CSE 505

Lecture #3
September 5, 2012

Evolution of FORTRAN

FORTRAN IV (widely available)

FORTRAN 77 (ANSI standard)

Fortran 90 – recursion, pointers, dynamic storage allocation

HPFF – High Performance Fortran 90

Fortran 2003 – object-oriented programming procedure pointers

Fortran 2008 – modules, concurrency

Parameter Passing Modes

- Value (in)
- Result (out)
- Value-Result (inout)
- Reference (var or ref)
- Procedure (proc)
- Name (name)

General Comments

➢ Value parameters are used for sending input values for a function, e.g.
`factorial(in int n)`

➢ Result parameters are used for obtaining resulting values from a function, e.g.
```
search(in int key, out int value,
                   out boolean status)
```

➢ Value-result parameters combine the capabilities of value and result parameters, e.g.
`normalize(inout int num, den)`

Contrasting the Parameter Modes

```
void main() {
   int p;
   void halve(int x, y){
      x := x / 2;
      y := y / 2;
   }
   p := 4;
   halve (p, p);
   print(p);
}
```

Formal Parameter

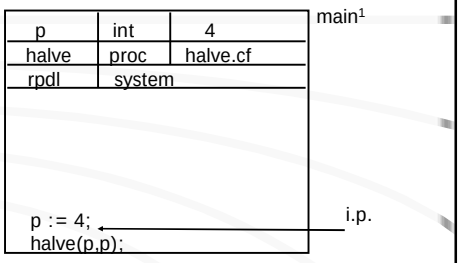
Actual Parameter

Actual Parameter Restrictions

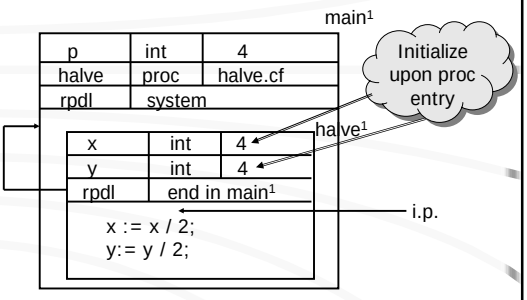
- The actual parameter corresponding to a value and name parameter can be a constant, variable, or a general expression.
- The actual parameter corresponding to a result, value-result, and reference parameter cannot be a constant or a general expression; it must be a variable.

Value or Call-by-Value Parameters

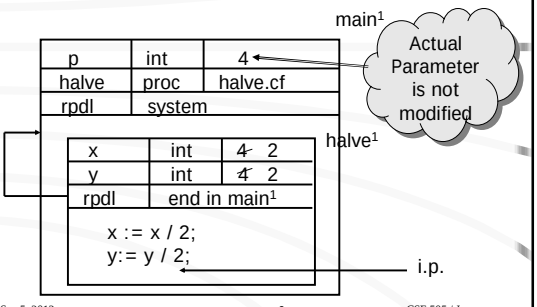
```
void halve(in int x, y)
```




```
void halve(in int x, y)
```

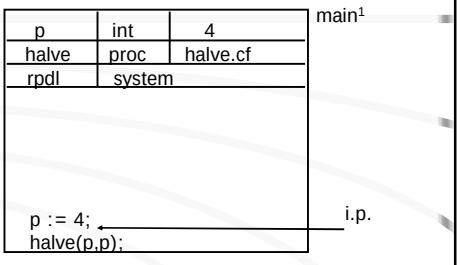



```
void halve(in int x, y)
```

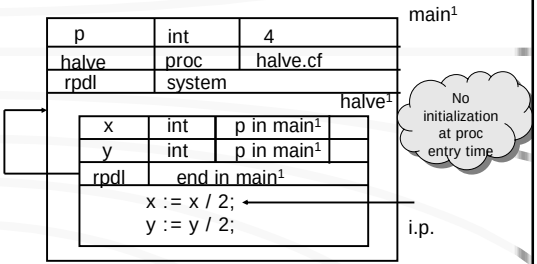


Result or Call-by-Result Parameters

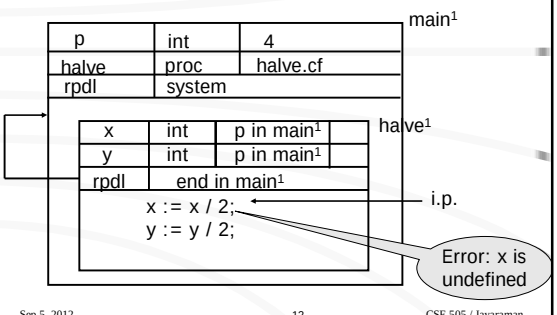
```
void halve(out int x, y)
```




```
void halve(out int x, y)
```




```
void halve(out int x, y)
```



Value vs Result Parameters

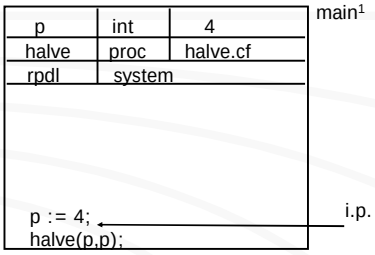
- Value: formal_par := actual_par (on entry)
- Result: actual_par := formal_par (on exit)
- Neither is appropriate for this example:

-If x and y are value parameters, then actual parameter p in main remains unchanged.

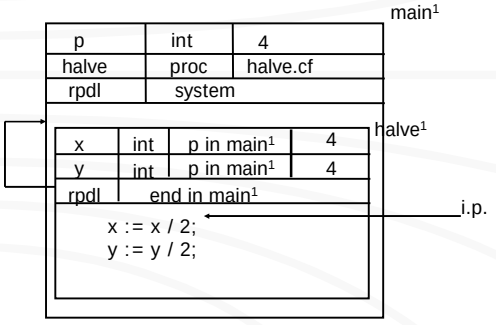
- If x and y are result parameters, there is a division error when x := x / 2 is executed.

Value-Result or Call-by-Value-Result

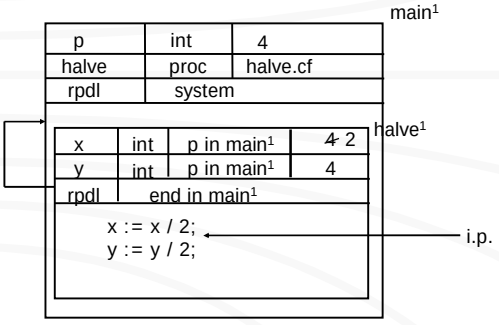
```
void halve(inout int x, y)
```



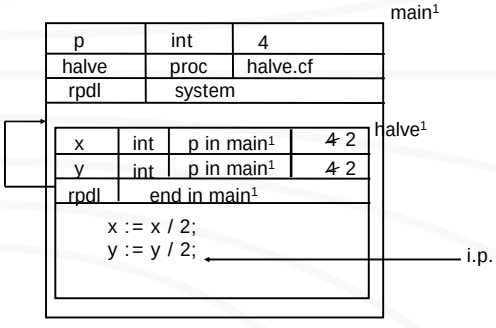

```
void halve(inout int x, y)
```



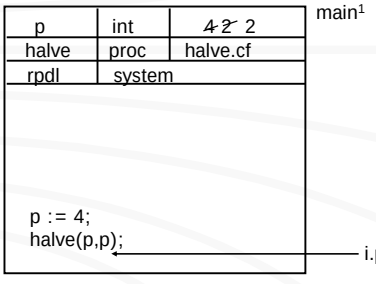

```
void halve(inout int x, y)
```




```
void halve(inout int x, y)
```




```
void halve(inout int x, y)
```



Notes on Value-Result Parameters

- Consider the following program:

```
void test(inout int x,y) {
     x := x / 2;
     y := y / 3;
}
… p := 30; test(p,p); print(p)
```

- Order in which result is returned affects the the printed value for **p**.

Notes on Value-Result Parameters

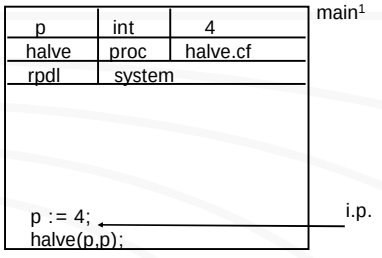
- Subscripted variables must be fixed at entry to procedure.

```
void halve(inout int x, y) {
     x := x / 2;
     y := y / 2;
}
p := 4;  halve(p, A[p]);
```

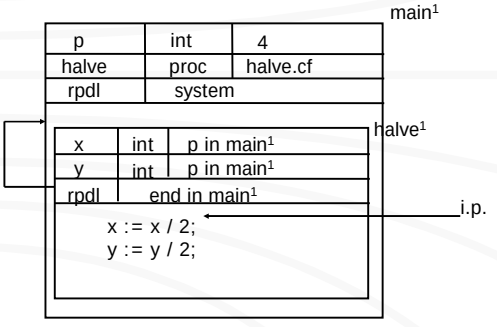
- The variable **A[p]** refers to **A[4]** at the entry to **halve** but refers to **A[2]** at the exit of **halve** .

Reference or Call-by-Reference

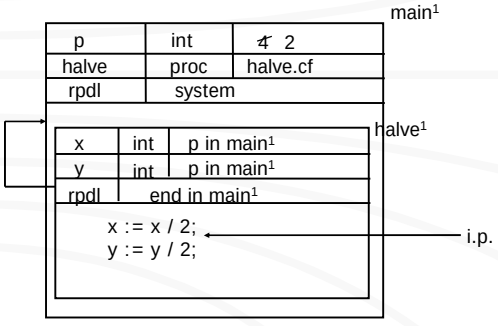
void halve(var int x, y)



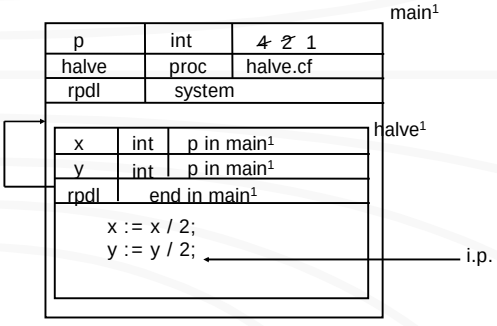
void halve(var int x, y)



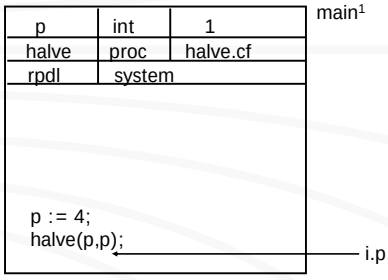
void halve(var int x, y)



void halve(var int x, y)



```
void halve(var int x, y)
```



Notes on Reference Parameters

Reference is not equivalent Value-Result!

- Reference parameters are used when large structures need to be passed.
- The language Ada omits reference parameters, but supports in, out, and inout. Compiler may optimize inout to reference when it can detect absence of aliasing – but is this easy to do?
- What does Java support?

More realistic example

```
void normalize(int num, den) {
        int  g;
        g := gcd(num, den);
        num := num / g;
        den := den / g;
}
```

Consider a call such as: normalize(k, k).

The expected answer for k is: ____
Value-result parameters will give: ____
Reference parameters will give: ____

Value-Result vs Reference

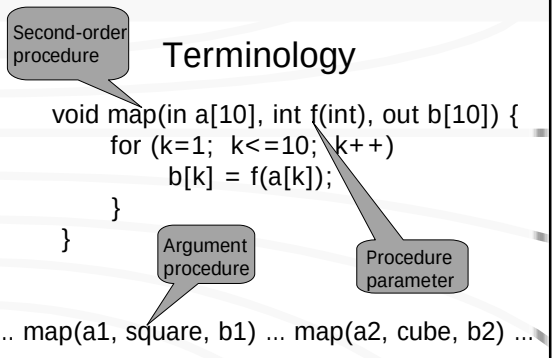
- Often they give identical answers, but the normalize example helps illustrate out their difference.
- Value-result is preferable to reference for "remote procedure calls," i.e., calls to procedures located elsewhere on the network.

Procedure Parameters

- The ability to parameterize a procedure by another procedure promotes greater code re-use and also greater program modularity.
- Code re-use example:

```
void map(in a[10], int f(int), out b[10]) {
        for (k=1;  k<=10;  k++)
                b[k] = f(a[k]);
        }
}
... map(a1, square, b1) ... map(a2, cube, b2) ...
```

Terminology



An Example in C: Integration

```
float integral( float(*f)(float),  float a, float b) {
    float  sum, h, fx;
    int    i;

    h = (b - a)/1000.0;
    sum = 0.0;

   for (i = 0;  i <= 1000;  i++) {
        fx  = (*f)(a + h*i);
        sum = sum + h * fx;
        }

   return sum;
}
```


Windows Programming

A 'call back' function is passed as a parameter, and is invoked when some notification is to take place. E.g.

Set_Timer(..., CallBackProc)

The CallBackProc is invoked when the time elapses.

Higher-Order Functions

Higher-Order Functions are used extensively in functional programming languages.

They can also be used to simulate advanced forms of control structures.

We will examine these two topics in more detail later.

Semantics of Procedure Parameters

Key Issue: Where should we create the contour when the argument procedure (ap) gets invoked via the procedure parameter (pp)?

Four plausible locations:

Static Scoping => #1

1. In the contour where ap is declared;
2. In the contour where ap is passed as actual par;
3. In the contour where pp is declared;
4. In the contour where pp is eventually invoked.

Why is this issue important?

Procedure parameters are useful in understanding other constructs. We will examine three such constructs:

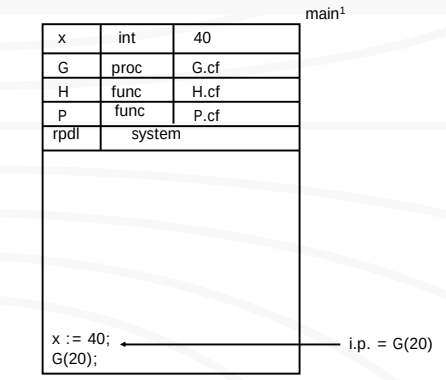
- name parameters
- iterators
- lazy constructors

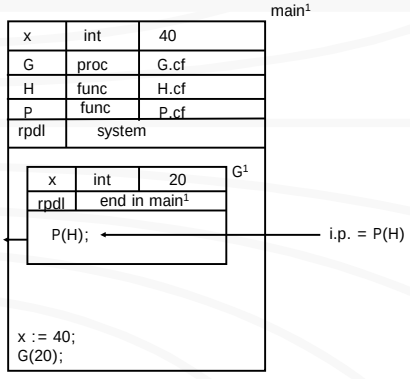
Static Scoping Rule

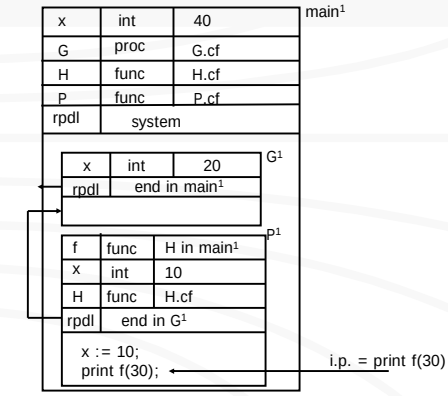
Pass the name of the argument procedure (actual parameter) as well as the contour in which it is declared – case (1) in previous slide.

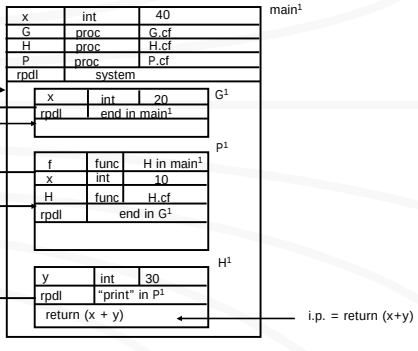
This ensures that the argument procedure can correctly access non-local variables, as per static scoping rules.

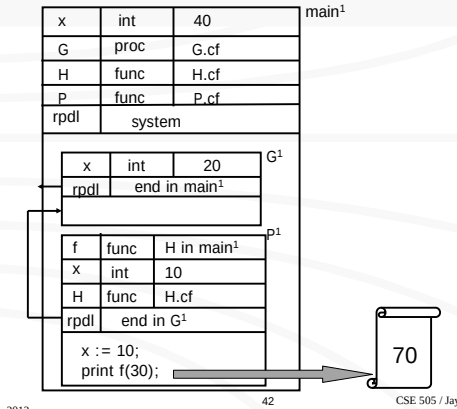
```
void main() {
   int x;
   void G(in int x) {
       P(H);
   }
   int H(in int y) {
       return (x + y);
   }
   int P(int f(in int)) {
    int x;
    int H(in int y) {
        return (x + y);
    }
    x := 10;  print f(30);
   }
   x := 40;  G(20);
}
```









Remarks on Procedure Pars.

Re four plausible locations for contour creation:

1. In the contour where ap is declared;
   - STATIC SCOPING
2. In the contour where ap is passed as actual par;
   - FLUID BINDING
3. In the contour where pp is declared;
   - FLUID BINDING
4. In the contour where pp is eventually invoked.
   - DYNAMIC SCOPING

Name Parameters - Motivation

$$\sum_{i=1}^{10} A[i] \qquad \sum_{n=5}^{15} X[n] * Y[n]$$

$$\sum_{i=1}^{10} \left(A[i] * \sum_{j=1}^{10} B[i,j] \right)$$

Plausible Translation of Σ

Suggested translation of $\sum_{i=1}^{10} A[i] * C[i]$

```
sum := 0;
for (i=1; i<= 10; i++) {
      sum := sum + A[i] * C[i];
}
```

Better:

By separating the code for array iteration from the code that operates on array elements, sigma promotes greater modularity and code re-use.

Suggested Translation

$\sum_{i=1}^{10} A[i]$ ➔ sigma(A[i], i, 1, 10)

$\sum_{n=5}^{15} X[n] * Y[n]$ ➔ sigma(X[n] * Y[n], n, 5, 15)

Sigma with 'name' parameter

```
int sigma(name int expr, var int k,
                    int lb, hb) {
    int sum := 0;
    for(k = lb; k <= hb; k++)
        sum := sum + expr;
    return sum;
    }
```

```
sigma(A[i]*C[i], i, 1, 10);
```

Informal Semantics of 'Name' Parameters

1. Don't evaluate actual parameter expression (e.g. A[i]*B[i]) at the point of call.
2. Re-evaluate actual parameter expression each time the name parameter (e.g. expr) is encountered while executing the body of the procedure (e.g. sigma).
3. Evaluation of actual parameter expression must take place according static scope rule.

Will sigma work correctly with in, out, inout, ref ? NO!

```
int sigma(int expr, var int k,
              int lb, hb) {
    int sum := 0;
    for(k = lb; k <= hb; k++)
        sum := sum + expr;
    return sum;
    }
```

```
sigma(A[i]* C[i], i, 1, 10);
```

Compile-time Transformation

1. Replace the name parameter, name T expr, by a a procedure parameter, T expr().
2. Replace all occurrences of the name parameter expr by a function call, expr().
3. Create one 'thunk procedure' for each actual parameter expression corr. to a name parameter; replace the actual parameter by the name of the thunk procedure.

Translation of $\sum_{i=1}^{10}$ A[i] * C[i]

```
int i; int A[10], C[10];
```

```
int thunk() { return A[i]* C[i]; }
```

```
int sigma(int expr(), var int k,
                        int lb, hb) {
    int sum := 0;
    for(k = lb; k <= hb; k++)
        sum := sum + expr();
    return sum;
    }
```

Function parameter

```
sigma(thunk, i, 1, 10);
```

Nested calls will also work!

Example:

sigma(A[i] * sigma(B[i,j], j, 1, 10), i, 1, 10)

This example requires the use of two thunks:

(i) For B[i,j])
(ii) For A[i] * sigma(B[i,j], j, 1, 10)

Discussion

The execution of the transformed procedure makes crucial use of the semantics of procedure parameters.

In particular, the static scoping semantics for procedure parameters (namely, passing the procedure-name + contour in which it is declared) is necessary for correct implementation of the semantics of 'name' parameters.

Parameter Passing Humor

Prof. N. Wirth
ETH, Zurich
Inventor of Pascal & Modula

In Europe, they say:
"W-i-r-t-h"

In America, we say:
"W-o-r-t-h"

Professor Wirth says:
In Europe, I am "called by name"
but in America I am "called by value"

Another use of Procedure Parameters

- Another use of procedure parameters is in translating "iterators".
- Before we explore this translation, we will examine:
  - Binding time
  - Structured Data Objects

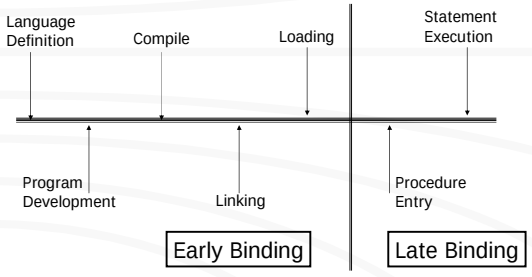
Binding Time

Definition: Binding time is the time at which an attribute is bound, or fixed, to an entity of a programming language.

Entity		Attributes
Variable	⇨	Type, Storage, Value, ...
Procedure	⇨	Body, Storage, ...

Binding Time: ..., compile-time, ..., run-time, ...

Binding Times



Binding Time	Attribute Bound *
Language Definition	Set of well-formed statements
Program Development	Set of variables
Compile	Code generated for expression
Linking	Code for library functions
Loading	Storage locations for program
Procedure Entry	Formal parameter value
Statement Execution	Value of an expression

* These examples don't hold for all languages.