**out mode**

- This is less familiar
- Useful when a subprogram needs to “return” multiple values, but language allows only a single return value from a subprogram.
- Of course, could also return a composite value, such as a tuple, like ML does by default.
Ada example

// two values “returned”
procedure ex(x, y: in INTEGER; s, l: out INTEGER) is begin
  if x<y then
    s = x;
    l = y;
  else
    s = y;
    l = x;
  end if;
end ex;

// Called as follows:
//   a : INTEGER := 5; b : INTEGER := 3;
//   smallest: INTEGER; largest : INTEGER;
//   ex(a, b, smallest, largest);
// Binds smallest to 3, largest to 5
C# example

// two values “returned”
void ex(int x, int y, out int s, out int l) {
    if (x<y) {
        s = x;
        l = y;
    }
    else {
        s = y;
        l = x;
    }
}

// Called as follows:
//   int a=5, b=3, smallest, largest;
//   ex(a, b, out smallest, out largest);
// Binds smallest to 3, largest to 5
Prolog example

% ex(X,Y,Smallest,Largest) iff Smallest is the
% smaller of X and Y, and Largest is the larger.

ex(X,Y,X,Y) :- X < Y.
ex(X,Y,Y,X) :- X >= Y.

// Called as follows:
// A=5, B=3, ex(A, B, Smallest, Largest)
// Binds Smallest to 3, Largest to 5
in out mode

• There are many ways to implement this:
  – value-return
  – references
  – pointers
Swapping routines

// C - using pointers
swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

// Called as follows:
// int a=5, b=3;
// swap(&a, &b);
// Binds a to 3, b to 5
// C# - pass by reference
swap(ref int x, ref int y) {
    int tmp = x;
    x = y;
    y = tmp;
}

// Called as follows:
//   int a=5, b=3;
//   swap(ref a, ref b);
// Binds a to 3, b to 5
// C++ - pass by reference
swap(int &x, int &y) {
    int tmp = x;
    x = y;
    y = tmp;
}

// Called as follows:
//   int a=5; b=3;
//   swap(a, b);
// Binds a to 3, b to 5
value vs. reference

- If language gives choice, what are issues to consider between value vs. reference passing?
- Should caller be able to modify value of argument? ➔ reference
- Should caller be prevented from modifying value of argument? ➔ value
- Is argument large & expensive to copy? ➔ reference
- Is argument not an l-value? ➔ value
  - depends on language: Fortran manufactures a temporary variable
- Will argument be referenced frequently? ➔ value
  - to avoid cost of reference indirection
Note about Fortran

Fortran passes all parameters by reference, but does not require that every actual parameter be an l-value. If a built-up expression appears in an argument list, the compiler creates a temporary variable to hold the value, and passes this variable by reference. A Fortran subroutine that needs to modify the values of its formal parameters without modifying its actual parameters must copy the values into local variables, and modify those instead.

[Scott, Programming Language Pragmatics, p. 444]
Read-only reference parameters

• Modula-3 has a `READONLY` modifier
  – combine efficiency of reference parameters with the safety of value parameters

• C/C++ has `const` modifier (creates “elaboration-time constants”, as opposed to “compile-time constants”).
  – C example [Scott, p 445]
    ```c
    void append_to_log(const huge_record *r) {...}
    ...
    append_to_log(&my_record);
    – `const` applies to record: `r` can be changed, but not `*r`.```
const in C++

- `void foo(const int& x) {...}`
- `void foo(const int* x) {...}`
- `void foo(int* const x) {...}`
- `void foo(int* const x) {...}`
- `void foo(const int* const x) {...}`
- `void foo(const int* const x) {...}`
value-result vs. reference

type t is record
  a, b: integer;
end record;

r:t;
procedure foo(s : in out t) is
begin
  r.a := r.a + 1;       -- by value-result, s is distinct from r
  s.a := s.a + 1;       -- by reference, s and r refer to same structure
end foo;
...

r.a := 3;
foo(r);
put (r.a);  -- does this print 4 or 5?

[Scott, pg. 447]
Closures as parameters

• A closure is a reference to a subprogram together with a referencing environment)

• [See examples from Scott, pp. 450-451]
Call-by-name

• “Explicit subroutine parameters are not the only language feature that requires a closure to be passed as a parameter.”

• “A call by name parameter is re-evaluated in the caller’s referencing environment every time it is used. The effect is as if the called routine had been textually expanded at the point of the call, with the actual parameter (...) replacing every occurrence of the formal parameter.

• To implement call by name, Algol 60 implementations pass a hidden subroutine that evaluates the actual parameter in the caller’s referencing environment. This hidden routine is usually called a thunk.

[Scott, pp. 451-452]
Function returns [457]

• Restrictions on returned values:
  – Algol 60, Fortran: scalar value
  – Pascal, Modula-2: scalar or pointer
  – Algol 68, Ada, C: also composite type
  – Modula-3, Ada 95: also closure to be returned
  – C (no closures): function pointer
  – Scheme, ML: any type (incl. closure)