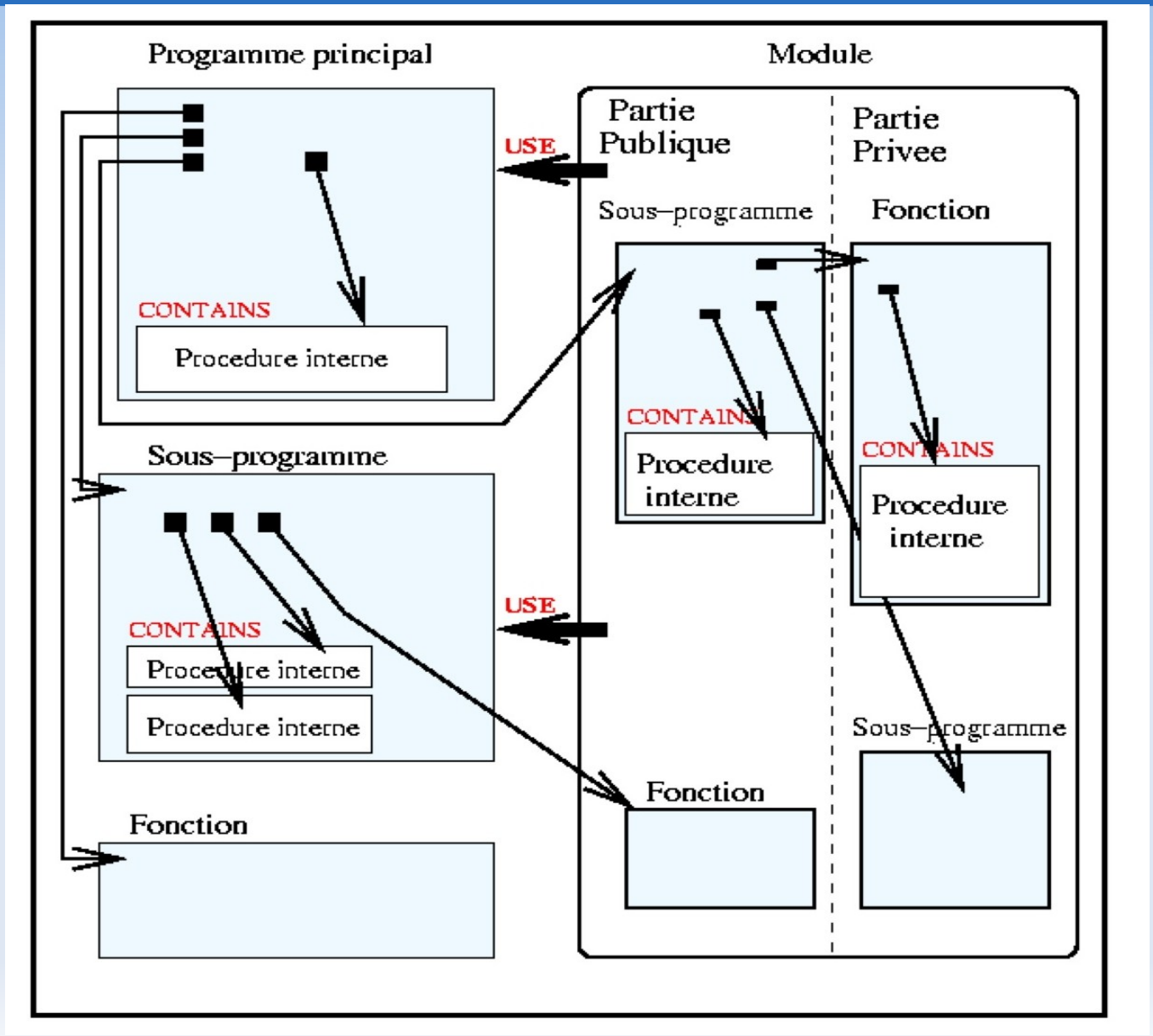
# La programmation scientifique en Fortran 90

UE LI 222 – Automne 2007
Université Pierre et Marie Curie

Pierre Fortin
*pierre.fortin @ lip6.fr*

**Cours 8 : modularité et gestion de projet**

# Structure d'un programme Fortran



# Module

- ***Module*** : unité de compilation autonome, pouvant contenir :
  - des définitions de procédures
  - des interfaces de procédures
  - des définitions de types dérivés
  - ...
- Le contenu d'un module peut être utilisé au sein d'une autre unité de programme (unité utilisatrice) avec l'instruction **USE** (doit être placée avant IMPLICIT NONE)

# Exemple de module

Fonction *lire_entier* (voir TD5 ex.1) :

```
MODULE lecture
  IMPLICIT NONE
  CONTAINS
    INTEGER FUNCTION lire_entier(msg, maximum)
      CHARACTER(LEN=*), INTENT(IN) :: msg
      INTEGER, INTENT(IN), OPTIONAL :: maximum
      PRINT *, msg
      IF (present(maximum)) THEN
        DO
          READ *, lire_entier
          IF (lire_entier <= maximum) EXIT
        END DO
      ELSE
        READ *, lire_entier
      END IF
    END FUNCTION lire_entier
END MODULE lecture
```

```
! suite

PROGRAM exemple_lecture
  USE lecture  ! avant le IMPLICIT NONE !
  IMPLICIT NONE
  INTEGER :: n

  n = lire_entier("Entrez  un entier <= 100 : ", 100);
  PRINT *, "n = ", n

  n = lire_entier("Entrez un entier : ");
  PRINT *, "n = ", n

END PROGRAM exemple_lecture
```

# Modules et interface explicite de procédures

- Comme indiqué au cours 5 sur les procédures, il existe deux possibilités d'interface explicite avec les modules :
  - le module contient la définition de la procédure appelée : utilisation du mot-clé **CONTAINS** (voir exemple précédent de la fonction *lire_entier*)
  - le module contient le bloc interface de la procédure (externe) appelée (voir exemple suivant)

# Exemple de module avec bloc interface

Fonction *lire_entier* :

```
MODULE lecture_interface
  IMPLICIT NONE
  INTERFACE
    INTEGER FUNCTION lire_entier(msg, maximum)
      CHARACTER(LEN=*), INTENT(IN) :: msg
      INTEGER, INTENT(IN), OPTIONAL :: maximum
    END FUNCTION lire_entier
  END INTERFACE
END MODULE lecture_interface

INTEGER FUNCTION lire_entier(msg, maximum)
  CHARACTER(LEN=*), INTENT(IN) :: msg
  INTEGER, INTENT(IN), OPTIONAL :: maximum
  PRINT *, msg
  IF (present(maximum)) THEN
    DO
      READ *, lire_entier
      ! suite
      IF (lire_entier <= maximum) EXIT
    END DO
  ELSE
    READ *, lire_entier
  END IF
END FUNCTION lire_entier

PROGRAM exemple_lecture
  USE lecture_interface
  IMPLICIT NONE
  INTEGER :: n
  n = lire_entier("Entrez  un entier <= 100 : ", 100);
  PRINT *, "n = ", n
  n = lire_entier("Entrez un entier : ");
  PRINT *, "n = ", n
END PROGRAM exemple_lecture
```

# Module et type dérivés

Un module peut contenir la définition d'un type dérivé :

```
MODULE definition_date
  IMPLICIT NONE
  TYPE DATE
    CHARACTER(LEN=10) :: nom_jour
    INTEGER           :: jour
    CHARACTER(LEN=10) :: mois
    INTEGER           :: annee
  END TYPE DATE
END MODULE definition_date

PROGRAM exemple_date
  USE definition_date
  IMPLICIT NONE
  TYPE(DATE) :: une_date = date('lundi', 12, 'decembre', 2028)
  PRINT *, une_date
END PROGRAM exemple_date
```

# Module et procédure avec type dérivé

- Reprise de l'exemple d'une procédure avec un type dérivé en argument (cours 6) :
  - sans module :
    - nécessité de définir le type dérivé dans le programme principal, dans l'interface de la procédure et dans la définition de la procédure
    - à chaque fois avec le mot-clé SEQUENCE
  - avec module (et CONTAINS) : une seule définition suffit

# Sans module :

```
SUBROUTINE affiche(d)
        IMPLICIT NONE
        TYPE DATE
            SEQUENCE
            CHARACTER(LEN=10) :: nom_jour
            INTEGER           :: jour
            CHARACTER(LEN=10) :: mois
            INTEGER           :: annee
        END TYPE DATE
        TYPE(DATE), INTENT(IN) :: d
        PRINT *, 'Nous somme le ', &
            d%nom_jour, ' ', d%jour, ' ', &
            d%mois, ' ', d%annee
END SUBROUTINE affiche
```

```
PROGRAM test_type_derive_arg
IMPLICIT NONE
TYPE DATE
        SEQUENCE
        CHARACTER(LEN=10) :: nom_jour
        INTEGER           :: jour
        CHARACTER(LEN=10) :: mois
        INTEGER           :: annee
END TYPE DATE
INTERFACE
    SUBROUTINE affiche(d)
        TYPE DATE
            SEQUENCE
            ... ! idem
        END TYPE DATE
        TYPE(DATE), INTENT(IN) :: d
    END SUBROUTINE affiche
END INTERFACE
TYPE(DATE) :: une_date = &
  date('lundi', 12, 'decembre', 2028)
CALL affiche(une_date)
END PROGRAM test_type_derive_arg
```

# Avec module (et CONTAINS) :

```
MODULE mod_date
  IMPLICIT NONE
  TYPE DATE
     CHARACTER(LEN=10) :: nom_jour
     INTEGER           :: jour
     CHARACTER(LEN=10) :: mois
     INTEGER           :: annee
  END TYPE DATE

  CONTAINS
    SUBROUTINE affiche(d)
     IMPLICIT NONE
     TYPE(DATE), INTENT(IN) :: d
     PRINT *, 'Nous somme le ', &
         d%nom_jour, ' ', d%jour, ' ', &
         d%mois, ' ', d%annee
  END SUBROUTINE affiche
END MODULE mod_date
```

```
! suite

PROGRAM test_type_derive_arg
  USE mod_date
  IMPLICIT NONE
  TYPE(DATE) :: une_date = &
  date('lundi', 12, 'decembre', 2028)

  CALL affiche(une_date)
END PROGRAM test_type_derive_arg
```

# Dépendances entre modules

Un module peut faire appel à un autre module, mais tout cycle de dépendance est interdit : module A $\xrightarrow{\text{USE}}$ module B $\xrightarrow{\text{USE}}$ module A

```
MODULE definition_date
  IMPLICIT NONE
  TYPE DATE
       CHARACTER(LEN=10) :: nom_jour
       INTEGER           :: jour
       CHARACTER(LEN=10) :: mois
       INTEGER           :: annee
  END TYPE DATE
END MODULE definition_date
MODULE affiche_date
  USE definition_date
  IMPLICIT NONE
  CONTAINS
    SUBROUTINE affiche(d)
       IMPLICIT NONE
       TYPE(DATE), INTENT(IN) :: d
```

```
! suite
          PRINT *, 'Nous somme le ', &
              d%nom_jour, ' ', d%jour, ' ', &
              d%mois, ' ', d%annee
  END SUBROUTINE affiche
END MODULE affiche_date

PROGRAM test_type_derive_arg
  USE definition_date ! facultatif
  USE affiche_date
  IMPLICIT NONE
  TYPE(DATE) :: une_date = &
  date('lundi', 12, 'decembre', 2028)

  CALL affiche(une_date)
END PROGRAM test_type_derive_arg
```

# Module et fichiers

- La définition du module peut être placée :
  - **soit dans le même fichier** que l'unité utilisatrice : le module doit alors apparaître **avant** l'unité utilisatrice dans le fichier (voir exemples précédents)
  - **soit dans un fichier différent** du fichier contenant l'unité utilisatrice
    - dans ce cas, si le module contient un bloc interface, il est préférable d'inclure dans le même fichier que le module les définitions des procédures externes correspondantes
    - exemple de *lire_entier* :
      - 1 fichier contenant le module et la définition de la fonction *lire_entier*
      - 1 fichier contenant le programme principal

# Exemple : module et fichiers séparés

Reprise du premier exemple avec la fonction *lire_entier* :

```
!!! Fichier 'lecture_entier.f90'
MODULE lecture
  IMPLICIT NONE
  CONTAINS
    INTEGER FUNCTION lire_entier(msg, maximum)
      CHARACTER(LEN=*), INTENT(IN) :: msg
      INTEGER, INTENT(IN), OPTIONAL :: maximum
      PRINT *, msg
      IF (present(maximum)) THEN
        DO
          READ *, lire_entier
          IF (lire_entier <= maximum) EXIT
        END DO
      ELSE
        READ *, lire_entier
      END IF
    END FUNCTION lire_entier
END MODULE lecture
```

```
!!! Fichier 'main.f90'
PROGRAM exemple_lecture
  USE lecture
  IMPLICIT NONE
  INTEGER :: n

  n = lire_entier("Entrez  un entier <= 100 : ", 100);
  PRINT *, "n = ", n

  n = lire_entier("Entrez un entier : ");
  PRINT *, "n = ", n

END PROGRAM exemple_lecture
```

# Plusieurs fichiers sources sans module (non recommandé)

- Possible avec un bloc interface :

```
!!! Fichier 'lecture_entier.f90'
INTEGER FUNCTION lire_entier(msg, maximum)
  IMPLICIT NONE
  CHARACTER(LEN=*), INTENT(IN) :: msg
  INTEGER, INTENT(IN), OPTIONAL :: maximum

  PRINT *, msg
  IF (present(maximum)) THEN
    DO
      READ *, lire_entier
      IF (lire_entier <= maximum) EXIT
    END DO
  ELSE
    READ *, lire_entier
  END IF
END FUNCTION lire_entier
```

```
!!! Fichier 'main.f90'
PROGRAM exemple_lecture
  IMPLICIT NONE
  INTERFACE
    INTEGER FUNCTION lire_entier(msg, maximum)
      CHARACTER(LEN=*), INTENT(IN) :: msg
        INTEGER, INTENT(IN), OPTIONAL ::
       maximum
    END FUNCTION lire_entier
  END INTERFACE
  INTEGER :: n

  n = lire_entier("Entrez  un entier <= 100 : ", 100);
  PRINT *, "n = ", n

  n = lire_entier("Entrez un entier : ");
  PRINT *, "n = ", n

END PROGRAM exemple_lecture
```

# Compilation

- Comment compiler un programme Fortran 90 réparti sur plusieurs fichiers sources ?
  - 1 unique programme principal dans tous les fichiers
  - *compilation séparée*

    $\rightarrow$ 2 étapes :
    - compilation des fichiers objets
    - édition de liens

# Compilation séparée

- Compilation des fichiers objets :
  - compilation individuelle de chaque fichier source (extension : *.f90*) en un fichier objet (extension : *.o*)

```
> g95 -c lire_entier.f90
```

→ création du fichier objet 'lire_entier.o' et du fichier 'lecture.mod' correspondant au module *lecture* contenu dans le fichier 'lire_entier.f90'

```
> g95 -c main.f90
```

→ création du fichier objet 'main.o'

  - Fichiers '.o' : fichiers binaires
  - Fichiers '.mod' : fichiers textes (contenant « l'interface » du module)

# Compilation séparée

- Compilation des fichiers objets (suite) :
  - attention aux dépendances entre modules :
    - on compile d'abord les modules qui n'utilisent pas aucun autre module,
    - puis les modules qui utilisent des modules déjà compilés,
    - et ainsi de suite jusqu'au à la compilation du programme principal
- Edition de liens :
  - les fichiers objets sont «  liés  » pour construire le programme exécutable (ici appelé 'prog', et 'a.out sans l'option '-o')

```
> g95 lecture_entier.o main.o -o prog
```

# Compilation séparée

- On peut faire les deux étapes en une commande :

```
> g95 lecture_entier.f90 main.f90 -o prog
```

  mais  il faut respecter les dépendances entre les modules dans l'ordre des fichiers.

- Supposons que l'on a un nombre élévé de fichiers sources :
  - modification d'un petit nombre de fichiers sources
  - comment éviter de recompiler tous les fichiers sources ?

→ Utilisation de la commande ***make***

# Commande make

- La commande make permet :
  - de rassembler toutes les commandes nécessaires à la compilation d'un projet comprenant plusieurs fichiers
  - après modification d'une partie des fichiers sources, de déterminer automatiquement les commandes nécessaires pour la reconstruction du programme
    → à l'aide des *dates de dernière modification* des fichiers
- La commande make utilise par défaut un fichier « *Makefile* » (ou « makefile »), qui :
  - décrit les dépendances entre les différents fichiers (modules)
  - contient les commandes à exécuter pour compiler le programme

# Fichier Makefile

- constitué de règles de la forme :

  ***cible : dépendances***
  ***→|commandes***

  - *cible* : nom du fichier cible à (re)créer
  - *dépendances* : nom des fichiers qui doivent être (re)créés avant le fichier *cible*
    - pour un fichier objet : le fichier source correspondant et les fichiers '.mod' de tous les modules utilisés
    - pour l'exécutable final : tous les fichiers objets
  - *commandes* : commandes à exécuter pour (re)créer le fichier *cible* à partir des fichiers *dépendances*
  - Chaque commande ***doit*** être précédée par une tabulation (représentée ici par →| )

# Fichier Makefile

- Evaluation de chaque règle en deux temps :
  - d'abord : analyse des dépendances

    → **si** une dépendance est la cible d'une autre règle du Makefile, **alors** cette autre règle est d'abord évaluée
  - puis : exécution des commandes

    → **si** le fichier *cible* n'existe pas, **ou si** un fichier *dépendance* est ***plus récent*** que le fichier *cible* existant, **alors** les *commandes* sont exécutées
- Lancement de l'évaluation d'une règle :

```
> make cible
```

Si aucune *cible* n'est indiquée, on commence par évaluer la première règle rencontrée.

# Exemple de Makefile (1)

- Makefile non optimisé : on recompile tous les fichiers objets à chaque modification

```
# Makefile 1
# ceci est un commentaire

prog : lecture_entier.f90 main.f90
        g95 -c lecture_entier.f90
        g95 -c main.f90
        g95 lecture_entier.o main.o -o prog

clean :
        rm  -f  *.o  *.mod
```

# Exemple de Makefile (1)

- Première compilation (et après chaque modification d'un des fichiers sources) :

```
> make
g95 -c lecture_entier.f90
g95 -c main.f90
g95  lecture_entier.o main.o -o prog
```

- Si les fichiers sources n'ont pas été modifiés :

```
> make
make: « prog » est à jour.
```

- Suppression des fichiers '.o' et '.mod' :

```
> make clean
rm  -f  *.o  *.mod
```

# Exemple de Makefile (2)

- Possibilité de définir des variables dans le Makefile :

```
# Makefile 2
CC=g95

prog : lecture_entier.f90 main.f90
	$(CC) -c lecture_entier.f90
	$(CC) -c main.f90
	$(CC) lecture_entier.o main.o -o prog

clean :
	rm  -f  *.o  *.mod
```

# Exemple de Makefile optimisé (3)

```
# Makefile 3
CC=g95

prog : lecture_entier.o main.o
	$(CC) lecture_entier.o main.o  -o prog

# le fichier 'main.f90' utilise le module 'lecture'
main.o : main.f90 lecture.mod
	$(CC) -c main.f90

lecture_entier.o : lecture_entier.f90
	$(CC) -c lecture_entier.f90

# regle a rajouter pour chaque module (due a une specificite de G95)
lecture.mod : lecture_entier.o
	@if [ ! -f $@ ]; then rm -f $< ; $(MAKE) $< ; fi

clean :
	rm  -f  *.o  *.mod
```

# Exemple de Makefile optimisé (3)

- Lancement de la compilation :

```
> make
g95 -c lecture_entier.f90
g95 -c main.f90
g95  lecture_entier.o main.o -o prog
```

- Après modification de 'lecture_entier.f90' (sans que 'lecture.mod' ne soit modifié) : 'main.f90' n'est pas recompilé

```
> make
g95 -c lecture_entier.f90
g95  lecture_entier.o main.o -o prog
```

# Gestion d'un projet informatique

Critères de réussite d'un projet informatique :

- un projet informatique est réussi lorsque
  - les coûts et les délais sont maîtrisés
  - les besoins des utilisateurs satisfaits
  - les critères de qualité respectés
- la majeure partie du coût d'un logiciel est consacrée à sa maintenance !
- Une bonne analyse est indispensable au départ du projet.

# Les différentes étapes

- Analyse fonctionnelle du problème
- Choix des structures de données
- Conception des algorithmes
- Implémentation des algorithmes (en Fortran 90)
- Compilation
- Tests (exhaustifs) et corrections
- Rédaction de la documentation (manuel utilisateur, manuel de maintenance...)
- Améliorations et maintenance

# Analyse fonctionnelle du problème

A partir du « cahier des charges » (contrat qui spécifie le travail à réaliser) :

- « Découpage » du problème en modules et en procédures
- Analyse fonctionnelle descendante :
  - on commence par les modules et les procédures de plus « haut niveau » : ceux qui sont directement utilisés dans le programme principal
  - puis on « raffine » jusqu'aux modules et procédures de plus « bas niveau » : ceux qui ne font appel à aucun autre module ou procédure

# Analyse fonctionnelle du problème (2)

- Chaque module correspond à une «  entité  » du projet, exemple du TD4 :
  - un module définissant une fiche (et les éventuelles procédures correspondantes)
  - un module définissant un répertoire, qui utilise le module définissant une fiche
- Chaque module contient les procédures associées à cette «  entité  »
  - pour le module définissant la fiche : procédure de création d'une fiche, procédure d'affichage d'une fiche...
- Inutile de trop raffiner : de petites entités (proches du point de vue conceptuel) peuvent être regrouper dans un unique module.

# Analyse fonctionnelle du problème (3)

- Pour chaque procédure à développer :
  - donner son interface complète
  - décrire précisément son rôle
  - établir les liens (les appels) entre les différentes procédures, ainsi qu'entre le programme principal et les procédures

# Algorithmique, implémentation et tests

- A partir de l'analyse fonctionnelle, pour chaque module et chaque procédure :
  - définition des structures de données (types dérivés, tableaux...)
  - conception des algorithmes
- Implémentation de chaque module et procédure (en Fortran 90)
- Test de chaque module et procédure
- Puis : écriture des Makefiles, compilation et tests du projet complet, rédaction de la documentation.