

Fortran95 and Fortran2003

Tips and Techniques to Help Build
Robust, Maintainable Code

2. What can Modules do for you?

Paul van Delst

Betty Petersen Memorial Library
Technical Seminar Series

Introduction

- Modules are a “new” program unit introduced in Fortran90.
- Goal of this seminar is to use modules as vehicle for explanation of Fortran95/2003 features.

Overview

- What we'll be covering this time
 - Software construction concepts
 - Concepts of “scope” and “association”
 - Explicit and implicit interfaces
 - What's the difference?
 - Why explicit interfaces are A Good Thing.™
 - Typical application of modules.
 - User defined generic procedures
 - User defined operators
 - If there's time, a brief introduction to Fortran2003 object-oriented programming capabilities and syntax.
- A lot of material is not specific to modules, but we'll discuss it in that context.
- As always, please ask questions for clarification.

Software Construction Concepts

- Abstraction
- Encapsulation
- Inheritance
- Information hiding
- Coupling
- Cohesion

Ability to engage with a
concept while abstraction
is left off. It defines the level
of detail you handle and different
properties in a specific
context are revealed.
Describes different levels.
Refers to how closely related
things are.
Allows you to hide
things, support a central
purpose - how focused
they are.

- All used to *manage complexity*.

Modules

- Modules allow you to package data and procedure specifications
 - They serve the following needs:
 - A reliable mechanism for specifying global data (variables, type definitions, procedure interfaces)
 - Facilitate information hiding
 - Provide explicit interfaces for procedures and thus reduce argument mismatch errors
 - Facilitate implementation of object-oriented (OO) concepts
- [§11.3 Fortran2003 Handbook]

Basic Anatomy of a Module

```
MODULE module_name
  [specification-part]
  [CONTAINS
    module-subprogram-1
    [module-subprogram-2]
    .
    .
    [module-subprogram-n]
  ]
END MODULE module-name
```

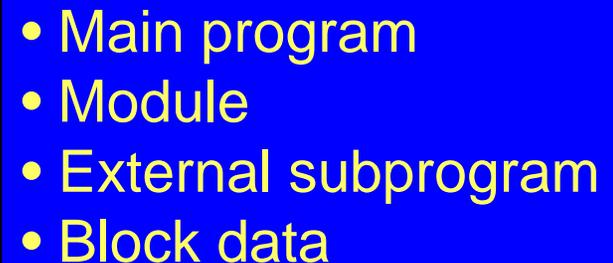
- USE statements (first)
- IMPLICIT NONE (please!)
- Declarations
 - Module parameters
 - Module variables
 - Derived type definitions
 - Interface blocks

Module subprograms are FUNCTIONS or SUBROUTINES. They can CONTAIN their own internal subprograms.

The part of the module for specifying subprograms is optional, i.e. modules can contain only specifications.

Scope

- The scope of a program entity is the part of the program in which that entity is known, is available, and can be used. [§2.3.3 Fortran2003 Handbook]
- In FORTRAN77, scope was defined in terms of program units.
- In Fortran90+, scope is defined in terms of “scoping units”:
 - Program unit or subprogram,
 - Derived type definition,
 - Interface body.

- 
- Main program
 - Module
 - External subprogram
 - Block data

Scope Example (1)

MODULE my_define	Scoping unit 4
INTEGER, PARAMETER :: N=5 ←	Scoping unit 4
TYPE :: my_type	Scoping unit 1
INTEGER :: n=2 ←	Scoping unit 1
REAL :: x=0.0,y=0.0	Scoping unit 1
END TYPE	Scoping unit 1
CONTAINS	Scoping unit 4
FUNCTION module_func(a) RESULT(b)	Scoping unit 3
TYPE(my_type), INTENT(IN) :: a	Scoping unit 3
TYPE(my_type) :: b	Scoping unit 3
INTEGER :: n = 3 ←	Scoping unit 3
... <i>procedure body</i> ...	Scoping unit 3
CONTAINS	Scoping unit 3
SUBROUTINE internal_sub	Scoping unit 2
INTEGER :: n ←	Scoping unit 2
... <i>procedure body</i> ...	Scoping unit 2
END SUBROUTINE internal_sub	Scoping unit 2
END FUNCTION module_func	Scoping unit 3
END MODULE my_define	Scoping unit 4

The various variables and parameters with the name “n” are *different* entities.

Scope Example (2)

```
PROGRAM demo_scope
```

```
  INTEGER :: n=5
```

```
  INTERFACE
```

```
    SUBROUTINE external_sub(n)
```

```
      INTEGER, INTENT(IN) :: n
```

```
    END SUBROUTINE external_sub
```

```
  END INTERFACE
```

```
  ...program body...
```

```
END PROGRAM demo_scope
```

```
SUBROUTINE external_sub(i)
```

```
  INTEGER, INTENT(IN) :: i
```

```
  INTEGER :: n = 4
```

```
  ...procedure body...
```

```
CONTAINS
```

```
  SUBROUTINE internal_sub
```

```
    INTEGER :: n
```

```
    ...procedure body...
```

```
  END SUBROUTINE internal_sub
```

```
END SUBROUTINE external_sub
```

Scoping unit 4

Scoping unit 4

Scoping unit 4

Scoping unit 1

Scoping unit 1

Scoping unit 1

Scoping unit 4

Scoping unit 4

Scoping unit 4

Interface block

Scoping unit 3

Scoping unit 2

Scoping unit 2

Scoping unit 2

Scoping unit 2

Scoping unit 3

D.R.Y. : Don't Repeat Yourself!

Association

- The concept that is used to describe how different entities in the same scoping unit or different scoping units can share values and other properties. [§2.3.4 Fortran2003 Handbook]
- Fortran95 associations:
 - Use association
 - Host association
 - Pointer association
- Fortran2003 associations:
 - Inheritance association (OO stuff)
 - Linkage association (C Interop)
 - Construct association (`SELECT TYPE, ASSOCIATE`)
- Storage association won't be covered.

We'll be concentrating on these this time.

Use Association

- Use association makes entities defined in modules accessible via the (surprise) `USE` statement.
- `USE` statements must be the first statements in a specification part.
- Some syntax examples:

```
USE modname
```

Use all public entities from the module.

```
USE modname, my_x=>x
```

```
USE modname, ONLY:x,y,z
```

Use *only* the public entities `x`, `y`, and `z` from the module.

Use all public entities from the module, but rename the module entity `x` with a *local* name of `my_x`.

Host Association

- Host association permits entities in a host scoping unit to be accessible in an internal subprogram, module subprogram, or derived type definition.
- No mechanism for renaming entities
- In Fortran95, an interface body does not access its environment by host association.
- Fortran2003 solved this problem via the `IMPORT` statement. Only for interface bodies. E.g.

```
INTERFACE
  FUNCTION func(f)
    IMPORT :: t, fp
    TYPE(t) :: func
    REAL(fp) :: f
  END FUNCTION func
END INTERFACE
```

The derived type definition `t` and the kind type `fp` from the host module are `IMPORTED` into the interface body.

Explicit Interfaces

- For calls to internal subprograms, compilers have access to its interface;
 - Is it a function or subroutine?
 - The names and properties of its arguments
 - Properties of the result if it's a function.
- This allows the compiler to check if the actual and dummy arguments match as they should.
- In this case we say the interface is *explicit*.
- Similarly for calls to module subprograms.
- By default, module subprograms have explicit interfaces.

Implicit Interfaces

- For calls to external subprograms, compilers typically do not have access to the code;
 - Same file, but different program unit.
 - Completely different file.
- The calling code knows nothing about the interface, e.g. argument type or rank.
- Here we say the interface is *implicit*.
- These interfaces can be made explicit via interface blocks.
- FORTRAN77 interfaces are always implicit.

Why are explicit interfaces good?

- Argument checking.
 - Type, Kind, and Rank are known at compile time.
- Assumed shape dummy arguments.
- Function result that is an array, pointer, or **allocatable**.
- Function result that is dynamically sized.
- **ELEMENTAL** attribute.
- Some dummy argument attributes require it.
 - **ALLOCATABLE**, **ASYNCHRONOUS**, **OPTIONAL**, **POINTER**, **TARGET**, **VALUE**, or **VOLATILE**
- **Dummy argument of parameterised derived type.**
- **Polymorphic dummy argument.**
- **BIND** attribute

Argument Checking

- Let's say you have a module subprogram:

```
SUBROUTINE sub(i)
  INTEGER :: i
```

...

```
END SUBROUTINE sub
```

- And let's say you call it like so,

```
REAL :: x
```

...

```
CALL sub(x)
```

- The compiler will flag an error because the interfaces don't match.
- Depending on your application, this may or may not be what you want.
- Deliberate argument mismatching like that above (e.g. assuming storage association) is discouraged.

Assumed Shape Dummy Arguments

- Once again, let's say you have a module subprogram:

```
FUNCTION func(x)
  REAL :: x(:, :)
  ...
END FUNCTION func
```

Data structure used to hold information about a data object.

- For an array actual argument, the compiler needs to know whether to pass a “dope vector” or a storage address.

- Which one depends on the dummy declaration:
 - Assumed shape → dope vector
 - Explicit shape or assumed size → storage address
- With assumed shape, you can query the dummy. Let's say you passed an actual argument `arr(0:5:2, 1:6:3)`,
`n = SIZE(x)` Total number of elements? 6
`PRINT *, SHAPE(x)` The array shape? (/ 3, 2 /) or [3, 2]

Dynamically Sized Function Result

- In Fortran90+, functions can return array results.
- What if you want your result to be the same size as your input?
- We use assumed shape dummy arguments and the RESULT clause.
- Once again, let's say you have a module subprogram:

```
FUNCTION func(x) RESULT(y)
  REAL, INTENT(IN) :: x(:, :)
  REAL :: y(SIZE(x, DIM=1), SIZE(x, DIM=2))
  ...
END FUNCTION func
```

- The result will be the same size as your input. And you call it like you would any other function,

```
REAL :: a(4,7), b(4,7)
...
b = func(a)
```

- The compiler will also check for conformance of the *rank* of the result with the actual argument. But not size.

ELEMENTAL *Attribute* (1)

- What if you want your array-valued result function to work for rank-3 input also? Or rank-4? Just like intrinsic functions do (e.g. SIN, EXP, etc)
- Fortran95 introduced the ELEMENTAL prefix.
- Elemental procedures are defined with scalar dummy arguments, but may be referenced with actual arguments that are of any rank, provided they are conformable.
- So, our module subprogram would become simply:

```
ELEMENTAL FUNCTION func(x) RESULT(y)
  REAL, INTENT(IN) :: x
  REAL :: y
  ...
END FUNCTION func
```
- Elemental procedures automatically have the PURE attribute.

ELEMENTAL *Attribute* (2)

- The Simple Set of Rules for PURE procedures:
 - If a function, does not alter dummy arguments,
 - Does not alter any part of a variable accessed by use or host association,
 - Contains no local variables with the SAVE attribute,
 - Performs no operation on an external file,
 - Contains no STOP statement.
- The Simple Set of Additional Rules for ELEMENTAL procedures:
 - It must not be recursive,
 - Dummy arguments must be a nonpointer, nonallocatable, scalar data object,
 - The result of an elemental function must be scalar and not a pointer or allocatable,
 - Dummy arguments must not be used in a specification expression (the exceptions are beyond the scope of this slide.)

Typical Applications of Modules

- Modules provide a way of packaging:
 - Data
 - User-defined types
 - User-defined operators
 - Data abstraction
 - Encapsulation
 - Procedure libraries
- [§11.3.9 Fortran2003 Handbook]

Packaging Data (1)

```
MODULE Type_Kinds
```

```
  IMPLICIT NONE
```

```
  PRIVATE
```

```
  PUBLIC :: Byte, Short, Long
```

```
  PUBLIC :: Single, Double
```

```
  ! Integer types
```

```
  INTEGER, PARAMETER :: Byte =SELECTED_INT_KIND(1)
```

```
  INTEGER, PARAMETER :: Short=SELECTED_INT_KIND(4)
```

```
  INTEGER, PARAMETER :: Long =SELECTED_INT_KIND(8)
```

```
  ! Floating point types
```

```
  INTEGER, PARAMETER :: Single=SELECTED_REAL_KIND(6)
```

```
  INTEGER, PARAMETER :: Double=SELECTED_REAL_KIND(15)
```

```
END MODULE Type_Kinds
```

Set default visibility to PRIVATE and specifically list PUBLIC entities.

Define the data entities you want to “package”.

Packaging Data (2)

```
MODULE Shared_Data
```

```
USE Type_Kinds, ONLY:fp=>Double
```

```
IMPLICIT NONE
```

```
REAL(fp), ALLOCATABLE, SAVE :: x(:), y(:)
```

```
REAL(fp), ALLOCATABLE, SAVE :: z(:, :)
```

```
END MODULE Shared_Data
```

```
PROGRAM Demo_Shared_Data
```

```
CALL Load_Data()
```

```
CALL Display_Data()
```

```
END PROGRAM Demo_Shared_Data
```

ONLY clause limits entities that are visible. Aliasing symbol => used in rename.

Data to be shared among USING modules or programs. But why the SAVE attribute?

The external subprograms USE the Shared_Data module. What happens to the data upon return from the Load_Data() subprogram without SAVE?

Adapted from Redwine, C., "Upgrading to Fortran90"

Packaging Derived Types

```
MODULE Rational_Define
```

```
USE Type_Kinds, ONLY: Long
```

```
IMPLICIT NONE
```

```
PRIVATE
```

```
PUBLIC :: Rational_type, Long
```

```
! Derived type definition
```

```
TYPE :: Rational_type
```

```
INTEGER :: num, denom
```

```
END TYPE Rational_type
```

```
END MODULE Rational_Define
```

Modules can USE other modules. ONLY clause limits entities made visible.

Explicit visibility. Note that Long is “passed through”.

Derived type definition. Any program unit gains access to this type definition via a USE Rational_Define statement.

Adapted from Redwine, C., “Upgrading to Fortran90”

Packaging Related Operations

```
MODULE Rational_Operators
```

```
USE Rational_Define
```

```
IMPLICIT NONE
```

```
PRIVATE
```

```
PUBLIC :: Mult, Add
```

```
CONTAINS
```

```
! Product of two rational numbers
```

```
FUNCTION Mult(r1,r2) RESULT(prod)
```

```
TYPE(Rational_type), INTENT(IN) :: r1,r2
```

```
TYPE(Rational_type) :: prod
```

```
prod%num =r1%num *r2%num
```

```
prod%denom=r1%denom*r2%denom
```

```
END FUNCTION Mult
```

```
! Sum of two rational numbers
```

```
FUNCTION Add(r1,r2) RESULT(sum)
```

```
...
```

```
END FUNCTION Add
```

```
END MODULE Rational_Operators
```

Module usage and explicit visibility.

Note RESULT clause for functions.

Adapted from Redwine, C., "Upgrading to Fortran90"

Abstraction (1)

- What is it *really*?
- As mentioned earlier, it's the ability to engage with a concept while safely ignoring some of its details.
- In the context of this seminar, it's a fancy term for the practice of combining derived type definitions and their related operators in the *same* package.
- It allows you to deal with complexity at different levels, while ignoring many of the implementation details.

Abstraction (2)

```
MODULE Rational_Define
```

```
  IMPLICIT NONE
```

```
  PRIVATE
```

```
  PUBLIC :: OPERATOR(*), OPERATOR(+)
```

```
  TYPE :: Rational_type
```

```
    INTEGER :: num, denom
```

```
  END TYPE Rational_type
```

```
  INTERFACE OPERATOR (*)
```

```
    MODULE PROCEDURE Mult
```

```
  END INTERFACE
```

```
  INTERFACE OPERATOR (+)
```

```
    MODULE PROCEDURE Add
```

```
  END INTERFACE
```

```
CONTAINS
```

```
  FUNCTION Mult(r1,r2) RESULT(prod)
```

```
    ...
```

```
  FUNCTION Add(r1,r2) RESULT(sum)
```

```
    ...
```

```
END MODULE Rational_Define
```

Defining visibility
of overloaded
operators.

Use of an interface
block to *overload* the
intrinsic operators '*'
and '+' to work with
derived types.

Abstraction (3)

Compare a “traditional” approach:

```
PROGRAM Demo_Rational_1
  USE Rational_Operators
  IMPLICIT NONE
  TYPE(Rational_type) :: frac1,frac2,prod,sum
  ...
  prod = Mult(frac1,frac2)
  sum  = Add(frac1,frac2)
END PROGRAM Demo_Rational_1
```

With one in which abstraction is used:

```
PROGRAM Demo_Rational_2
  USE Rational_Define
  IMPLICIT NONE
  TYPE(Rational_type) :: frac1,frac2,prod,sum
  ...
  prod = frac1 * frac2
  sum  = frac1 + frac2
END PROGRAM Demo_Rational_2
```

Abstraction (4)

- Who has compared floating point numbers?
- Rather than
 `IF (x==y) THEN...`
we do something like,
 `IF (ABS (x-y) < tolerance) THEN...`
- What tolerance value? Typically depends on magnitudes of `x` and `y`.
- How to dynamically select a tolerance in a comparison operator?

Abstraction (5)

```
MODULE Compare_Float_Numbers
  PRIVATE
  PUBLIC :: OPERATOR (.EqualTo.)
  INTERFACE OPERATOR (.EqualTo.)
    MODULE PROCEDURE EqualTo_Single
    MODULE PROCEDURE EqualTo_Double
  END INTERFACE OPERATOR (.EqualTo.)
CONTAINS
  ELEMENTAL FUNCTION EqualTo_Single(x,y) RESULT(EqualTo)
    REAL(Single), INTENT(IN) :: x, y
    LOGICAL :: EqualTo
    EqualTo = ABS(x-y) < SPACING( MAX(ABS(x),ABS(y)) )
  END FUNCTION EqualTo_Single
  ELEMENTAL FUNCTION EqualTo_Double(x,y) RESULT(EqualTo)
    REAL(Double), INTENT(IN) :: x, y
    LOGICAL :: EqualTo
    EqualTo = ABS(x-y) < SPACING( MAX(ABS(x),ABS(y)) )
  END FUNCTION EqualTo_Double
END MODULE Compare_Float_Numbers
```

Scalar or any rank arguments and result.

Abstraction (6)

Again, a “traditional” approach:

```
PROGRAM Demo_Compare_Float_1
  IMPLICIT NONE
  REAL(Single) :: xs,ys,tols=1.0e-07_Single
  REAL(Double)  :: xd,yd,told=1.0e-15_Double
  ...
  IF (ABS(xs-ys)<tols) THEN...
  IF (ABS(xd-yd)<told) THEN...
END PROGRAM Demo_Compare_Float_1
```

Compared to one with some abstraction:

```
PROGRAM Demo_Compare_Float_2
  USE Compare_Float_Numbers
  IMPLICIT NONE
  REAL(Single) :: xs,ys
  REAL(Double)  :: xd,yd
  ...
  IF (xs.EqualTo.ys) THEN...
  IF (xd.EqualTo.yd) THEN...
END PROGRAM Demo_Compare_Float_2
```

Encapsulation (1)

- Encapsulation explicitly specifies what level of detail is made available.
- It allows you to manage complexity by forbidding you to look at the complexity.
- Along with abstraction, it allows you to hide the implementation details that are not necessary to use something.
- Think of a car. When you drive one do you:
 - Set the timing of the valves and pistons?
 - Modify how the fuel/air mix is injected into the cylinders?
 - Adjust the suspension when the road gets bumpy?

No. All those complex systems are encapsulated - depending on the age of your car, you can only affect them by how you choose to drive...which is why you have a car.

Encapsulation (2)

```
TYPE :: MyType_type
  INTEGER :: n_Allocates, n_x, n_y
  REAL, ALLOCATABLE :: x(:), y(:), z(:, :)
END TYPE MyType_type
```

All components
are public.

```
TYPE :: MyType_type
  PRIVATE
  INTEGER :: n_Allocates, n_x, n_y
  REAL, ALLOCATABLE :: x(:), y(:), z(:, :)
END TYPE MyType_type
```

All components
are private.

```
TYPE :: MyType_type
  PRIVATE
  INTEGER :: n_Allocates
  INTEGER, PUBLIC :: n_x, n_y
  REAL, ALLOCATABLE :: x(:), y(:), z(:, :)
END TYPE MyType_type
```

Some components
are public, others
are private.

Encapsulation (3)

- With our derived data type fully private, what functionality might we need in our module?
 - Must haves:
 - Ability to allocate the internals,
 - Ability to assign x, y, and z values to their respective components,
 - Ability to retrieve the x, y, and z values.
 - Would be nice:
 - Ability to combine data from different instances of data types?
 - Ability to search data?
 - Arithmetic operations?
 - etc.
- Users can interact with the data type *only* via the functionality provided in the module.
- All the internal implementation details are hidden.
- In OO-speak, the derived data type is like a “class”, and the module subprograms are like its “methods”.

Encapsulation (4)

- What's so great about our encapsulated setup?
 - Users don't have to worry about the details.
 - They just interact with the data type as needed.
 - Functionality can be easily added.
 - Testing is much, much easier. You test each little bit as you go.
- What's not so great?
 - A lot more overhead writing software.
 - Users might not have the functionality they need.
 - Scientists are a funny lot...some simply want access to the guts of things (which is why Windows completely bamboozles me 😊)
- Sometimes a compromise works. For example:
 - The CRTM adopted a convention where the various datatype internals are *not* private.
 - Avoids the overhead of `Get ()` and `Set ()` type of functions.
 - `Allocate`, `Destroy`, `Assign`, `Equal`, `Info` procedures are provided.

Fortran2003 OOP

- Type extension
- Polymorphic entities
- ASSOCIATE and SELECT TYPE constructs
- Type-bound procedures
- Abstract types
- Finalization
- Type inquiry intrinsics

Type Extension (1)

Type definitions

```
TYPE :: base_type
  REAL :: a
  INTEGER :: i
END TYPE base_type
```

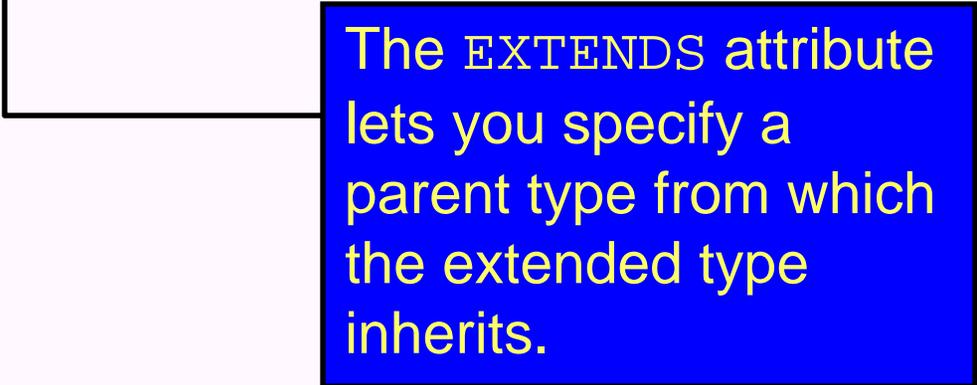
```
TYPE, EXTENDS(base_type) :: char_type
  CHARACTER(20) :: c
END TYPE char_type
```

Variable definitions

```
TYPE(base_type) :: x
TYPE(char_type) :: y
```

Usage

```
x%a=1.0; x%i=3
y%a=3.14159; y%i=7; y%c='pi'
```



The EXTENDS attribute lets you specify a parent type from which the extended type inherits.

Type Extension (2)

```
module jon
  type base_type
    real    :: a = 1.0
    integer :: i = 4
  end type base_type
  type, extends(base_type) :: char_type
    character(20) :: c = 'woohoo'
  end type char_type
end module jon
```

```
program test_jon
  use jon
  type(base_type) :: x
  type(char_type) :: y
  print *, y%base_type
  print *, y%a, y%i
end program test_jon
```

```
pooter:~/scratch $ gfortran --std=f2003 jon.f90
pooter:~/scratch $ a.out
1.00000000      4
1.00000000      4
```

The base type components are accessible via *either* the `y%base_type` reference, or `y%a`, `y%i`

Summary

- What module related topics we've looked at:
 - Software construction concepts
 - Concepts of “scope” and “association”
 - Explicit and implicit interfaces
 - Typical application of modules.
- What to do with the information?
 - How can the concepts be applied in your current work
- Was the information useful?
 - Let me know: `paul.vandelst@noaa.gov`

Next Time

- Object-oriented programming?
- C-Interoperability?
- Or...?
- What would you like to see in any future Fortran95/2003 seminars?

Where to Get More Information

- Fortran Forum?
- I would like to get a bulletin-board type of forum set up on the library website.
- IT issues are getting in the way.
- Talk to your colleagues and supervisor about it - maybe with more managerial support, it will happen sooner.
- This building contains a wealth of expertise.

Where to Get More Information

- `comp.lang.fortran` newsgroup.
 - Many members of the current and past standards committee, and some compiler vendors, frequent it.
- Books!
 - I recommend people have a read of “Code Complete” by Steve McConnell.
 - He also has a website:
<http://cc2e.com>

