

# Parameter Passing Methods

# Procedures

- **Modularize program structure**
  - *Argument*: information passed from caller to callee (actual parameter)
  - *Parameter*: local variable whose value (sometimes) is received from caller (formal parameter)
- **Procedure declaration**
  - name, formal parameters, procedure body with local declarations and statement list, optional result type

```
void translateX(point *p, int dx)
```

# Parameter Association

- **Positional association**
  - Arguments associated with formals one-by-one
    - E.g., C, Pascal, Scheme, Java
- **Keyword association**
  - E.g., Ada uses a mixture
    - procedure** plot (x,y: **in real**; penup: **in boolean**)
    - .... plot (0.0, 0.0, penup=> **true**)
    - ....plot (penup=>**true**, x=>0.0, y=>0.0)

# Parameter Passing Modes

- **pass by value**
  - C, Pascal, Ada, Scheme, Algol68
- **pass by result**
  - Ada
- **pass by value-result (copy-in, copy-out)**
  - Fortran, sometimes Ada
- **pass by reference**
  - Fortran, Pascal var params, sometimes Cobol
- **pass by name (outmoded)**
  - Algol60

# Pass by Value

```
{ c: array [1..10] of integer;  
  m,n : integer;  
  procedure r (k,j : integer)  
  begin  
    k := k+1;  
    j := j+2;  
  end r;  
  ...  
  m := 5;  
  n := 3;  
  r(m,n);  
  write m,n;  
}
```

**By Value:**

<u>k</u>	<u>j</u>
<del>5</del>	<del>3</del>
6	5

<b>Output:</b> 5 3
-----------------------

# Pass by Value

- **Advantages**
  - **Argument protected from changes in callee**
- **Disadvantages**
  - **Copying of values takes execution time and space, especially for aggregate values**

# Pass by Result

```
{ c: array [1..10] of integer;  
  m,n : integer;  
  procedure r (k,j : integer)  
  begin  
    k := k+1;  
    j := j+2;  
  end r;
```

*Error in procedure r:  
can't use parameters which  
are uninitialized!*

...

```
  m := 5;  
  n := 3;  
  r(m,n);  
  write m,n;  
}
```

# Pass by Result

- Assume we have *procedure*  $p(k, j : int)$  with  $k$  and  $j$  as result parameters. what is the interpretation of  $p(m, m)$ ?
  - Assume parameter  $k$  has value 2 and  $j$  has value 3 at end of  $p$ . What value is  $m$  on return?

# Pass by Value-Result

```
{ c: array [1..10] of integer;  
  m,n : integer;  
  procedure r (k,j : integer)  
  begin  
    k := k+1;  
    j := j+2;  
  end r;  
  ...  
  m := 5;  
  n := 3;  
  r(m,n);  
  write m,n;  
}
```

**By Value-Result**

<u>k</u>	<u>j</u>
<del>5</del>	<del>3</del>
6	5

<b>Output:</b> 6 5
-----------------------

# Pass by Value-Result

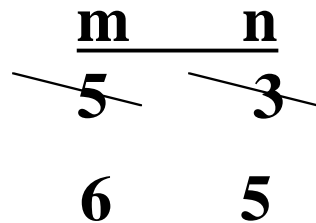
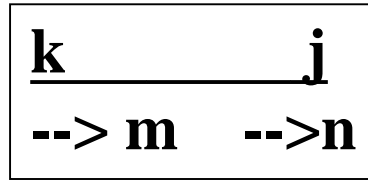
```
{ c: array [1..10] of integer;  
  m,n : integer;  
  procedure r (k,j : integer)  
  begin  
    k := k+1;  
    j := j+2;  
  end r;  
  /* set c[m] = m */  
  m := 2;  
  r(m, c[m]);  
  write c[1], c[2], ..., c[10];  
}
```

k	j
<del>2</del>	<del>2</del>
3	4

*What element of c  
has its value changed?  
c[2]? c[3]?*

# Pass by Reference

```
{ c: array [1..10] of integer;  
  m,n : integer;  
  procedure r (k,j : integer)  
  begin  
    k := k+1;  
    j := j+2;  
  end r;  
  ...  
  m := 5;  
  n := 3;  
  r(m,n);  
  write m,n;  
}
```



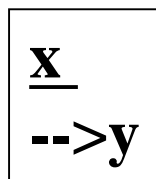
*Value update happens in storage of the caller while callee is executing*

# Comparisons

- **Value-result**
  - **Has all advantages and disadvantages of value and result together**
- **Reference**
  - **Advantage: is more efficient than copying**
  - **Disadvantage: can redefine constants**
    - r(0, X)* will redefine the constant zero in old Fortran'66 compilers
  - **Leads to aliasing: when there are two or more different names for the same storage location**
    - **Side effects not visible from code itself**

# Aliasing: by Reference

```
{ y: integer;  
  procedure p(x: integer)  
  { x := x + 1;  
    x := x + y;  
  }  
}
```



*during the **call**,  
x and y are the  
same location!*

...

```
y := 2;  
p(y);  
write y;  
}
```

~~2~~  
~~3~~  
6      output: 6

# No Aliasing: Value-Result

```
{ y: integer;
  procedure p(x: integer)
    { x := x + 1;
      x := x + y;
    }
  ...
  y := 2;
  p(y);
  write y;
}
```

x  
~~2~~  
~~3~~  
5

y  
~~2~~  
5

output: 5

# Another Aliasing Example

```
{ j, k, m :integer;
  procedure q( a, b: integer)
  { b := 3;
    m := m *a;
  }
...
s1: q(m, k);
...
s2: q(j, j);
...
}
```

*global-formal aliases:  
<m,a> <k,b> associations  
during call S1;*

*formal-formal aliases:  
<a,b> during call S2;*

# Pass by Reference

- **Disadvantage: if an error occurs, harder to trace values since some side-effected values are in environment of the caller**
- **What happens when someone uses an expression argument for a by reference parameter?**
  - **$(2*x)??$**

# Pass by Name

```
{ c: array [1..10] of integer;
  m,n : integer;
  procedure r (k,j : integer)
  begin
    k := k+1;
    j := j+2;
  end r;
  /* set c[n] to n */
  m := 2;
  r(m,c[m]);
  write m,n;
}
```

$m := m + 1$
--------------

$c[m] := c[m] + 2$
--------------------

<u>m</u>	<u>c[ ]</u>
<del>2</del>	1 2 <del>3</del> 4 5 6 7 8 9 10
3	1 2 5 4 5 6 7 8 9 10

# Pass by Name

- **Algol60 device**
  - **Deferred calculation of the argument until needed; like textual substitution with name clashes resolved**
  - **THUNK - evaluates argument in caller's environment and returns address of location containing the result**
- **Characteristics**
  - **Inefficient**
  - **Same as pass by reference for scalars**

# Procedures as Parameters

- To type check the call, need the full function signature of the function argument

**<function name> :**

**<vector of parameter types> → <return type>**

**e.g.,** `translateX: (point *, int) → void`

```
procedure q( x: integer;
```

```
    function s (y, z: integer): integer)
```



**s takes 2 integer arguments and returns an integer!**

# Example

```
{ m, k : integer;
  procedure q(x : integer; function s(y,z: integer): integer)
  { k, l : integer;
    ...
    s(...); /*call to function parameter s */
    ...
  } /* end of q*/
  integer function f(w,v: integer)
  { ...
    w := k*v; /* which k is this? k or k?*/
  }
  ...
  q(m, f);
  ...
}
```