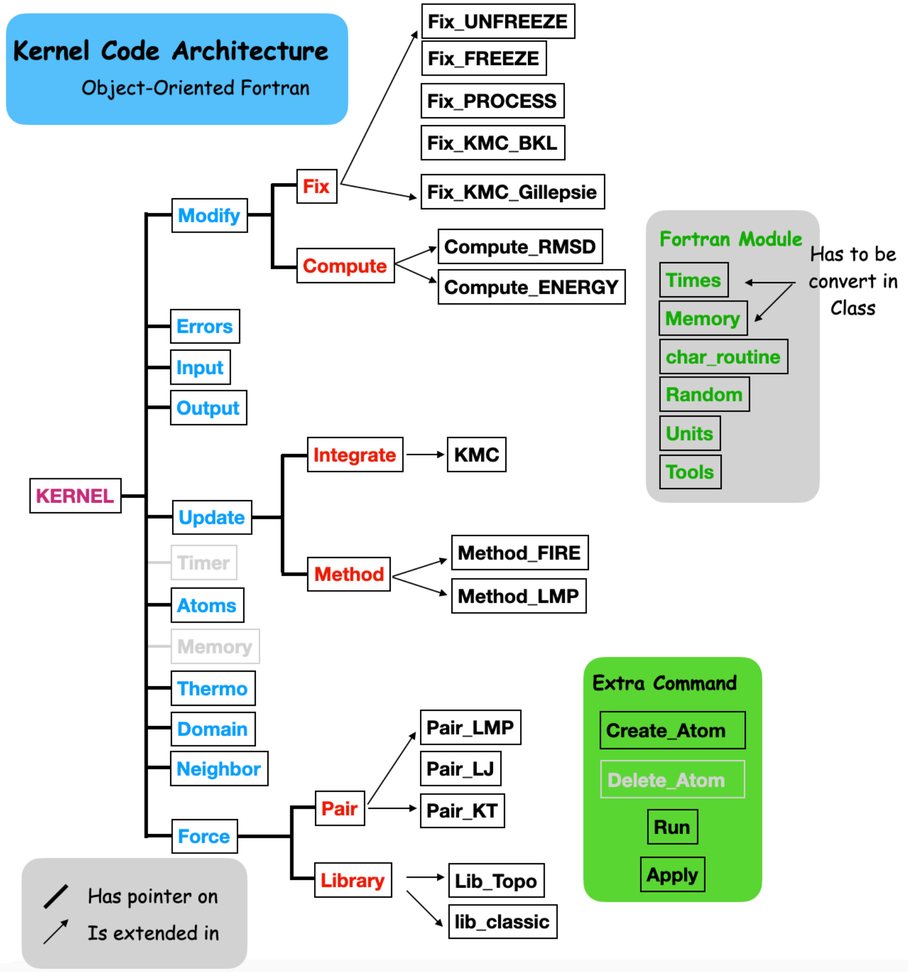
Allocate **action classes**



- The simulation happen inside the `type kernel`
- The simulation is customised by the input script through `action classes`

Interaction between the `class`

- 3 `main class` contains `action class`
- The `action class` (`abstract`) are extendable by the users



Interaction between the `class`

- 3 `main class` contains `action class`
- The `action class` (`abstract`) are extendable by the users
- `Fix`/`compute` are used to modify/compute some properties: position, temperature, pressure, …

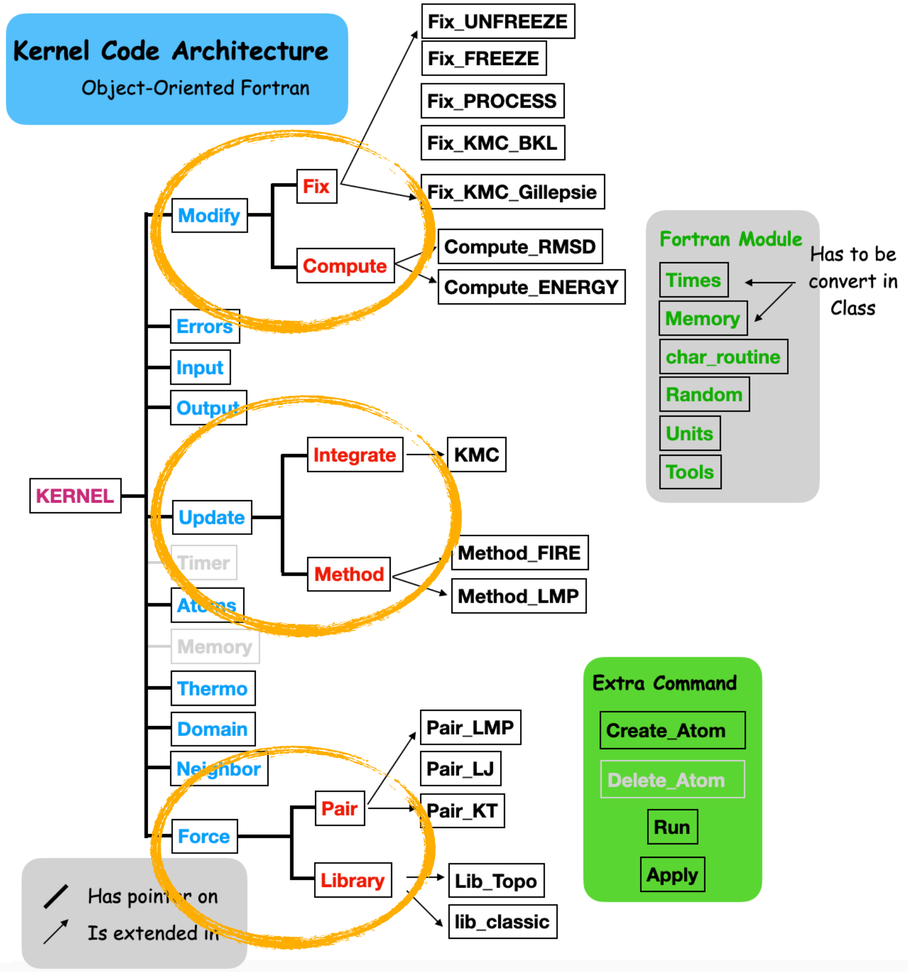
  Answer to keyword in input script: `fix/compute name args`

- `Integrate`/`method` are used to integrate the system/ apply an algorithm chose: kMC, Verlet/FIRE, Elastic, …

  Answer to keyword in input script:
  `integrate_style/method_style name`

- `Pair`/`Library` are compute properties needed for the integrate/method loaded. Basically the pair compute the energy force for verlet integrate, library compute the possible event in the system

  `pair_style/library_style name`



Interaction between the `class`

- 3 `main class` contains `action class`
- The `action class` (`abstract`) are extendable by the users

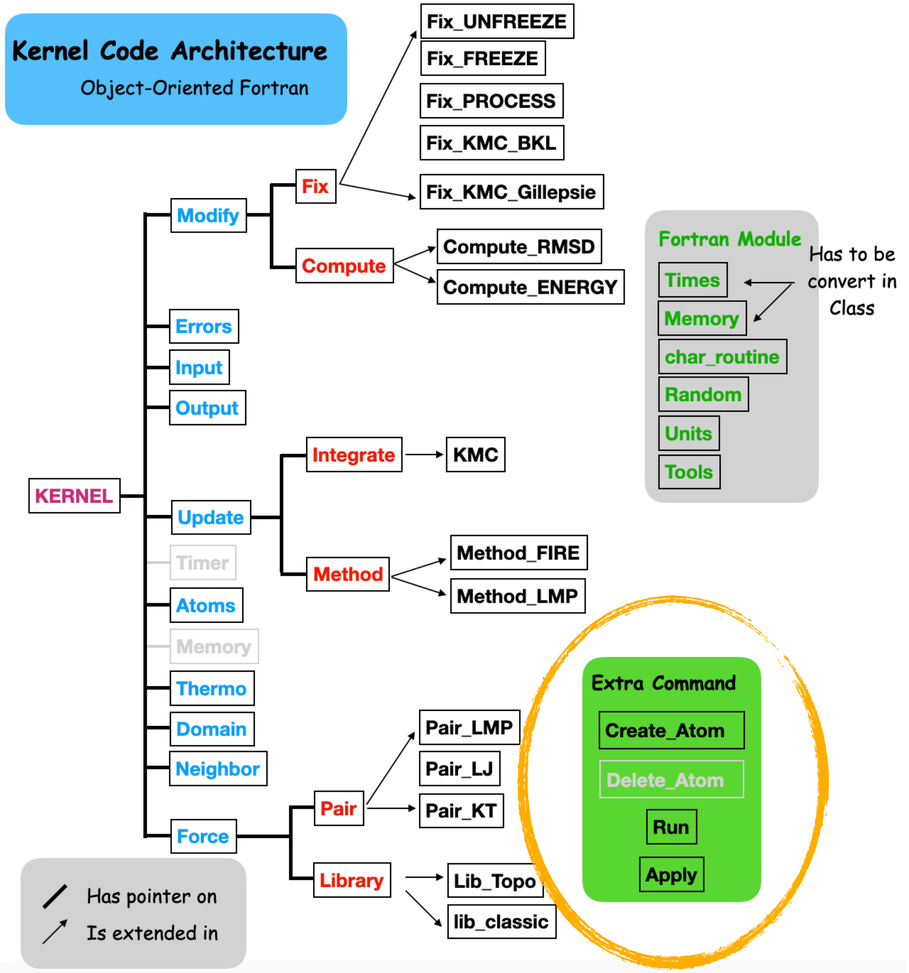
- `Extra Command` are class outside the Architecture but can act on the kernel because they are `extends`(`pointers`)
  - Answer to it own keyword in input script

  `Type command_run`: `run args…`

  Run the `update%integrate class` loaded

  `Type command_apply`: `apply args…`

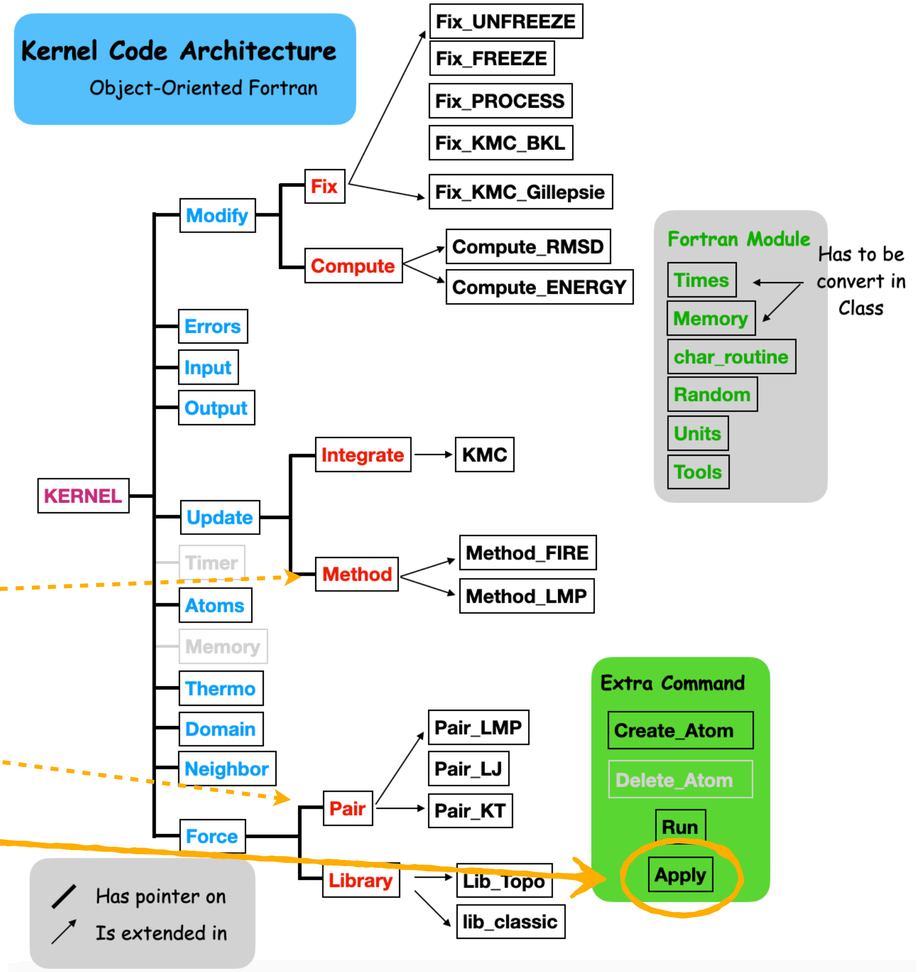
  Apply the `update%method class` loaded



Script Modularity

- Each simulation is a compilation of many `action classes`

```
system_dimension        3
bound_condition         p p p

input_data              Al_dumbbell.in


# — Relax the structure
method_style    fire

pair_style      smtb
pair_coeff      SMTB_Alparam.dat

# — Launch the method class
Apply           1.0e-3 1.0e-5 1000 10000
```



- The user can define its own `action classes`
  - → How to easily introduce a new `action class`?

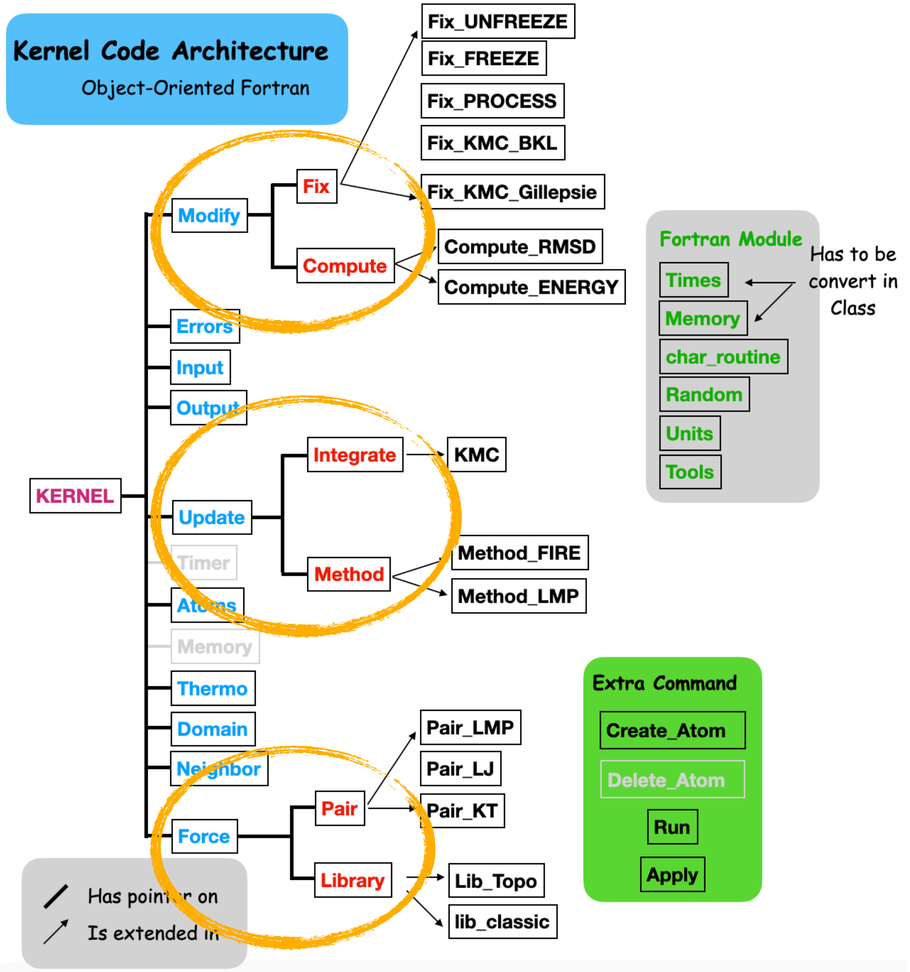
User Friendly

Action Class modularity

- Each simulation is a compilation of many `action classes`

```
system_dimension        3
bound_condition         p p p

input_data              Al_dumbbell.in


# — Relax the structure
method_style    fire

pair_style      smtb
pair_coeff      SMTB_Alparam.dat

# — Launch the method class
Apply           1.0e-3 1.0e-5 1000 10000
```

- Each action classes is built by
  `subroutine main_class% create_<actionclass>`
  - → `Call modify% Create_Fix()`
  - → `Call update% Create_Integrate()`
  - → `Call force% Create_Pair()`

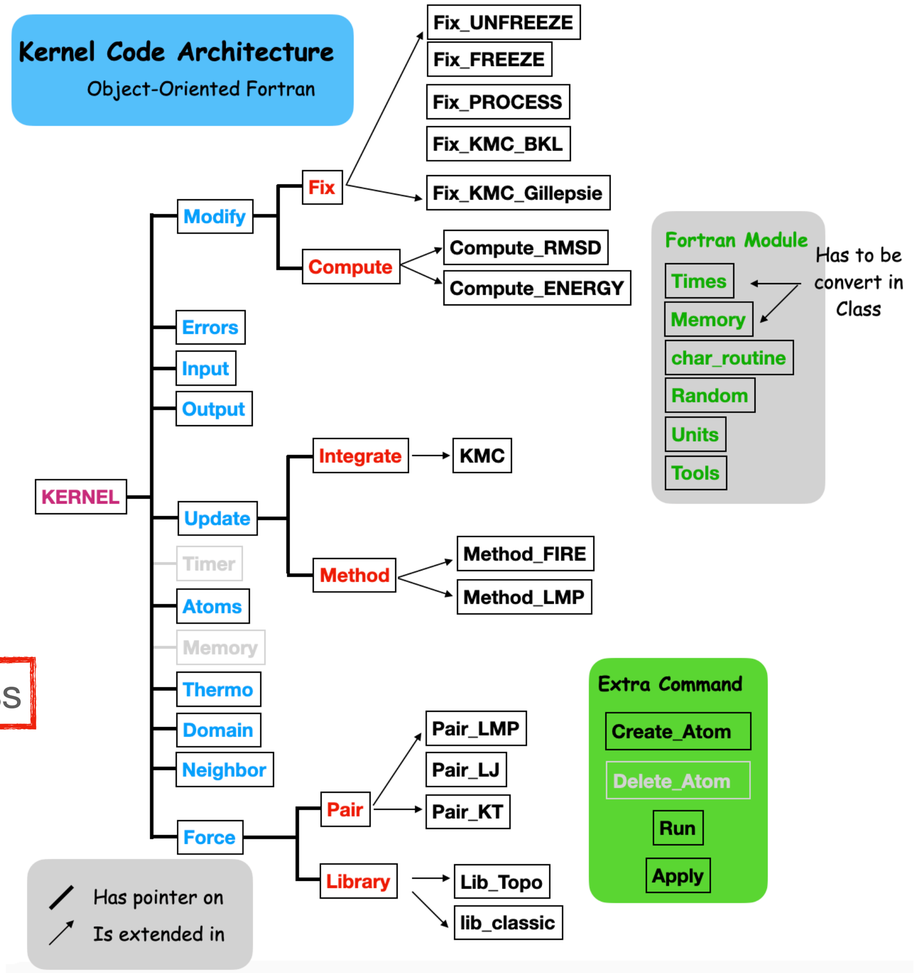


Action Class modularity

- Answer to the command: `integrate_style  kmc`
  - `class Update` contains a `Type(integrate), pointer :: integrates`

```
module subroutine create_integrate( self, nwords, words )
  use header_kmc
  class( update ), intent( inout ) :: self
  integer,         intent( in ) :: nwords
  character(*),    intent( in ) :: words(:)
  class( integrate ), pointer :: new_integrate
     […]
  select case( words(2) )

    case( “kmc” ); allocate( kmc::new_integrate )

    case default
      call bugs% error("UPDATE->create_integrate","The run_style kerword does not exist",words)
  end select

  call new_integrate% command( self% kernels, nwords-1, words(2:) )
  self% integrates => new_integrates
  self% nintegrates = self% nintegrates + 1

end subroutine create_integrate
```

Link pointers class



Action Class modularity

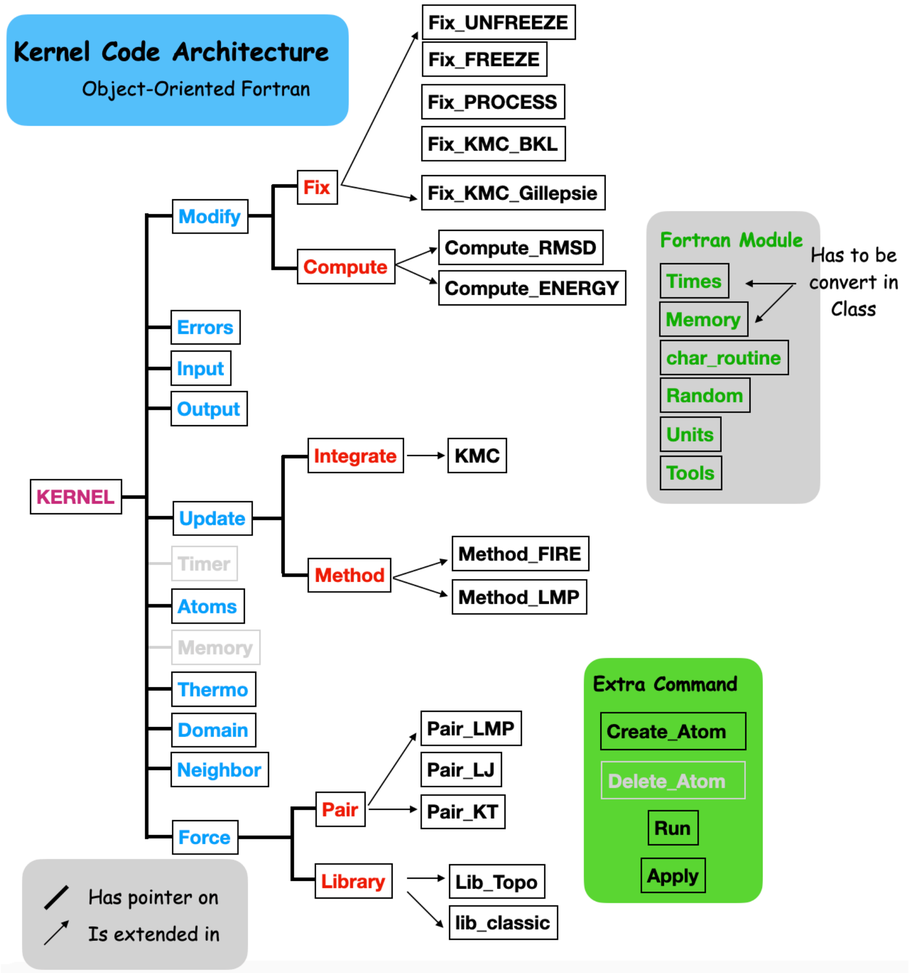
- Answer to the command: `integrate_style  kmc`
  - `class Update` contains a `Type(integrate), pointer :: integrates`

```
module subroutine create_integrate( self, nwords, words )
  use header_kmc
  class( update ), intent( inout ) :: self
  integer,         intent( in ) :: nwords
  character(*),    intent( in ) :: words(:)
  class( integrate ), pointer :: new_integrate
     […]
  select case( words(2) )

    case( “kmc” ); allocate( kmc::new_integrate )

    case default
      call bugs% error("UPDATE->create_integrate","The run_style kerword does not exist",words)
  end select

  call new_integrate% command( self% kernels, nwords-1, words(2:) )
  self% integrates => new_integrates
  self% nintegrates = self% nintegrates + 1

end subroutine create_integrate
```

Use the module header_kmc

Use keyword

- Action class needs 2 elements to by recognised
  - The module name: `use header_kmc`
  - Script Keyword: `case( “kmc” )`



Action class recognition

- LAMMPS use a preprocessing tricks in the header file

```
#ifdef INTEGRATE_CLASS

IntegrateStyle(kmc,kmc)

#else

module header_kmc
  […]
  type, extends( integrate ) :: kmc
  […]
  End type kmc
end module header_kmc

#endif
```

2 parts in header file:
- `class_pattern(key,type_name)`
- `module` declaration

- The `key` is the keyword used in the command script
- The `type_name` is the name of the type

Action class recognition

- Replace the `use module`

```
module subroutine create_integrate( self, nwords, words )

  use header_kmc

  class( update ), intent( inout ) :: self
  integer,         intent( in ) :: nwords
  character(*),    intent( in ) :: words(:)
  class( integrate ), pointer :: new_integrate
     […]
```

```
#define INTEGRATE_CLASS

#ifdef __GFORTRAN__
#define IntegrateStyle(key,class) use header_/**/class
#elif
#define IntegrateStyle(key,class) use header_##class
#endif
#include “header_kmc.f90”

#undef IntegrateStyle
#undef INTEGRATE_CLASS
```

```
#ifdef INTEGRATE_CLASS
IntegrateStyle(kmc,kmc)
#else
module header_kmc
  […]
end module header_kmc
#endif
```

Action class recognition

- Replace the `select case`

```
module subroutine create_integrate( self, nwords, words )

  class( integrate ), pointer :: new_integrate
     […]
  select case( words(2) )
    case( "kmc" ); allocate( kmc::new_integrate )
    case default
      call bugs% error("UPDATE->create_integrate","The run_style kerword does not exist",words)
  end select
```

```
#define INTEGRATE_CLASS

#define IntegrateStyle(key,Class) \
 case( "key" ); \
 allocate( Class::new_integrate )
#include “header_kmc.f90”

#undef IntegrateStyle
#undef INTEGRATE_CLASS
```

```
#ifdef INTEGRATE_CLASS
IntegrateStyle(kmc,kmc)
#else
module header_kmc
  […]
end module header_kmc
#endif
```

Action class recognition

- Generalisation at all `type, extends(integrate) ::`

- The extended `action class` is define in 2 file

  `integrate_kmc.f90` ← Module

  `submodule_kmc.f90` ← Submodule

  → Impose the name of extended class in file name

- Before to compile a script take all the user class and include them one file

`integrate_*` → `style_integrate.f90`

```
#include “integrate_kmc.f90”
#include “integrate_verlet.f90”
…
```

```
module subroutine create_integrate( self, nwords, words )

#define INTEGRATE_CLASS
#ifdef __GFORTRAN__
#define IntegrateStyle(key,class) use type_/**/class
#elif
#define IntegrateStyle(key,class) use type_##class
#endif
#include "style_integrate.f90"
#undef IntegrateStyle
#undef INTEGRATE_CLASS

  class( update ), intent( inout ) :: self
  integer,         intent( in ) :: nwords
  character(*),    intent( in ) :: words(:)
  class( integrate ), pointer :: new_integrate
     […]
  select case( words(2) )

#define INTEGRATE_CLASS
#define IntegrateStyle(key,Class) \
 case( "key" ); \
 allocate( Class::new_integrate )
#include "style_integrate.f90"
#undef IntegrateStyle
#undef INTEGRATE_CLASS

    case default
      call err% error(“UPDATE->create_integrate”,”…”,words)

  end select
  call new_integrate% command( self% kerr, nwords-1, words(2:) )
  self% integrates => new_integrate
  self% nintegrates = self% nintegrates + 1

end subroutine create_integrate
```

Interaction between the `class`

- The extended `action class` is define in 2 file

  `integrate_kmc.f90` ← Module

  `submodule_kmc.f90` ← Submodule

- Name rules: the **header file name** has to indicate from which `class` is extended

		`#ifdef`	`pattern`
From `type Fix`:	fix_*.f90	`FIX_CLASS`	`FixStyle(key,class)`
From `type Compute`:	compute_*.f90	`COMPUTE_CLASS`	`ComputeStyle(key,class)`
From `type Integrate`:	integrate_*.f90	`INTEGRATE_CLASS`	`IntegrateStyle(key,class)`
From `type Method`:	method_*.f90	`METHOD_CLASS`	`MethodStyle(key,class)`
From `type Pair`:	pair_*.f90	`PAIR_CLASS`	`PairStyle(key,class)`
From `type Library`:	library_*.f90	`LIBARY_CLASS`	`LibraryStyle(key,class)`
From `type command`:	command_*.f90	`COMMAND_CLASS`	`CommandStyle(key,class)`

- Name rules: no rule for the submodule file name

Now you know how to make a FLAMMPS!!