Functions in Fortran

Victor Eijkhout, Susan Lindsey

Fall 2019

Subprogram basics

Subprograms in contains clause

```
Program foo
  < declarations>
  < executable statements >
  Contains
    < subprogram definitions >
End Program foo
```

Subroutines

```
subroutine foo()
  implicit none
  print *,"foo"
  if (something) return
  print *,"bar"
end subroutine foo
```

- Looks much like a main program
- Ends at the end, or when `return` is reached
- Note: `return` does not return anything
- Activated with

```
call foo()
```

Subroutine with argument

Code:

```
program printing
  implicit none
  call printint(5)
contains
  subroutine printint(invalue)
    implicit none
    integer :: invalue
    print *,invalue
  end subroutine printint
end program printing
```

Output
[funcf] printone:

```
5
```

Subroutine can change argument

Code:

```
program adding
  implicit none
  integer :: i=5
  call addint(i,4)
  print *,i
contains
  subroutine addint(inoutvar,addendum)
    implicit none
    integer :: inoutvar,addendum
    inoutvar = inoutvar + addendum
  end subroutine addint
end program adding
```

Output
[funcf] addone:

```
9
```

Function vs Subroutine

Subroutines can only ‘return’ results through their parameters.

Functions have an actual return result.

Function example

Code:

```
program plussing
  implicit none
  integer :: i
  i = plusone(5)
  print *,i
contains
  integer function plusone(invalue)
    implicit none
    integer :: invalue
    plusone = invalue+1 ! note!
  end function plusone
end program plussing
```

Output
[funcf] plusone:

6

Function definition and usage

- `subroutine` vs `function`:
  compare *`void`* functions vs non-void in C++.
- Return type, keyword `function`, name, parameters
- Function body has statements
- Result is returned by assigning to the function name
- Use: `y = f(x)`

Why a 'contains' clause?

```
Program NoContains
  implicit none
  call DoWhat()
end Program NoContains

subroutine DoWhat(i)
  implicit none
  integer :: i
  i = 5
end subroutine DoWhat
```

Warning only, crashes.

```
Program ContainsScope
  implicit none
  call DoWhat()
contains
  subroutine DoWhat(i)
    implicit none
    integer :: i
    i = 5
  end subroutine DoWhat
end Program ContainsScope
```

Error, does not compile

Why a 'contains' clause, take 2

Code:

```
Program NoContainTwo
  implicit none
  integer :: i=5
  call DoWhat(i)
end Program NoContainTwo

subroutine DoWhat(x)
  implicit none
  real :: x
  print *,x
end subroutine DoWhat
```

Output

[funcf] nocontaintype:

```
7.00649232E-45
```

At best compiler warning if all in the same file

For future reference: if you see very small floating point numbers, maybe you have made this error.

Exercise 1

Write a program that asks the user for a positive number; negative input should be rejected. Fill in the missing lines in this code fragment:

Code:

```
program readpos
  implicit none
  real(4) :: userinput
  print *,"Type a positive number:"
  userinput = read_positive()
  print *,"Thank you for",userinput
contains
  real(4) function read_positive()
    implicit none
  /* ... */
  end function read_positive
end program readpos
```

Output
[funcf] readpos:

```
Type a positive number:
No, not  -5.00000000
No, not   0.00000000
No, not  -3.14000010
Thank you for   2.48000002
```

Subprogram arguments

Arguments are declared in subprogram body:

```
subroutine f(x,y,i)
  implicit none
  integer,intent(in) :: i
  real(4),intent(out) :: x
  real(8),intent(inout) :: y
  x = 5; y = y+6
end subroutine f
! and in the main program
call f(x,y,5)
```

declaring the 'intent' is optional, but highly advisable.

Fortran nomenclature

The term dummy argument is what Fortran calls the parameters in the subprogram definition. The arguments in the subprogram call are the actual arguments.

Parameter passing

- Everything is passed by reference.
  Don't worry about large objects being copied.
- Optional intent declarations:
  Use `in`, `out`, `inout` qualifiers to clarify semantics to compiler.

Intent checking

Compiler checks your intent against your implementation. This code is not legal:

```
subroutine ArgIn(x)
  implicit none
  real,intent(in) :: x
  x = 5 ! compiler complains
end subroutine ArgIn
```

Why intent checking?

Self-protection: if you state the intended behaviour of a routine, the compiler can detect programming mistakes.

Allow compiler optimizations:

```
x = f()
call ArgOut(x)
print *,x
```

Call to f removed

```
do i=1,1000
  x = ! something
  y1 = .... x ....
  call ArgIn(x)
  y2 = ! same expression as y1
```

y2 is same as y1 because x not changed

(May need further specifications, so this is not the prime justification.)

Exercise 2

Write a subroutine `trig` that takes a number α as input and passes $\sin\alpha$ and $\cos\alpha$ back to the calling environment.

Exercise 3

Take your prime number testing function `test_if_prime`, and use it to write a program that prints multiple primes:

- Read an integer `how_many` from the input, indicating how many (successive) prime numbers should be printed.
- Print that many successive primes, each on a separate line.
- (Hint: keep a variable `number_of_primes_found` that is increased whenever a new prime is found.)

Turn it in!

- If you have compiled your program, do:

  `sdstestprimef yourprogram.F90`

  where '`yourprogram.F90`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `sdstestprimef -s yourprogram.F90`

  where the `-s` flag stands for 'submit'.
- If you don't manage to get your code working correctly, you can submit as incomplete with

  `sdstestprimef -i yourprogram.F90`