Structuration de codes

28 mars 2018

DEVCAL

Séparation par type de fichiers

Objectif : Avoir un répertoire racine lisible

Organisation possible

- src : fichiers sources
- doc : documentation
- obj : fichier objets
- data : résultats de simulation (lien symbolique)
- mod : fichiers générés pour les modules (fortran)
- tests : tests unitaires
- include : fichiers d'entête (C/C++)

Adaptation du Makefile

- Déplacement des sources dans src et fichiers objets dans obj

```
SRC=initialisation.f90 ...
OBJS_NOPREFIX= $(SRC:.f90=.o)
OBJS=$(addprefix obj/,$(OBJS_NOPREFIX))
obj/%.o : src/%.f90
        ...
```

- Fichiers générés pour les modules sont placés dans le dossier mod
  - gfortran -Jmod
  - ifort -module mod
- Séparation possible des fichiers d'entete (.h) en C/C++ : -Iinclude
- Désactivation de l'affichage de commandes : @....

```
obj/%.o : src/%.f90
        @echo "compilation de " $<
```

Exercice de structuration

- Structurer le code exercice_structuration/codedossier
- Ajouter un Makefile dans le dossier doc pour compiler la documentation latex

Génération automatique des dépendances avec ifort

- Ajouter dans le Makefile

```
FFLAGS= .... -gen-dep=Makefile.depend -gen-depformat=make
clean :
        ...
        echo >Makefile.depend

include Makefile.depend
```

- Créer un fichier vide : touch Makefile.depend

Bonnes pratiques....

Fonctions

- une fonction doit être courte avec un seul objectif
- décomposer en sous-fonctions si le traitement dépasse une page.
- limiter le nombre d'arguments
- fonctions utilisant les mêmes groupes d'arguments : regroupement par la technique du "Parameter Object"

```
void f(Date start, Date end);
void g(Date start, Date end);
void h(Date start, Date end);
```

$\Rightarrow$

```
void f(DateRange date);
void g(DateRange date);
void h(DateRange date);
```

Quelques fonctions par fichiers

- Des fichiers courts, sans tomber dans l'excès
- En orienté objet
  - une classe = un fichier
  - séparer le code template (.hpp) du fichier d'entête (.h)

Modules en fortran

- Un module = un fichier
- Déclaration des types dans un module
- Interdire l'accès aux fonctions internes : private.
- utiliser le mot clé ONLY pour l'utilisation de quelques membres d'un module

```
USE mymodule , ONLY: name1
```

- Eviter les variables globales dans un module ayant l'attribut save. Préférer un type dérivé englobant toutes les variables globales du module et passer en argument des routines.

Modules en fortran

A éviter

```
module MyMod

  integer,allocatable, private, save :: A(:)

contains

  subroutine init()
   ! allocating and initializing A
    allocate(A(n))
   end subroutine init

  subroutine calc()
   ! Using A
  end subroutine
 end module
```

Modules en fortran

A préférer

```
module MyMod

  type myType
    integer, allocatable :: A(:)
  end type myType

contains

  subroutine init(obj, n)
    type(myType), intent(inout) :: obj
    integer, intent(in) :: n

    allocate(obj%A(n))
  end subroutine init
end module MyMod
```

const - intent

Utiliser au maximum les mots clés intent et const.

- Documentation et protection
- Fonctions ne modifiant pas les attributs de l'objet

```
bool f (const myObj& obj);
bool myObj::isValid () const { return m_bvalid; }
```

```
function aire(self) result(aire)
    type(cercle), intent(in) :: self
    real :: aire
    aire = pi * this%rayon**2
end function aire
```

Enumération

Même radical pour tous les membres d'une énumération

```
/*
  * Valeurs possibles de l'état du voyant
  * OFF   : éteint
  * ON    : allume fixe
  * BLINK : clignotant
  */
  enum
  {
    VOYANT_STS_OFF,
    VOYANT_STS_ON,
    VOYANT_STS_BLINK,

    VOYANT_STS_INVALID
  };
```

Supprimer la réaffectation de paramètres

Utiliser une variable locale

```
int Discount(int x)
{
  if (x > 50)
  {
    x -= 2;
  }
  //...
}
```

⇒

```
int Discount(int x)
{
  int resultat = x;

  if (x > 50)
  {
    resultat -= 2;
  }
  //...
}
```

Règles de nommage

- être régulier dans sa notation
- utiliser des noms courts pour les variables à vie courte ou de boucle.
- éviter les noms génériques du type temp.
- utiliser des noms de constantes ayant un sens.

Suggestion

Identificateur	Règles	Exemple
Variables	commence par une miniscule	mass
Constantes	toujours en majuscules	MAX_HEIGHT
Classes	nom, commence par une majuscule	PhoneNumber
Méhodes	verbe, commence par une miniscule	draw()

Indentation

- indent pour C/C++ : https ://www.gnu.org/software/indent
- findent pour Fortran : https ://github.com/wvermin/findent
- Grand choix des styles (K&R, Allman, GNU). Réindentation possible.

Documentation du source

- doxygen : multi-langages et simple à mettre en oeuvre
- FORD : uniquement pour fortran mais très performant. génère des graphes d'appels sans documentation spécifique.
- docstrings : intégré à python

Doxygen

```
!-----------------------------------------------
!> \brief      Calcule la fonction g(x)=3*x^2+5 au point x
!! \details   Detail du traitement optionnel
!! \return     3*x^2+5
!-----------------------------------------------
 function g(x)
  implicit none
  real(kind=8), intent (in) :: x !< abscisse
  real(kind=8) :: g
  g = 3*x*x+5
end function
```

Protection contre les inclusions multiples

Fichiers d'entête (.h) du C/C++

- définition d'une macro
- Eviter les problèmes de compilation (redéfinitions).

nomdufichier.h

```
#ifndef __NOMDUFICHIER_H__
#define __NOMDUFICHIER_H__

/* zone protégée contre les inclusions multiples */
struct MaStructure {
        int x;
};

#endif /* __NOMDUFICHIER_H__ */
```

Analyse statique du code

Objectifs

- Évaluation de la qualité du code : commentaire, conformité aux règles de codage
- Analyse statique du code : indication d'erreurs potentielles.

Outils

- pylint : python
- sonar : java, PHP, C, javascript, ...
- sonarlint : plugins pour certains éditeurs

Log d'appel de fonctions - fortran

- Afficher l'appel des fonctions avec leurs arguments en mode débogage.
- Eviter un surcoût lors d'une exécution normale.
- compiler avec le préprocesseur : -cpp -DDEBUG

debug.f90

```
#ifdef DEBUG
#define mydebug write(*,*) '(',__FILE__,__LINE__,'):',
#else
#define mydebug if (.false.) write(*,*)
#endif
```

- inclure au début de chaque fichier : #include debug.f90

Log d'appel de fonctions - fortran

main.f90

```
#include debug.f90

subroutine f(x, n)
  implicit none
  real(kind=8), intent (in) :: x
  integer, intent (in) :: n

  mydebug "f", x, n
  call g(2*x)

end subroutine
```

sortie

```
(main.f90 16 ) : f 3.1400000000000001 122
```

Log d'appel de fonctions - C

debug.h

```
#ifdef DEBUG
#define mydebug(fmt, ...)                \
 printf("%s %d : %s(" fmt ")\n",         \
        __FILE__, __LINE__, __FUNCTION__, __VA_ARGS__);
#else
#define mydebug
#endif
```

Log d'appel de fonctions - C

main.c

```
void f(double x, int n)
{
    mydebug("%g, %d", x, n);
    g(2.*x);
}
```

sortie

```
main.c 12 : f(3.14, 122)
```

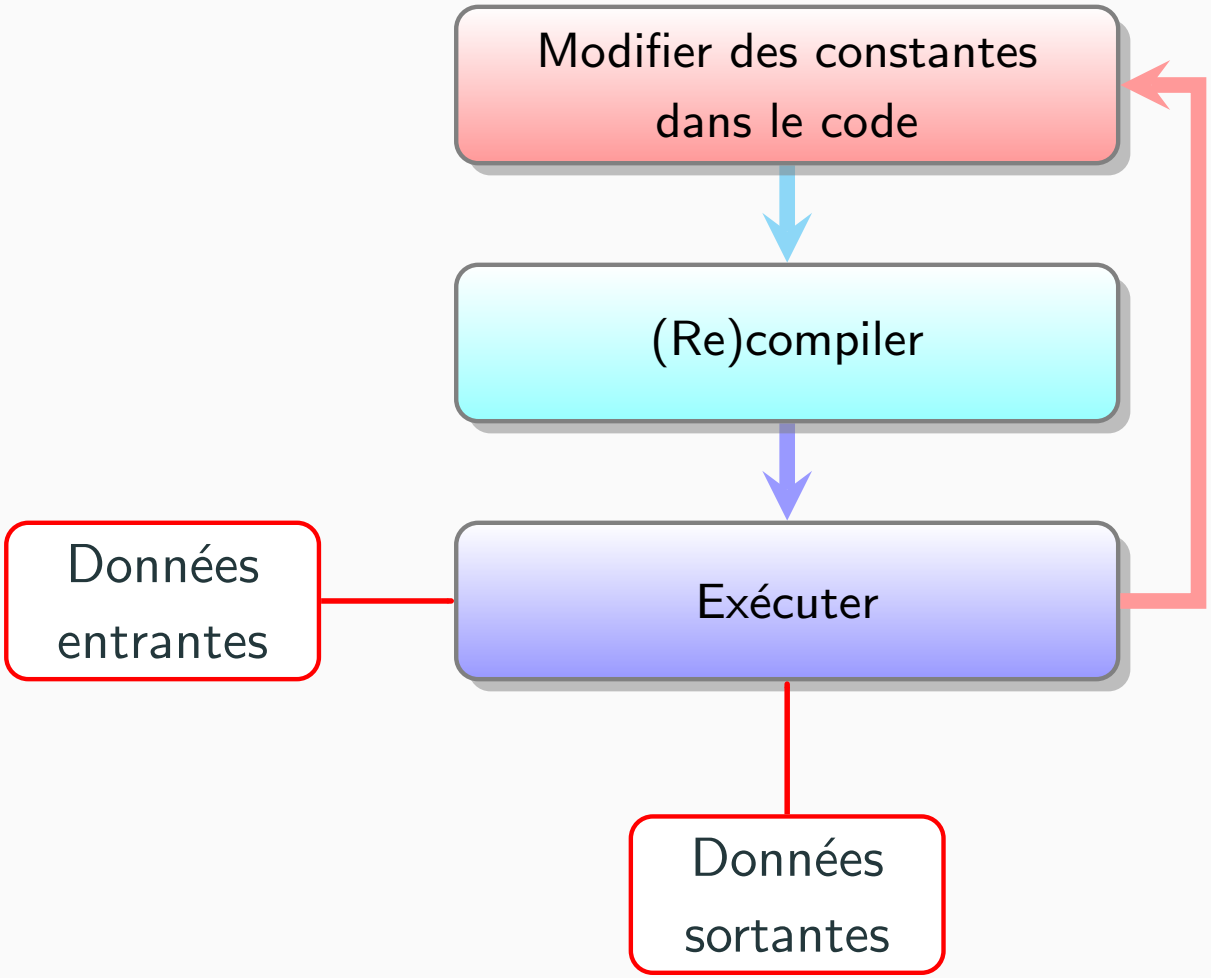
- en C++, surcharge de l'opérateur <<

Exercice

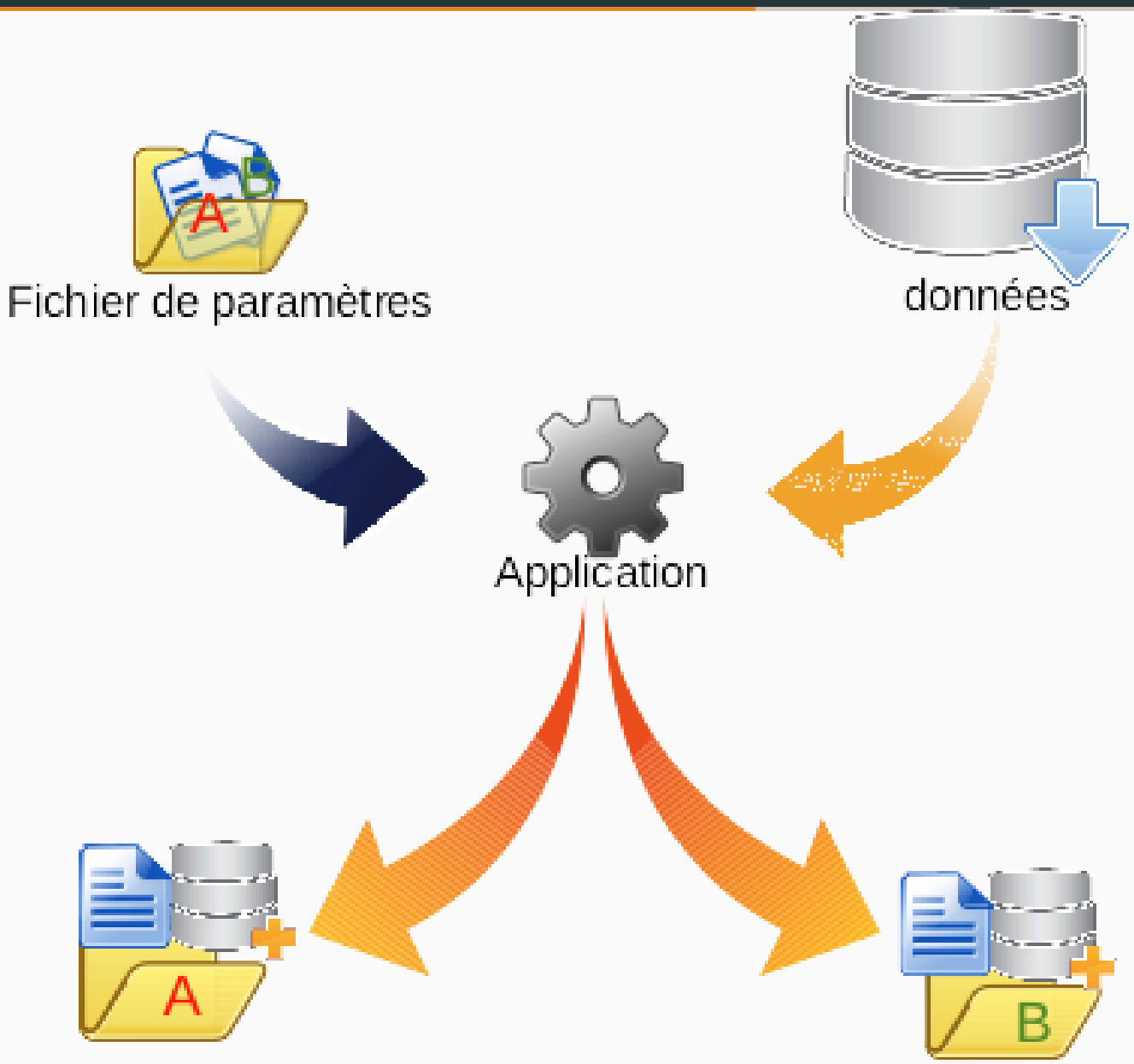
- Modifier le code exercice_structuration/debug
- Définir DEBUG_INFO=0, DEBUG_WARN=1 et DEBUG_ERROR=2
- Afficher le debug de la fonction f uniquement
  si DEBUG = DEBUG_INFO, DEBUG_WARN ou DEBUG_ERROR
- Afficher le debug de la fonction g uniquement
  si DEBUG = DEBUG_WARN ou DEBUG_ERROR
- Compiler avec -DDEBUG=DEBUG_INFO ou
  -DDEBUG=DEBUG_WARN

Fichiers de paramètres

Pourquoi utiliser un fichier de paramètres ?



Fichier de paramètres



Format des fichiers de paramètres

- maison

Gadget2

```
%  Caracteristics of run
...
Omega0                  0.3
OmegaLambda              0.7
OmegaBaryon              0.0
HubbleParam              0.7
BoxSize                  0
```

- Namelist (fortran)
- Json
- XML : assez vite illisible
- ...

Namelist en fortran

- Groupe nommé de variables

```
namelist /GROUP_NAME/ var1, var2, var3, ...
```

- Type des variables : entiers, réels, chaînes, tableaux, type dérivés
- Lecture/écriture de fichiers texte contenant un ou plusieurs namelist.
- Format du fichier texte

```
  &GROUP_NAME
var1 = value1
var2 = value2
var3 = value3  ! un commentaire
...
/
```

- ordre des variables indifférents

Exemple de fichier namelist

FichierEntree.nml

```
!-------------------------------------
!-------------------------------------
&GROUPE_1
  char_variable = "a char variable",
  logical_variable = T,  ! = .TRUE.
  nitems = 5,
  list_variable = 0.1  0.2  0.3  0.4  0.5
/

!-------------------------------------
!-------------------------------------
&GROUPE_2
  float_array(1) = 3.14
  float_array(2) = 6.28
/
```

Lecture/Ecriture de fichier namelist

- Lecture

```
open(unit=10, file="FichierEntree.nml", status="old")
read(unit=10, GROUP_NAME)  ! read(10, nml=GROUP_NAME)
close(unit=10)
```

Les tableaux obligatoirement pré-alloués avant la lecture.

- Ecriture

```
open(unit=10, file="FichierSortie.nml", status="new")
write(unit=10, GROUP_NAME) ! write(10, nml=GROUP_NAME)
close(unit=10)
```

- Conseil : créer 2 fonctions séparées qui sont appelées par le programme principal.

Stockage des données

- Classe contenant uniquement les champs du namelist

```
&GROUPE_1
  vitesse_m_s = 9.5
/
```

```
type GROUPE_1
  real(kind=8) :: vitesse_m_s !< vitesse en m/s
end type
```

- Variables secondaires dans une classe dérivée.

```
type, extends(GROUPE_1) ::  classe_derivee
  real(kind=8) :: vitesse_km_h !< vitesse en km/h
end type
```

Exercice namelist.f

input.nml

```
&mesinputs
    fichiersortie = 'output.nml'
    npas          = 10
    t0            = 2000.5
    angles        = 0.12D0 , 3.14, 1.4
/
```

- tparam : type dérivé pour stocker les données
- readparam : fonction de lecture
- writeparam : fonction d'écriture

JSON : JavaScript Object Notation

- Format léger d'échanges de données
- Un objet (entre $\{\}$) = ensemble de couples : nom, valeur
- Valeurs : chaînes, entiers, flottants, booléns, tableaux (entre []) de valeurs ou de couples
- Format indépendant du langage

```
{
   "nom_1" : valeur1,
   "nom_2" : valeur2,
   "nom_3" : valeur3,
   "nom_4" : [ valeur4a,  valeur4b,  .... ],
   "nom_5" : {  "nom5a" :  valeur5a, ... }
}
```

Exemple de fichiers JSON

FichierEntree.json

```
{
   "fichiersortie" : "output.nml",
   "npas"          : 10,
   "t0"            : 2000.5,
   "angles"        : [ 0.12E0, 3.14, 1.4 ],
   "masse"        : {
                       "MERCURY" : 0.0553,
                       "VENUS"   : 0.815,
                       "EARTH"   : 1
                     } ,
  "s"            : "machaine",
  "d"            :  7.2
 }
```

Librairies

- C++ : jsoncpp https ://github.com/open-source-parsers/jsoncpp
- Fortran : json-fortran https ://github.com/jacobwilliams/json-fortran
- Python : module json

C++ - jsoncpp

- Documentation : https ://en.wikibooks.org/wiki/JsonCpp
- 1 seul fichier .h et .cpp à copier avec ces sources
- Génération du fichier json/json.h et jsoncpp.cpp

```
git clone \
https://github.com/open-source-parsers/jsoncpp.git
cd jsoncpp
python amalgamate.py
```

C++ - jsoncpp

Lecture

```cpp
#include <json/json.h>

std::string s;
double d;
std::ifstream ifs("FichierEntree.json");
Json::Value datain;
ifs >> datain;
s = datain.get("s", "invalidfile").asString();
d = datain.get("d", 0).asDouble();
```

C++ - jsoncpp

Ecriture

```cpp
#include <json/json.h>

Json::Value dataout;
dataout["s"] = "machaine";
dataout["d"]  = 7;
std::ofstream ofs("FichierSortie.json");
ofs << dataout << std::endl;
```

Fortran - json-fortran

```
pip install fobis.py
wget https://github.com/jacobwilliams/json-fortran\
/archive/master.zip
unzip master.zip
cd json-fortran-master
./build.sh
```

- Module json_module
- compilation
  gfortran -Jchemin/lib -Lchemin/lib -ljsonfortran ....

Fortran - json-fortran

Lecture

```
use json_module

type(json_file) :: json
logical :: found
character (len=:), allocatable :: s
real(kind=8) :: d

! chargement du fichier
call json%initialize()
call json%load_file(filename='FichierEntree.json')

!récuperation des valeurs
call json%get('s', s, found)
call json%get('d', d, found)
```

Fortran - json-fortran

Ecriture

```
use json_module

type(json_file) :: json

! préparation des données json en memoire
call json%initialize(compact_reals=.false.)
call json%add('s', 'machaine')
call json%add('d', 7)

!écriture du fichier
call json%print_file('FichierSortie.json')
```

Python - module json

- inclus dans la librairie standard de python
- charge les données sous forme de dictionnaire.

```
import json

with open('FichierEntree.json') as json_data_file:
    datain = json.load(json_data_file)
print(datain)
w = datain["angles"]
print(w)

dataout = { "s": "machaine", "d": 7 }
with open('FichierSortie.json', 'w') as outfile:
    json.dump(dataout, outfile)
```

Exercice exjson

- Pour fortran : instructions d'installation dans : exjson/fortran/install.txt
- source : exjson.cpp, exjson.f90 ou exjson.py
- exécution : exjson input.json

input.json

```
{
    "fichiersortie" : "output.json",
    "npas"          : 10,
    "t0"            : 2000.5,
    "angles"        : [ 0.12E0 , 3.14, 1.4 ]
}
```

- tparam : type dérivé ou classe pour stocker les données
- readparam/read : fonction de lecture
- writeparam/write : fonction d'écriture