— AR
— parameter passing

[Ch 9. Subprograms
[ch 10. Implementing subprograms) — sebesta book (11)

Ch 9.

[— procedures — procedure call is an atomic statement
[— function — called from within expression / has return value.

— benefits of using subprogram
- abstraction — users don't have to know details
- implementation hiding
- modular programm
- libraries

— parameter passing methods
- call by value — pass the r-value
- ← call by result
- call by value-result — actuals $\xrightarrow{copy}$ formals; then, formals → actuals.
- call by reference — pass the l-value (location)
- call by name — pass the text
- macro-expansion

[formal para. — in func. definition
[actual para. (≡ argument) — in calling statement.

— Call by value — formal para. corresponds to the value of actual.

r-value

ex) C — uses only call-by-value.

```
void swap (int x, int y)
{ int z;
  z=x;
  x=y;
  y=z;
}
```

— doesn't work for swap.

ex)
```
a=3;
b=2;
swap(a,b);
```
— doesn't change a, b.

formals (x, y) ← 3, 2

```
z = x (3)
x = y (2)
y = z (3)
```
— values of x and y are exchanged (not a, b).

↓

working version (swap in C)

l-value, l-value

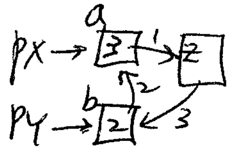
```
— void swap (int *px, int *py)
{ int z;
  z = *px;   // px — pointer to int type.
  *px = *py; // *px — content of px (dereferencing)
  *py = z;
}
```

ex)
```
a=3, b=2;
swap (&a, &b);
```
⇒ effect

l-value

px ← &a
py ← &b

```
z = *px;
*px = *py;
*py = z;
```



⇒ (*px, *py) exchanged.

— Call by reference

ex) C++ supports [call-by-value, call by ref.]

ref. para

```
— void swap (int &x, int &y)
{ int z=x;
  x=y;
  y=z;
}
```

ex)
```
i=2
A[i]=99;
swap (i, A[i]);
```

— i and A[i] values exchanged.

effect

```
l.x ⇐ l.i
l.y ⇐ l.A[i]
z=x (2)
x=y (99)
y=z (2)
```

⇒ effect

i $\boxed{\cancel{2}}$ 99 ← x

A[2] $\boxed{\cancel{99}}$ 2 ← y

(i and x, A[2] and y) — are aliases

(two expr's denoting the same location)

- Call by value-result (copy-in/copy-out)

  - actuals → formals (copy-in)
  - :
  - final values of formals → actuals (copy-out)

- actuals
  - if r-value (e.g., 2+3) — pass by value
  - if l-value (name)
    - copy-in phase:
      - Compute both value and location of actuals
        - value → formal
        - save location of actuals for corresponding formals (e.g. in AR.)
    - Copy-out phase:
      - final values of formals → locations of corresponding actuals.

Ada
- call by reference
- copyin/copyout

— same effect

- parameter types
  - in para. — value
  - out " — copyout phase
  - in out " — reference or value result.

↓ call-by-value-result

ex) — in C/C++ like syntax

```
int i, j ;  — globals
func foo (int X, int Y)
{ i = Y ;
  j = X ;
}

main ()
{ i = 2 ;
  j = 3 ;
  foo (i, j) ;
}
```

l-value (name)

i = 2
j = 3 — no change

copy-in phase

formals (X ← i (2)
        (Y ← j (3)

r-value

save (l.i for X — X becomes alias for i
     (l.j for Y — Y becomes alias for j

execution of func foo's body.

(i = Y(3) ;
(j = X(2) ;

Copy-out phase

restore i, j with the final values of X, Y.

(2) X → i
(3) Y → j

---

ex) Swap (int X, int Y)

```
{ int Z ;
  Z = X ;
  X = Y ;
  Y = Z ;
}
  ⋮
i=2 ; A[i]=99 ;
call swap (i, A[i]) ;
```

i ~~2~~ 99
A[2] ~~99~~ 2

exchanged
(Same effect as Call by reference)

Copy-in

(X ← i(2)
(Y ← A[2](99)

save (l.i for X
     (l.A[2] for Y

execution

(Z = X(2)
(X = Y(99)
(Y = Z(2)

Copy-out

(X(99) → i
(Y(2) → A[2]

— Macro expansion

— in C like lang.

ex) #define aa 4

# — macro

C++

in-line expansion

\#define swap(- -){- - -}

procedure/func.

control moves at run time

macro

text is copied at compile time

① text of actuals → formals (substituted)

② text of macro-body replaces call statement

at compile time

⇒ dynamic scope rule

parameter passing method — textual substitution.

ex) swap(X, Y) — "i" "A[i]"

```
{ ~~int~~ z = x;
        X = Y;
        Y = z;
}
  :
i=2; A[i]=99;
swap(i, A[i]);
```

phase1:

```
swap(i, A[i])
{ z = i;
  i = A[i];
  A[i] = z;
}
```

phase2: copy

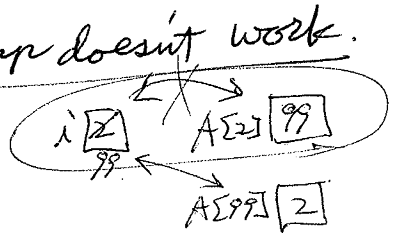
becomes

```
z = i;        2
i = A[i];     99
A[i] = z;     99  2
```

$\Rightarrow$ i ← 99

A[99] ← 2

— swap doesn't work.



— Call by name (ALGOL 60) — where name conflicts, rename — lexical (static) scope.

different from macro-expansion (naive copying) — dynamic scope

naive copying → dynamic scope — when name conflicts, take dynamic scope rule

(macro-expansion)

(allows the context vary)

Call-by-name (in ALGOL 60)

if lexical scope is needed, use renaming.

1. actual para → textually substituted into formals

if actual name ⟷ local name in procedure body

(conflicts)

⇒ rename locals in callee

2. procedure body is substituted for the call statement

if non-locals in proc. body ⟷ locals in caller

(conflict)

⇒ rename locals of caller.

So, all names become unique → keeping lexical scope

(i.e., context doesn't vary)

ex)

↓

#2 ALGOL60

```
begin
  integer A;
  procedure Inc3(n);          callee   "i'"
    begin
      integer i;              conflict
      i := 3;
      n := n+i;
      A := 3;
    end
  procedure B;   caller
    begin
      integer i, A;
      i:=2;
      Inc3(i);
      A:=1;
    end
  :
  B;
end.
```

①

1. text of actuals → formals

(if name conflicts, rename locals of callee

```
integer i';
i':=3;
i := i+i';
A := 3;
```

A: non-local

②

2. body of callee → copy to call statement

(if non-locals in callee ↔ locals in caller, rename locals in caller.

```
begin
  integer i, A';         renamed
  i :=2;
    integer i';
    i' :=3;
    i := i+i';
    A :=3;
  A' :=1;                renamed
end
```

memory

- global A 3
- procedure B

i ~~2~~ 5   i' 3

~~A~~ A' 1

§9.5.5.

— parameter type checking

actual → formal

— original C, Fortran77 — don't check it.

— Perl, JavaScript, PHP, python, Ruby.

(var's don't have type

ex) C.

```
double sin (double x);  — prototype
double value;
int count;
   ⋮
value = sin (count);  — legal.
```

int → to double coercion

— if coercion is not possible, → semantic error.

— multi-dimensional array as parameter

ex) C/C++ 2D array → row-major order.

00	01	02	03
10	11	12	13
20	21	22	23

Storage mapping function: — needs only #cols.

$$\text{addr.}(\text{matx}[i, j]) = \underbrace{\text{addr.}(\text{matrix}[0, 0])}_{\text{base}} + \underbrace{i * \text{num\_cols} + j}$$

⇒ So, in C/C++,

```
ex) void funcA (int matrix [ ][10])
    {  ≡
    }
    caller ( )
    {
      int mat [5][10];
        ⋮
      funcA (mat);
        ⋮
    }
```

only #cols is needed in formals.

→ problem: only for #col=10 case.

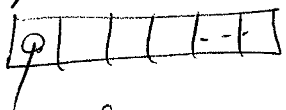
⇒ better idea: using pointers. — matrix is passed as a pointer.

```
void funcA (float *mat_ptr, int num_rows, int num_cols) {
{  ⋮
ex)  *(mat_ptr + (i * num_cols) + j) = x;
     ⋮
}
            // mat[i][j] ← x
```

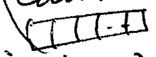
— funcA is generic for any #row/#cols.

ex) Java (also C#) — arrays are objects

single-dimensioned



each ele. can be array

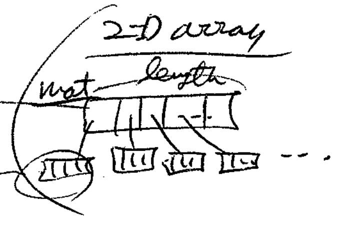


```
float sumer(float mat[][])
{ float sum=0.0 ;
  for (int i=0 ; i<mat.length ; i++)
    for(int j=0 ;
        ~~col~~ j<mat[i].length ; j++)
      sum += mat[i][j] ;
  return sum ;
}
```

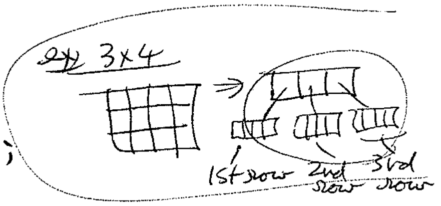
mat.length — 1st dimension's length.

mat[i].length — 2-D's length.

2-D array



ex) 3x4



in caller,

```
float[][] mat = new float[3][4];
//assign values to array elements
float sum = sumer(mat);
```

§9.6 Parameters that are subprograms. — not in C/C++, but pointers to func's can.

only in nested subprograms lang's. (~~dynamic scope~~)

— 3 choices of binding non-locals

1. shallow binding — bound to caller's environment
2. deep binding — bound to body owner's env.
3. ad hoc binding — bound to the environment in which the call statement passed the subprog. as para.

ex) JavaScript

```
function sub1()
{ var x ;
  function sub2()
  { alert (x) ;
  } ;
  function sub3()
  { var x ;
    x=3 ;
  } sub4 (sub2) ;
```

```
  function sub4 (subx)
  { var x ;
    x=4 ;
    subx() ;
  } ;
  x=1 ;
  sub3() ;
} ;
```

subx — sub2

subx() ← sub2 is called

⟹ under

1. shallow binding output = 4
2. output = 1
3. output = 3