The Use and I:
Transitivity of Module Uses and its Impact

Lydia Duncan, Cray Inc.

CHIUW 2016

May 27th, 2016

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

User Features

The Use and I: Transitive Uses

- **Symbols visible to B via a 'use' also visible:**
  - within C's main
  - other uses of B

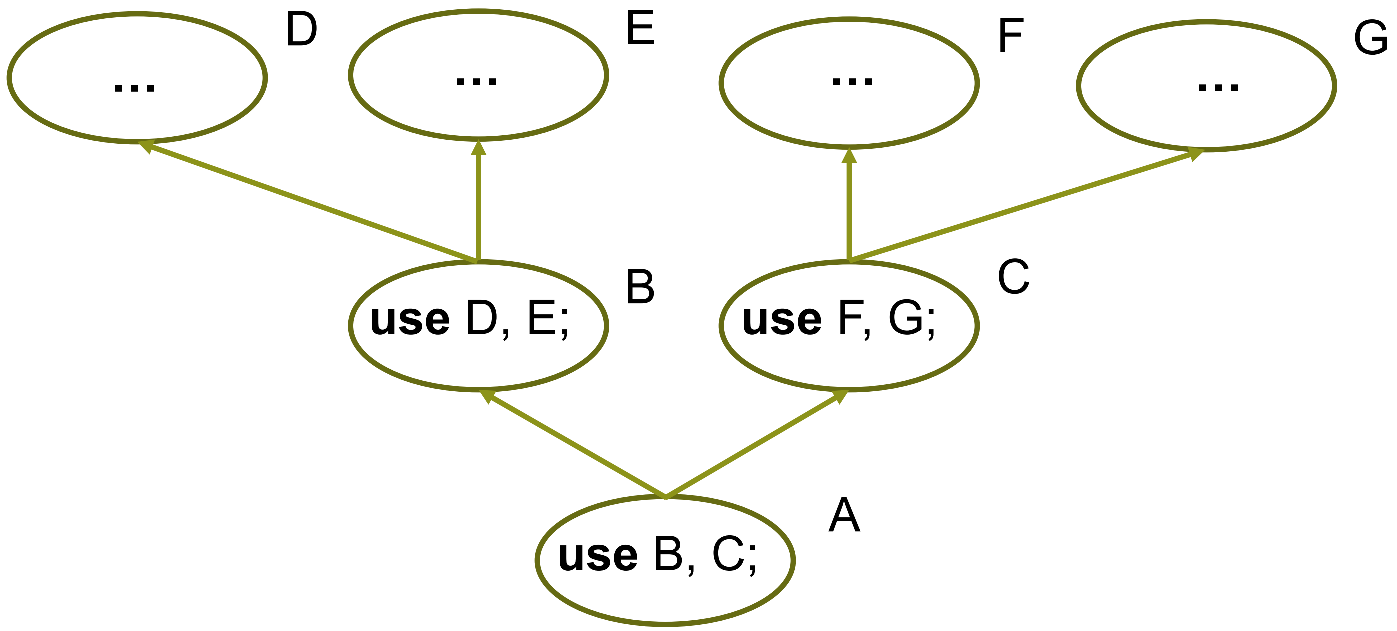
```
module A {
 var bar = 13;
 proc foo() { … }
}
```

```
module B {
 use A;

 var baz = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var flop = 7;

 proc main() {
  use B;

  flip(bar);
 }
}
```

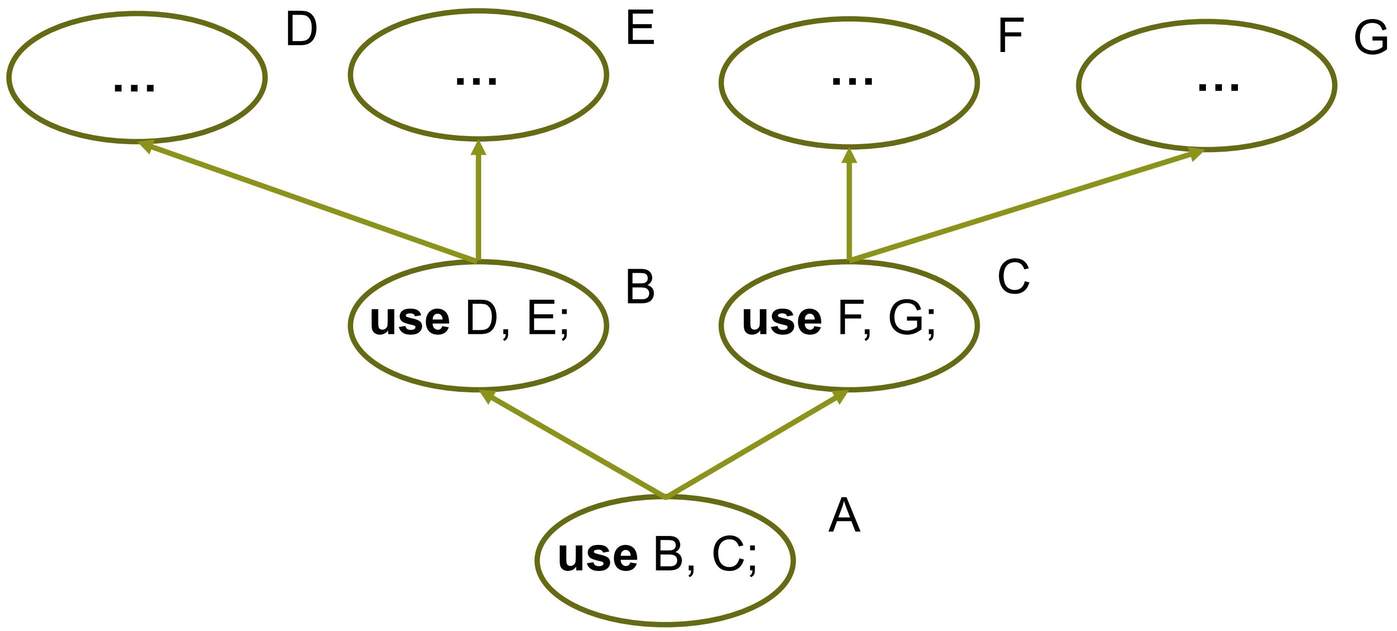
The Use and I: Transitive Uses

- **Symbols visible to B via a 'use' also visible to uses of B**
  - Best represented as a tree of 'use's



The Use and I: Transitive Uses

- **Symbols visible to B via a 'use' also visible to uses of B**
  - Best represented as a tree of 'use's
  - Each path in tree is a "use chain" (e.g. A->B->D, A->C->F)



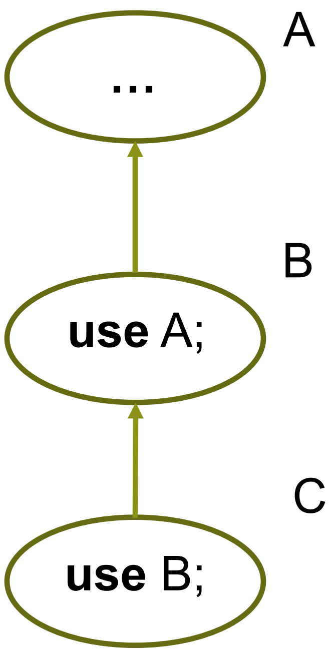
The Use and I: Transitive Uses

- **Symbols now visible to B also visible:**
  - within C's main
  - other uses of B

```
module A {
  var bar = 13;
  proc foo() { … }
}
```

```
module B {
  use A;

  var baz = 19;
  proc flip(x: int) { … }
}
```

```
module C {
  var flop = 7;

  proc main() {
    use B;

    flip(bar);
  }
}
```



The Use and I: Transitive Uses

- **Symbols visible to B via a 'use' also visible to uses of B**
  - Can avoid extra work

```
module A {
 class foo { … }
}
```

```
module B {
 use A;

 proc foo.bar() { … }
}
```

```
module C {
 proc main() {
  use B; // Instead of use A, B;

  var baz = new foo();
  // foo visible because B uses A
  baz.bar();
 }
}
```

The Use and I: Transitive Uses

- **Symbols visible to B via a 'use' also visible to uses of B**
  - Can avoid extra work
  - But can lead to unexpected issues
    - C's writer might not notice use of A

```
module A {
 var bar = 13;
 proc foo() { … }
}
```

```
module B {
 use A;

 var baz = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var bar = 7;

 proc main() {
  use B;

  flip(bar); // Finds A.bar, not C.bar
 }
}
```

The Use and I: Transitive Uses

- **Symbols visible to B via a 'use' also visible to uses of B**
  - Can avoid extra work
  - But can lead to unexpected issues
  - Same issues can occur with just B

```
module B {
 var bar = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var bar = 7;

 proc main() {
  use B;

  flip(bar); // Finds B.bar, not C.bar
 }
}
```

The Use and I: Import Control

- **Chapel 1.13 adds import control for use statements**
  - 'except' keyword prevents unqualified access to symbols in list

    ```
    use B except bar; // All of B's symbols other than bar can be named directly
    ```
  - 'only' keyword limits unqualified access to symbols in list

    ```
    use B only flip;     // Only B's flip can be named directly
    ```
  - Permits user to avoid importing unnecessary symbols
    - Including symbols which cause conflicts

```
module B {
 var bar = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var bar = 7;

 proc main() {
  use B except bar;

  flip(bar); // Finds C.bar, not B.bar
 }
}
```

  - Can rename imported symbols

    ```
    use B only bar as baz;
    // Can reference B.bar via baz
    ```

The Use and I: Import Control

- **Import control must affect all uses in use chain**
  - Would be equally incorrect to find A's bar or B's bar.

```
module A {
 var bar = 13;
 proc foo() { … }
}
```

```
module B {
 use A;

 var bar = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var bar = 7;

 proc main() {
  use B except bar;

  flip(bar); // Finds C.bar
 }
}
```

The Use and I: Import Control

- **Nested import control must be considered**
  - Shouldn't find symbols excluded by deeper import control

```
module A {
 var bar = 13;
 proc foo() { … }
}
```

```
module B {
 use A only foo;

 var goop = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var bar = 7;

 proc main() {
  use B except goop;

  flip(bar); // Finds C.bar
 }
}
```

The Use and I: Renaming

- Renaming a symbol should not allow access to old name

```
module A {
 var bar = 13;
 proc foo() { … }
}
```

```
module B {
 use A;

 var bar = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var bar = 7;

 proc main() {
  use B only bar as baz;

  flip(bar); // Finds C.bar
 }
}
```

The Use and I: Renaming

- **Renaming a symbol should not allow access to old name**
  - And nested renaming should not break this condition

```
module A {
 var bar = 13;
 proc foo() { … }
}
```

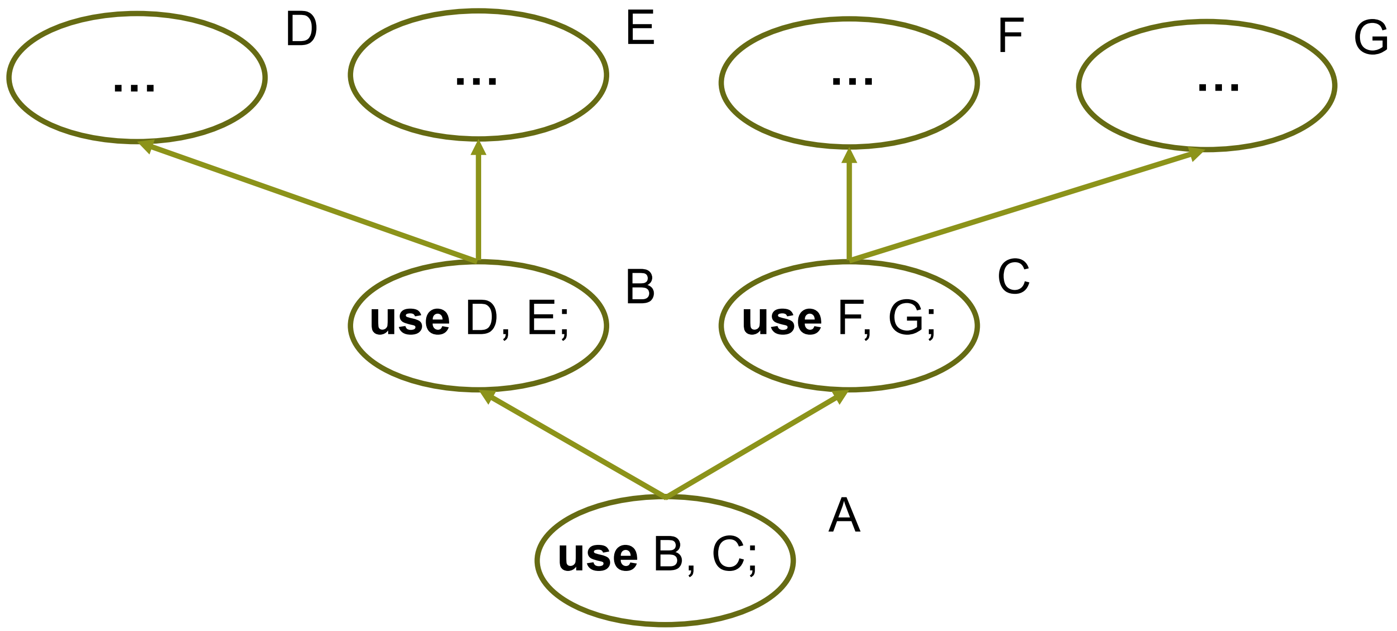
```
module B {
 use A only bar as baz;

 var goop = 19;
 proc flip(x: int) { … }
}
```

```
module C {
 var bar = 7;

 proc main() {
  use B only baz as biff;

  flip(bar); // Finds C.bar
 }
}
```

Compiler Implementation

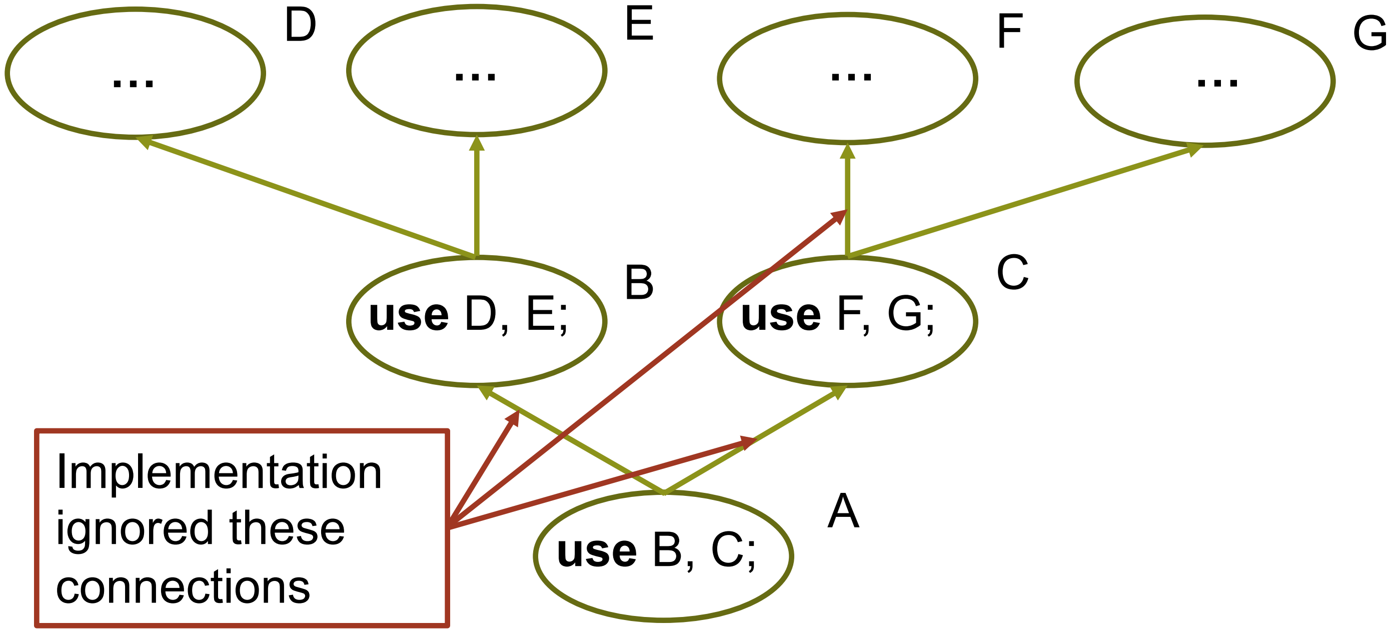
The Use and I: Transitive Uses

- **Symbols visible to B via a ‘use’ also visible to uses of B**
  - Best represented as a tree of ‘use’s
  - Each path in tree is a “use chain” (e.g. A->B->D, A->C->F)



The Use and I: Scope Resolve

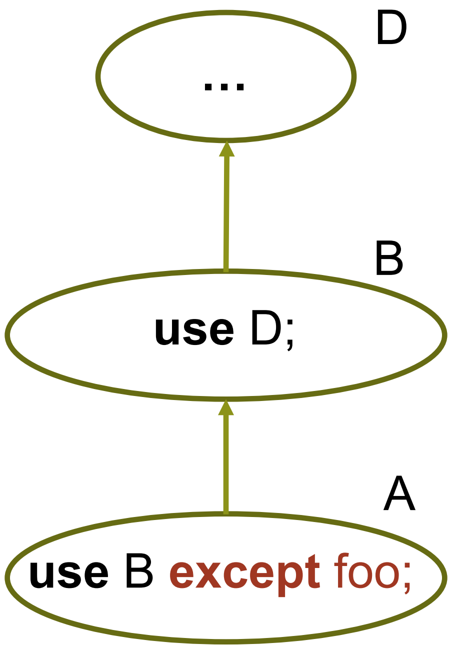
- **Scope resolution**
  - Handles variable, module name resolution
  - Traverses in breadth-first order
  - 'Use' tree built once per scope after module names resolved
    - Traversed many times



Traversal order:
- A
- <gap>
- B
- C
- <gap>
- D
- E
- F
- G
- …

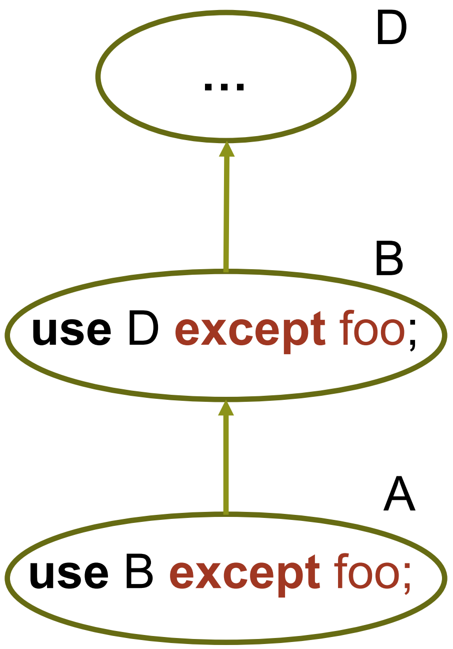
The Use and I: Scope Resolve

- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree



The Use and I: Scope Resolve

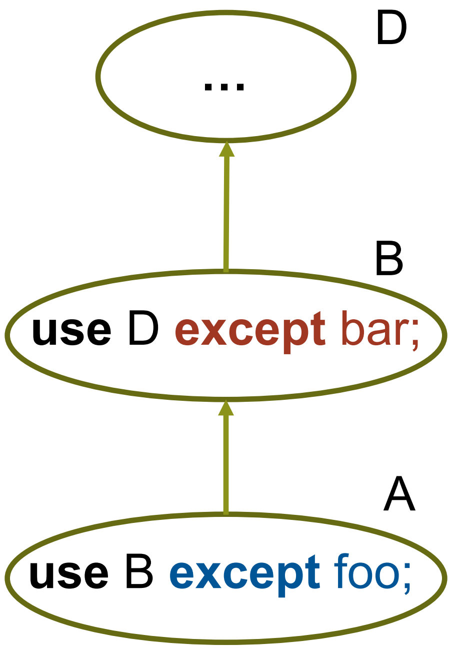
- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree



Note: In the case where B is 'use'd in multiple 'use' chains, these modifications should not be visible outside of the 'use' from A

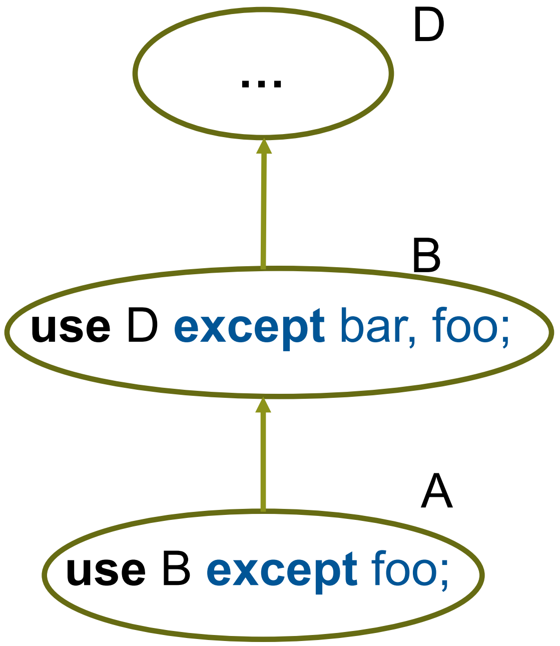
The Use and I: Scope Resolve

- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree
    - This can get tricky when multiple limits are present



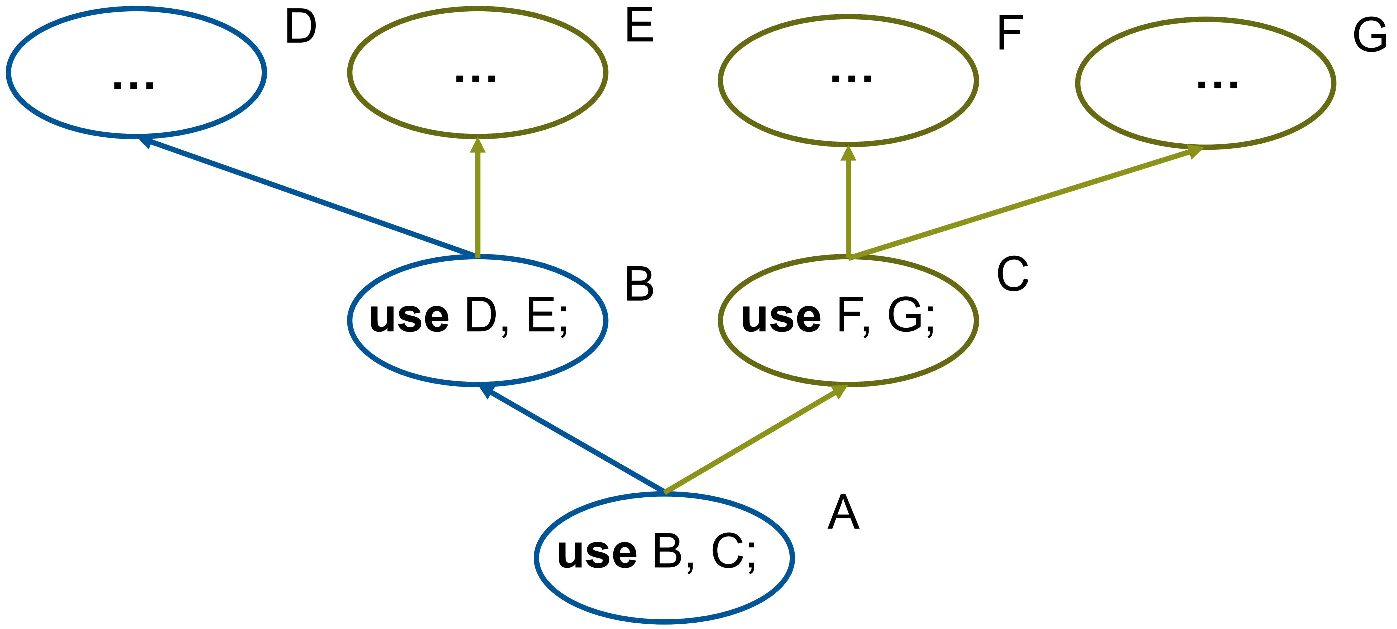
The Use and I: Scope Resolve

- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree
    - This can get tricky when multiple limits are present



The Use and I: Function Resolution

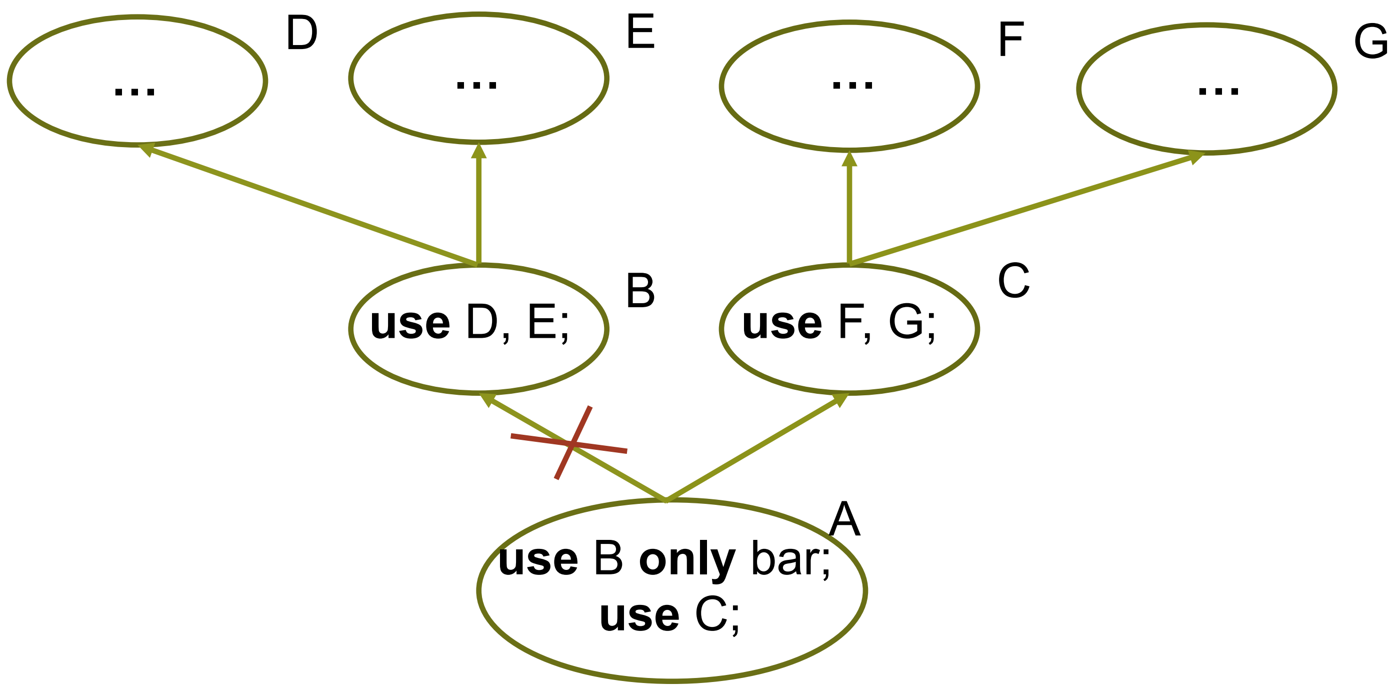
- **Function resolution**
  - Handles functions, some field resolution
    - Chooses best match from all matches at all visible scopes
  - Traverses ‘use’s depth-first
  - Later ‘use’s in chain accessed through earlier ‘use’s



Traversal order:
- A
- B
- D
- E
- C
- F
- G
- …

The Use and I: Function Resolution

- **Can determine whether to follow a 'use' chain**
  - If 'except' or 'only' list precludes desired name, skip that branch
  - Single check saves compilation time



Traversal order:
- A
- ~~B~~
- ~~D~~
- ~~E~~
- C
- F
- G
- …

The Use and I: Conclusions

- **Control over 'use' transitivity should be in user's hands**
  - Module designer has best knowledge of symbols to expose/hide
  - Intend to provide via reuse of 'public'/'private' keywords

```
private use M;
public use N;
```

- **Starting from "transitive by default" was beneficial**
  - Design of features forced to account for transitivity immediately
  - Found tricky cases early
- **Still deciding on default behavior**

Legal Disclaimer

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY

The Use and I: Private and Public

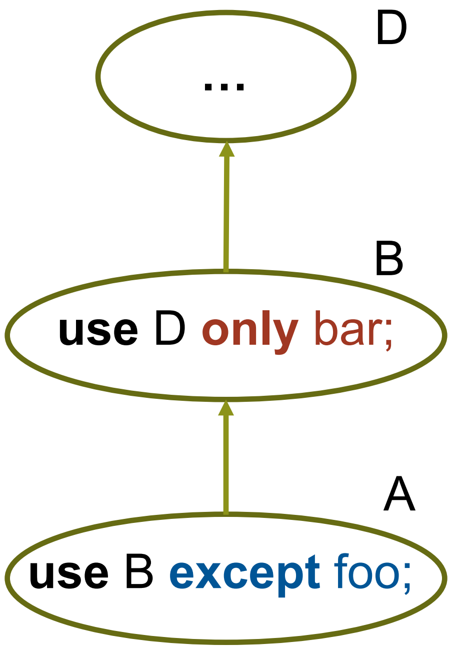
- **Declaring symbol “private” impacts outside access**
  - No explicit naming allowed from outer scope
  - ‘Use’ will not allow unqualified access of symbol either
  - Still visible from scopes nested within defining scope

```
private var foo = …;
proc bar() { … }  // Can reference foo within bar, etc.
```

- **Implementation:**
  - Same check on symbol match visibility used in both passes
    - Scope resolve looks at further ‘use’ depth if only private symbols found
    - Function resolution merely avoids that match

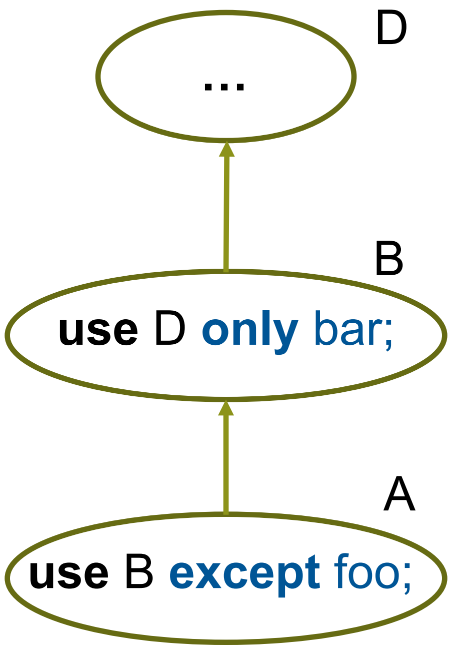
The Use and I: Scope Resolve

- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree
    - This can get tricky when multiple limits are present



The Use and I: Scope Resolve

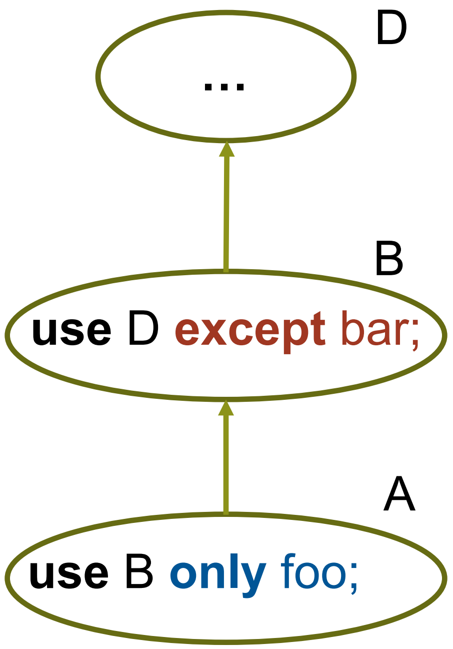
- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree
    - This can get tricky when multiple limits are present



If an outer 'except' list is distinct from an inner 'only' list, the 'only' list will be unchanged

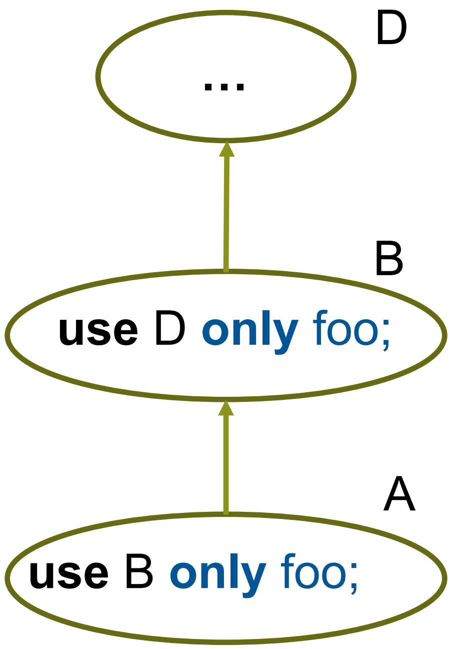
The Use and I: Scope Resolve

- **With ‘except’ and ‘only’ keyword, ‘use’ chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating ‘use’ tree
    - This can get tricky when multiple limits are present



The Use and I: Scope Resolve

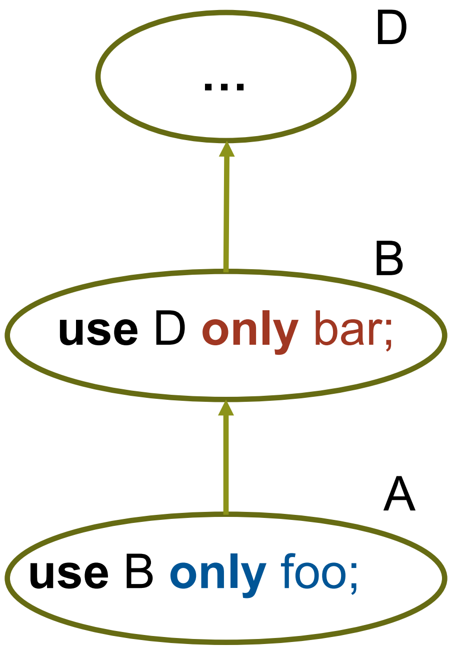
- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree
    - This can get tricky when multiple limits are present



If an outer 'only' list is distinct from an inner 'except' list, the 'only' list will replace the 'except' list.

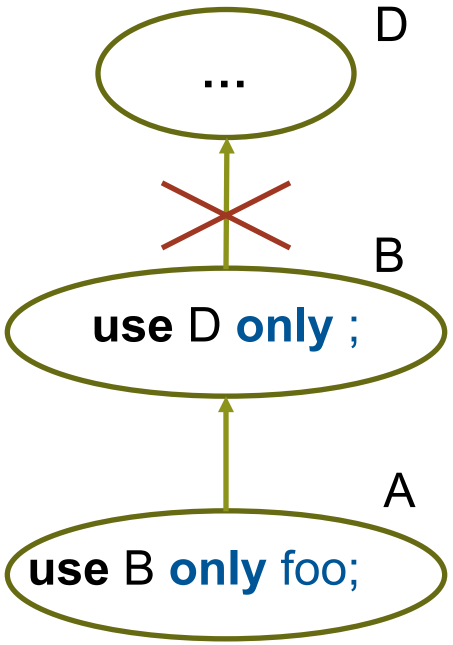
The Use and I: Scope Resolve

- **With ‘except’ and ‘only’ keyword, ‘use’ chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating ‘use’ tree
    - This can get tricky when multiple limits are present



The Use and I: Scope Resolve

- **With 'except' and 'only' keyword, 'use' chains matter more**
  - Earlier limits should affect search of later modules in chain
  - Need to apply these limits when creating 'use' tree
    - This can get tricky when multiple limits are present



If an outer 'only' list is distinct from an inner 'only' list, it will be as if that 'use' does not occur.

And any overlap will be handled appropriately