Procedures in Modules (1)

Including all procedures within modules works very well in almost all programs

- There really isn’t much more to it

It doesn’t handle very large modules well
Try to avoid designing these if possible
Procedures in Modules (2)

These are very much like *internal procedures*. Works very well in almost all programs.

Everything accessible in the *module* can also be used in the *procedure*.

Again, a *local name* takes precedence. But reusing the same name is very confusing.
Procedures in Modules (3)

MODULE thing
    INTEGER, PARAMETER :: temp = 123
CONTAINS
    SUBROUTINE pete ()
        INTEGER, PARAMETER :: temp = 456
        PRINT *, temp
    END SUBROUTINE pete
END MODULE thing

This will print 456, not 123
Avoid doing this as it's very confusing
Interfaces in Modules

The module can define just the interface. The procedure code is supplied elsewhere. The interface block comes before CONTAINS.

- Be absolutely sure they are consistent!

The interface and code are not checked.

Example: Cholesky decomposition
What Are Interfaces?

The **FUNCTION** or **SUBROUTINE** statement
And everything **directly connected** to that

Strictly, the **argument names** are not part of it
You are **strongly** advised to keep them the same

**Local variables** can be left out
Example

SUBROUTINE cholesky(A)       YES
  USE DOUBLE                      YES
  INTEGER :: j, n            NO
  REAL(KIND=dp) :: A(:,,:), X  YES for A
                                NO for X

  ...                                 

END SUBROUTINE cholesky           YES
Interfaces in Procedures

Can use an interface block as a declaration
Provides an explicit interface for a procedure

Can be used for ordinary procedure calls
But using modules is almost always better

• Essential for using certain specific features
e.g., keyword arguments within a module

Example: proc_as_arg

Generic procedure example: genericswap.f90
Accessibility (1)

Can separate exported from hidden definitions

Fairly easy to use in simple cases
  • Worth considering when designing modules

PRIVATE names are accessible only within the module (i.e., in module procedures after CONTAINS)

PUBLIC names are accessible by USE
This is commonly called exporting them
Accessibility (2)

They are just another attribute of declarations

MODULE fred
  REAL, PRIVATE :: array(100)
  REAL, PUBLIC :: total
  INTEGER, PRIVATE :: error_count
  CHARACTER(LEN=50), PUBLIC :: excuse
END MODULE fred
PUBLIC/PRIVATE statement sets the default
The default default is PUBLIC

MODULE fred
  PRIVATE
  REAL :: array(100)
  REAL, PUBLIC :: total
CONTAINS
  ...
END MODULE fred

Only TOTAL is accessible by a USE statement
Accessibility (4)

You can specify names in the statement. Especially useful for included names.

```
MODULE workspace
  USE double
  PRIVATE :: dp
  REAL(KIND=dp), DIMENSION(1000) :: scratch
END MODULE workspace
```

DP is no longer exported via workspace.
Partial Inclusion (1)

You can include only some names in USE

USE bigmodule, ONLY : errors, invert

Makes only errors and invert visible regardless of how many names bigmodule exports

Using ONLY is good practice
Makes it easier to keep track of uses

Can find out what is used where with grep
Partial Inclusion (2)

- One case when ONLY is strongly recommended: When using USE within modules
- All included names are exported
  Unless you explicitly mark them PRIVATE
  Perhaps only a few, perhaps dozens
- Ideally, use both ONLY and PRIVATE
  Almost always use at least one of them
- Another case when it is almost essential:
  If you don’t use IMPLICIT NONE liberally!
Partial Inclusion (3)

If you don’t restrict exporting and importing then a typing error could trash a module variable

Or forget that you had already used the name in another file far, far away...

• The resulting chaos is almost unfindable
From bitter experience in many years of Fortran!
Example (1)

MODULE settings
   INTEGER, PARAMETER :: DP = KIND(0.0D0)
   REAL(KIND=DP) :: Z = 1.0_DP
END MODULE settings

MODULE workspace
   USE settings
   REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
Example (2)

PROGRAM main
  IMPLICIT NONE
  USE workspace
  Z = 123
  ...
END PROGRAM main

• DP is inherited, which is okay
• Did you mean to update Z in settings?
• No problem if workspace had used ONLY : DP
Example (3)

The following are **better** and **best**

```fortran
MODULE workspace
  USE settings, ONLY : DP
  REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace

MODULE workspace
  USE settings, ONLY : DP
  PRIVATE :: DP
  REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
```
Renaming Inclusion (1)

You can rename a name when you include it

**WARNING**: this is footgun territory i.e., point gun at foot, pull trigger

This technique is sometimes *incredibly useful*

• But it is also *incredibly dangerous*

Use it only when you *really need to*
And even then *as little as possible*
Renaming Inclusion (2)

MODULE corner
    REAL, DIMENSION(100) :: pooh
END MODULE corner

PROGRAM house
    USE corner, sanders => pooh
    INTEGER, DIMENSION(20) :: pooh
    ...
END PROGRAM house

pooh is accessible under the name sanders
The name pooh is the local array
Why Is This Lethal?

MODULE one
   REAL :: X
END MODULE one

MODULE two
   USE one, Y => X
   REAL :: Z
END MODULE two

PROGRAM three
   USE one
   USE two
   !-- Both X and Y refer to the same variable!
Makefile Disclaimer

This course will give a brief overview of how to use make with Fortran.

Will cover the basics only!

Then look at how modules complicate the use of make.
What is Make?

**Make** is a **tool** which controls the generation of **executables** from a program’s **source** files.

It gets its knowledge of how to build your program from a file called the **makefile**.

The compilation procedure is much faster:

- The compilation is done with a **single command**.
- Only files that have been **modified** are recompiled.
- Allows managing **large programs** with lots of dependencies.
Makefile Basics (1)

A rule in the makefile tells Make how to execute a series of commands in order to build a target file from source files.

It also specifies a list of dependencies of the target file.

Here is what a simple rule looks like:

```
target:  dependencies ...
<tab> commands
```

The <tab> is absolutely necessary!
Makefile Basics (2)

Make uses timestamps to locate the files that have been modified since the last time make was executed.

By default when you type make it looks for the file makefile or Makefile. You can designate a specific name with make -f <thismakefile>.

Can also use macros to give names to variables within the makefile. NOTE these are case-sensitive!

If no specific target is given in the make command then Make starts with the first target listed in the makefile.

Let’s start with a very simple example (abc program).
Makefile Basics (3)

Comments are delimited by the `#` symbol.

A backslash `\` can be used as a continuation character.

Common extra tidbit: Create a “phony target” called `clean` which can be run to do a fresh recompile of all source code.
Makefile Automatic Variables

These can only values in the *recipe*. They cannot be used in the *target list* of a rule

$<$ The name of the first prerequisite

$^$ The names of the all prerequisites

$@$ The file name of the target of the rule

And there are even more available
Compiling Modules

When modules are compiled both a .o and .mod file are created.

A .mod file is like a compiled header. This is what the compiler searches for when it sees a USE statement.

The dependencies can start to get cumbersome and complicated when many modules are USED and inherited.

Make has no method for determining these for you.

Take a look at example2
Helpful Tools

**mkDepends** - generate a list of dependencies

**mkSrcfiles** - generate a list of all source files

Versions of these perl scripts are used in atmospheric models like **SAM** and **CAM**

**mkdep** - requires both GNU make and Python

**fmfmk.pl** - generate a makefile

**foraytool** - made especially for compiling large Fortran codes