Goals for This Lecture:

- Understand the various kinds of REAL types
- Understand how to select precision in a processor independent manner
- Introduction to Makefiles

Kinds of REAL variables

- The default REAL type on most machines is a 32-bit (4 byte) real
- This usually offers only seven digits of precision and can usually only represent numbers between 10^{38} and 10^{-38}
- Such 4-byte real values are often referred to as single precision variables
- Most systems can support a 64-bit (8 byte), a.k.a. double precision, real type
- Always use double precision whenever possible!
  - We will do so throughout the remainder of this course!
- The precision can be specified with a `KIND=` parameter to the `real` declaration statement

```
real(kind=4) :: x_velocity
real(kind=8) :: acceleration
```

- There is no guarantee that either single or double precision types will have the same representation of different compilers or machines
  - On some machines double precision may be specified by `kind=8` while on others it might be specified by `kind=2`

Parameterizing your REAL declarations

- The kind of REAL type can be parameterized through the use of integer parameters contained in modules
- Makes it trivial to change type declarations by changing one or two lines in a module
- Example:

```
module real_kinds
  integer, parameter :: single_type=4
  integer, parameter :: double_type=8
end module real_kinds
program kind_example
use real_kinds
implicit none
real(kind=single_type) :: x_velocity
real(kind=double_type) :: acceleration
…
stop
end program kind_example
```

Parameterizing your REALs in a processor independent manner

- The fact the single and double precision real types can change from compiler to compiler is still a pain.
- Fortunately, FORTRAN provides an intrinsic function `kind` to determine the correct kind parameter based on the specification of a constant
- Example:

```
module real_kinds
  integer, parameter :: single_type=kind(1.0)
  integer, parameter :: double_type=kind(1.0d0)
end module real_kinds
program kind_example
use real_kinds
implicit none
real(kind=single_type) :: x_velocity
real(kind=double_type) :: acceleration
…
stop
end program kind_example
```

Always use double precision whenever possible!

Function Example: Midpoint Rule Program

- We can create a version of the midpoint rule program that uses functions instead of subroutines
- Create separate functions for upper & lower limits of integration as well as for integrand
- Function names must be declared with the appropriate type

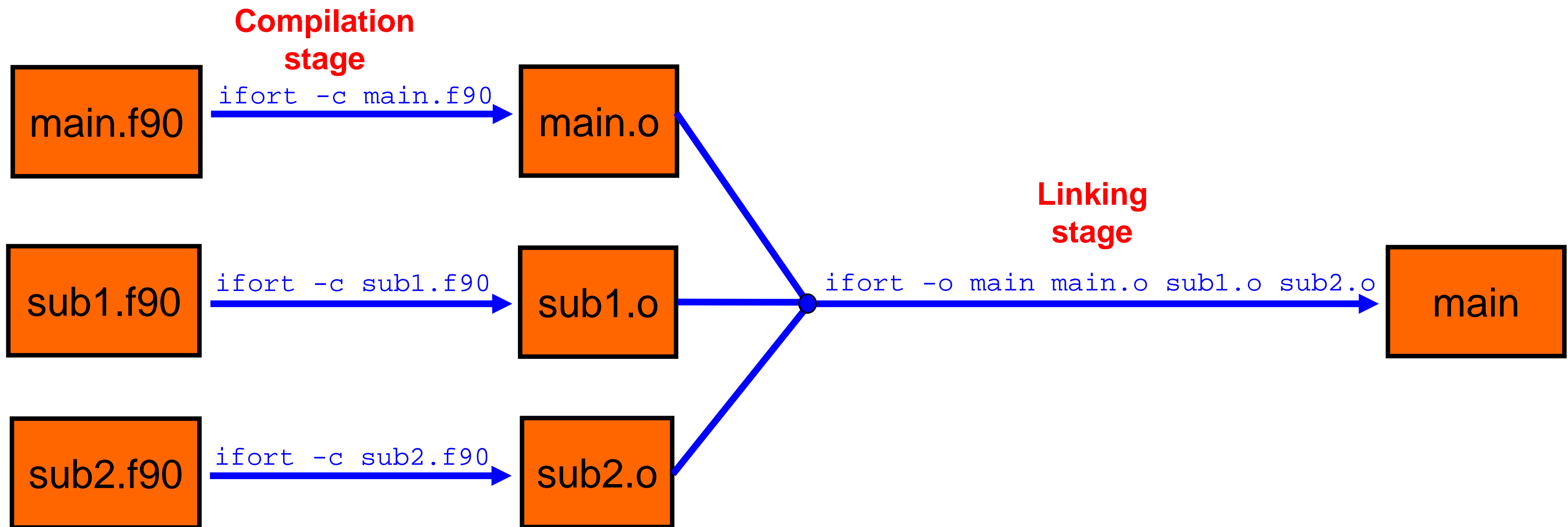
```
! Purpose:  Integrate a function via midpoint rule
! Author:   F. Douglas Swesty
! Date:     10/28/2005
program midpoint
implicit none      ! Turn off implicit typing
integer, parameter :: n=100   ! Number of subintervals
integer :: i          ! Loop index
real :: xlow, xhi  ! Bounds of integral
real :: dx            ! Variable to hold width of subinterval
real :: sum=0.0   ! Variable to hold sum
real :: xi            ! Variable to hold location of ith subinterval
real :: fi             ! Variable to value of function at ith subinterval
real :: upper_limit, lower_limit, integrand  ! Function subprogs

xhi_= upper_limit()        ! Obtain upper limit
xlow_= lower_limit()      ! Obtain lower limit

dx = (xhi-xlow)/(1.0*n)  ! Calculate with of subinterval

xi = xlow+0.5*dx             ! Initialize value of xi
do i = 1,n,1        ! Initiate loop
  fi =  integrand(xi)              ! Evaluate function at ith point
  sum = sum+fi*dx               ! Accumulate sum
  xi = xi+dx                         ! Increment location of ith point
enddo                ! Terminate loop
write(*,'(t2,a13,1x,1es12.5)') ' integral = ',integral
stop                                ! Stop execution of the program
end program midpoint
```

- Functions **upper_limit** and **lower_limit** supply the limits of integration.
- Function **integrand** evaluates the function at a specified point
- Good programming tip:
  - Use **implicit none** in function declaration section.
  - Declare function type in both definition and calling program
- Functions can be contained in modules and can access data in modules via use association

```
! Purpose:  Evaluate f(x) = x+abs(sin(2x)) and
!           supply limits of integration
! Author:   F. Douglas Swesty
! Date:     10/28/2005
real function upper_limit()
implicit none      ! Turn off implicit typing
upper_limit = 2.0*3.14159273   ! Return value
return
end function upper_limit
!-----------------------------------------------------
real function lower_limit()
implicit none       ! Turn off implicit typing
lower_limit = 0.0  ! Return value
return
end function lower_limit
!-----------------------------------------------------
real function integrand(x)
implicit none                      ! Turn off implicit
    typing
real, intent(in) :: x              ! Value of x
integrand = x+abs(sin(2.0*x))   ! Evaluate function
return
end function integrand
```

An introduction to Make

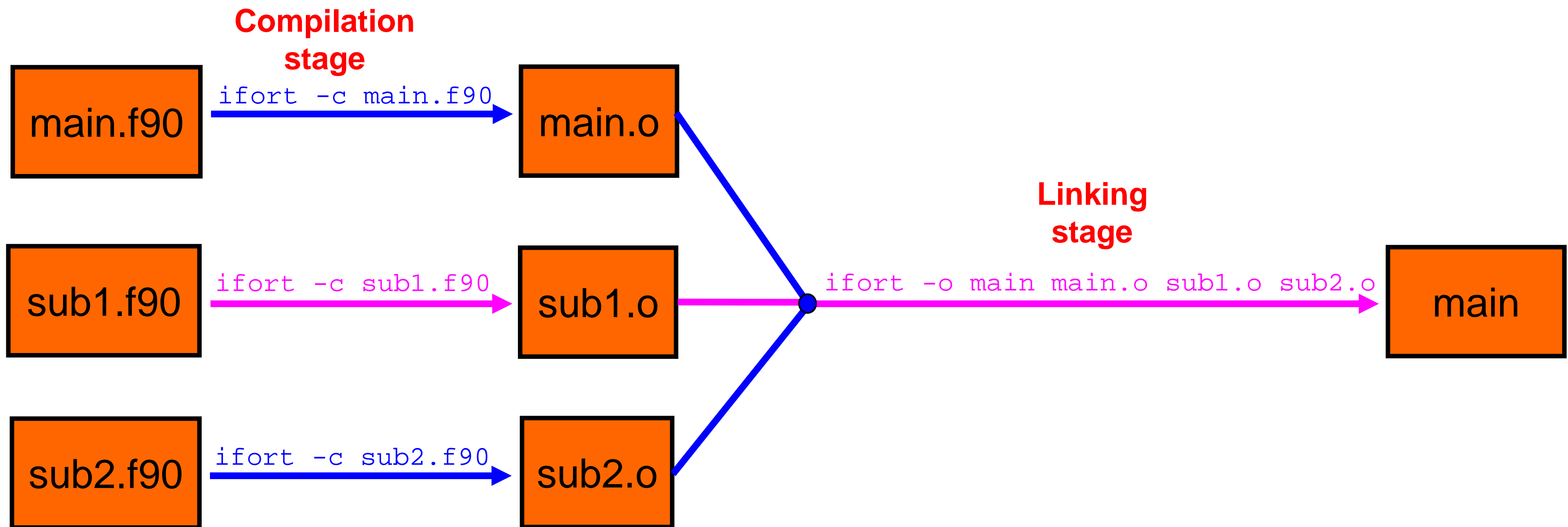
- The “make” command allows the management of programs consisting of multiple files
- Make keeps track of which files have been changed and only recompiles the portions that have changed
- Make exploits the separation of compilation and linking steps



- The make command recompiles only those modules that have changed

An introduction to Make

- Suppose you edit the code in sub1.f90 and want to produce an executable
- Only two steps are needed:
  - Recompilation of sub1.f90 to produce a new sub1.o file
  - Relinking of all the object files to produce the executable file “main”



- The make command automates this process

Dependencies

- Make operates on the principle of dependencies
- For example:

  main.o depends on: main.f90

  sub1.o depends on: sub1.f90

  sub2.o depends on: sub2.f90

  main depends on: main.o sub1.o sub2.o

- In order for make to work these dependencies must be specified in a Makefile

- The Makefile specifies the dependencies and how to create each file from the files it depend on.

- Once a Makefile has been created the building of the program becomes as simple as executing the command “make” on the command line.

An example Makefile

```
main: midpoint.o upper_limit.o lower_limit.o integrand.o
          ifort -o main main.o sub1.o sub2.o
main.o: main.f90
          ifort -c main.f90
sub1.o: sub1.f90
          ifort -c sub1.f90
sub2.o: sub2.f90
          ifort -c sub2.f90
```

- The make command looks for a file named Makefile (or makefile) in the current working directory
- The make command starts with the "main" program in the first line and checks to see if the dependencies, i.e. the files that main depends on (main.o sub1.o and sub2.o), are newer than the executable file main.
- If they are then the command `ifort -o main main.o sub1.o sub2.o` is executed
- The dependencies of main.o sub1.o and sub2.o are each checked in turn

Makefile rules

- Each stanza of the Makefile has the following form:

```
target: dependencies
        command
```
(command preceded by a tab character)

- Target can be a name of a file that will be created when one of it dependent files changes
  - Targets can be things other than file names as we will see shortly

- Dependencies is a list (separated by spaces) of files which the target depends on

- Command is a series of command that are need to accomplish a task (usually the creation of the target file from the dependency files)

Executing make

- Make will try to execute the commands associated with first target listed in the Makefile

- If necessary, make will execute additional targets associated with the dependency files

- You can also have make execute a specific target by giving make an argument
- Example:

```
> make sub1.o
```

- It is common to create a "clean" target in a Makefile to cleanup after the compilation process

```
clean:
      /bin/rm main.o sub1.o sub2.o
```

`>make clean` executes the "clean" target

Makefile macros

- make allows the definition of macros (think “variables”) which simplify the construction and maintainence of makefiles
- Changing compiler flags for an entire code is simplified to changing a single line

```
F90 = ifort -c
LINK = ifort -o
main: midpoint.o upper_limit.o lower_limit.o integrand.o
          $(LINK) main main.o sub1.o sub2.o
main.o: main.f90
          $(F90) main.f90
sub1.o: sub1.f90
          $(F90) sub1.f90
sub2.o: sub2.f90
          $(F90) sub2.f90
```

Reading Assignment

- Read the "Introduction to make" tutorial linked on the course home page
- Read Chapter 1 of Savitch