Purpose

• First step in modular program design

• Cannot always anticipate all of the steps that will be needed to solve a problem
  – Easier to break problem up into a series of smaller steps
  – Subprograms can be written to implement each of these small steps.

• In the completely modular world, each subprogram has one and only one purpose.

• FORTRAN 90: Functions, modules, and subroutines
Functions

- Intrinsic, or library, functions and programmer-defined functions

- Programmer-defined function: Behave just like library functions when written.

- Function sub-programs
  
  function heading
  specification part
  execution part
  END FUNCTION statement
FUNCTION statement

- FUNCTION function-name (formal-argument list)
  OR
- type-identifier FUNCTION function-name (formal-argument list)

- Function-name: any legal Fortran identifier.
- Formal-argument list: identifier or a list of identifiers separated by commas
  - Formal or dummy arguments
  - Pass information to the function sub-program
- type-identifier: name of a type (REAL, INTEGER, etc.)
Specification/Execution Sections

• Same form as the specification part of a Fortran program plus:
  – The type of the function if this has not been included in the function heading.
  – The type of each formal argument.
    • INTENT specifier: tells how the arguments are to transfer information.

• Execution section has same form as Fortran program plus:
  – Include at least one statement that assigns a value to the identifier that names the function
    • Function-name = expression

• END FUNCTION function-name

• Aside: RETURN statement
  – RETURNS values of the function when executed.
  – Not necessary in Fortran 90, but is probably something you will run into.
Example: Temperature conversion

• Write a function to convert a temperature measured in degrees Fahrenheit into degrees Celsius.
  – $C = \frac{F - 32}{1.8}$

• REAL, INTENT(IN) :: Temperature
  – Temperature will only be used to transfer information into the function

• OK! Now we have this cool function, how do we use it?
Main program syntax

• This subprogram can be made accessible to the main program in three ways:

1. Placed in a subprogram section in the main program just before the END PROGRAM section (internal subprogram).

2. Placed in a module from which it can be imported into the program (module subprogram).

3. Placed after the END PROGRAM statement of the main program (external subprogram).
Internal subprogram

• Main program includes, just before END PROGRAM statement:

    CONTAINS
        subprogram_1
        subprogram_2
        subprogram_3

• Ok, let’s see the main program for our temperatures conversion program.
Method of Execution

• Main program as usual until the assignment statement containing the reference to the function.
• Actual argument ‘FahrenheitTemp’ is copied to ‘Temp’ argument in function.
• Control is transferred from the main program to the function subprogram, which begins execution.
• Assignment statement is evaluated using ‘Temp’
• Value computed is returned as the value of the function.
• Control is transferred back to the main program and the value of the function is assigned to ‘CelsiusTemp’.
• Execution continues on through the remainder of the main program.
INTENT(IN)

• When a function is referenced, the values of the actual arguments are passed to the function
  – Values are used in the calculation, but should not change during execution of the function.

• INTENT(IN) protects the corresponding actual argument by ensuring that the value of the formal argument cannot be changed during function execution.

• If not used, the value of the formal argument may be changed in the function and the value of the corresponding actual argument will also change.

• Number and type of actual arguments must agree with the number and type of formal arguments.

• NOTE: Local identifiers can be defined within the function, just as in the main program.
Scope

- May be several points where variables, constants, subprograms, types are declared
  - Main program, subprograms, modules.

- Scope: portion of program where these are visible, or where they are accessible and can be used.

- Fundamental Principle: The scope of an entity is the program or subprogram in which it is declared.
Rule #1

• An item declared within a subprogram is not accessible outside that subprogram.

• Item is ‘local’ to that subprogram.

• Item is ‘global’ if declared in the main program.
Rule #2

- A global entity is accessible throughout the main program and in any internal subprograms in which no local entity has the same name as the global item.

- Factorial example

- Warning: Although global variables can be used to share data between the main program and internal subprograms, it is usually unwise to do so.
  - Reduces the independence of the various subprograms making modular programming more difficult.
  - Changing a global variable in one part of a program changes it throughout the program, including all internal subprograms.

- Statement labels are not governed by scope rule #2.
  - FORMAT statements in the main program cannot be used within subprograms.

- IMPLICIT is global.
  - Not necessary to include it in these subprograms.
Saving values of local variables

• Values of local variables in sub-programs are not retained from one execution to the next, unless:
  – They are initialized in their declarations, or
  – They are declared to have the SAVE attribute.

• type, SAVE :: list-of-local variables

• SAVE list-of-local variables
  – If list is omitted, values of all variables will be saved.
External Subprograms

• Attached after the END PROGRAM statement of program unit.
  – Example: Temperature conversion revisited.

• Note #1: Function name is declared in the main program and subprogram.

• Note #2: Compiler may not be able to check references to subprogram.
  – Argument type, number of arguments, type of return value, etc.
Interface blocks

• Internal functions and modules have an ‘explicit interface’
  – Allows compiler to check arguments and results are returned correctly.

• For external subprograms, an ‘implicit interface’ must be provided for this functionality
  – Page 140 in text for syntax of interface block.
  – Looks like a function header in C or C++.
  – ‘interface block’ is same as function declarations within the actual function.

• Example: Temperature-conversion revisited, again.
Subroutines

- subroutine heading
  specification part
  execution part
END subroutine statement

- Specification and execution sections are the same as before.
Similar to Functions......

• Designed to perform particular tasks under control of some other program.

• Same basic form (heading, specification, execution, END).

• May be internal, module, or external.

• Scope rules apply.
yet different

• Functions are designed to return a single value.
  – Subroutines: several values or no value at all.

• Functions return values as function names.
  – Subroutines: return values via arguments.

• Functions are referenced through the function name.
  – Subroutines are referenced by a call statement.
Subroutines

- subroutine heading
  specification part
  execution part
END subroutine statement

- Specification and execution sections are the same as before.
Subroutine syntax

• Subroutine heading

  SUBROUTINE subroutine-name(formal-argument-list)

• End statement

  END SUBROUTINE subroutine-name

• That’s it. Now all you need to know is how to incorporate them into a program.
Using a subroutine

• CALL subroutine-name(actual-argument-list)
  – Arguments must match SUBROUTINE statement in number and type.
  – subroutine-name is not given a type like in functions.

• Examples
  – Displaying an angle in degrees.
  – Converting coordinates.
Argument association

• Coordinate conversion example.
  – R, Theta: Variables are only to be passed to them.
    • Not intended to return values.
  – INTENT(IN)

  – X, Y: Intended only to pass values back to the calling program unit
  – INTENT(OUT)

• INTENT(INOUT)
  – Used to pass information both to and from the subroutine.

• Note: Because both OUT and INOUT are intended to pass values back to calling program, the corresponding actual arguments must be variables!

• Read section 7.2 (subroutines and functions as arguments).
Modules

- Often similar calculations occur in a variety of applications.
  - Convenient to use the same sub-program in each of these applications.

- Module: a program unit used to package together type declarations and subprograms

  MODULE Name
  CONTAINS
    subprogram #1
    subprogram #2
    etc.
  END MODULE name

- Packages the subprograms, called module subprograms, together in a library that can be used in any other program unit.
Using a module

- Temperature-conversion library

- USE module-name
  - Placed at the beginning of the specification section of your main program.
  - All identifiers used in the specified module are imported into the program unit.

- USE module-name, ONLY: list
  - Only identifiers listed are imported.

- USE Temperature, ONLY: Fahr_to_Celsius
Translation to source program

• Two steps
  – Compilation
    • Source program is translated into an object file (.o extension)
  – Linking
    • References to functions contained in a module are linked to their definitions in that module
    • Creates the executable program

• Could take up to three steps
  1. Separate compilation of the program’s source file, creating an object file.
  2. Separate compilation of the module, creating a different object file.
  3. Linking the function calls in the program’s object file to the function definitions in the module’s object file.
    • Creates the executable program.
Examples

• Assume you have a module called temperature_library.f90 and a main program temperature_conversion.f90

  • gfortran temperature_library.f90 temperature_conversion.f90
  • gfortran temperaure_conversion.f90 temperature_library.f90? Still works…..
  • gfortran –c temperature_library.f90
gfortran temperature_library.o temperature_conversion.f90
  • gfortran –c temperature_library.f90
gfortran –c temperature_conversion.f90
gfortran temperature_library.o temperature_conversion.o

• Last examples used in ‘make’ files.
What are all these file types?

- Program file: contains your main program
- Module subprogram file: contains your function subprograms.
- Object file (.o): Machine language program.
- Executable: Finished (contains all links), executable program.
- Module (.mod): Meant to be a portable object, that doesn’t need to be recompiled.
  - Not always the case (more later)
Practice

- Take a *working* version of your CAPE/CIN program and put your function into a module.

- Compile and run your program to see that it works as advertised.