Principles of Programming Languages

Lecture 06

Parameters

Parameter Passing Modes

- Definitional Modes (call time binding)
  - Call as constant — CAC
  - Call by reference — CBR
- Copying Modes (call time copying)
  - Call by value — CBV
  - Call by copy — CBC
    - Copy-in/copy-out
  - Call by value-result — CBVR
- Delay Modes (reference time copying)
  - Call by text — CBT
  - Call by name — CBN
  - Call by need
    - R-value — CBNeed-R
    - L-Value — CBNeed-L

Example Program

```
{ local i,j
  local a[1..12]
  proc P( X , Y )
  { local j
    j = 2
    i = i + 1
    print X
    X = X + 2
    print Y
    i--; print Y
  }
  a[1]=1; a[2]=2; … ; a[12]=12;
  i = 1
  j = 3
  P( i , a[i*j] )
  print i
  print a[9]
}
```

Program Execution Sequence

```
i = 1
j = 3
P( i , a[i*j] )
   X ,     Y
j = 2                   // this j is local
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

Call by text (macrosubstitution)

```
i = 1
j = 3
P( i , a[i*j] )
   X ,      Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

rewrite proc body

```
X -> i  Y -> a[i*j]
j = 2 // j==2
i=i+1 // i==2
print i                 2
i = i+2 //i==4
print a[i*j] //a[4*2]8
i--; print a[i*j]   6
print i                 3
print a[9]              9
```

Call by name (copy rule)

```
i = 1
j = 3
P( i , a[i*j] )
   X ,     Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

Rename locals **j->j'** *& rewrite*
X -> i Y -> a[i*j]

```
j = 2 //j'==2   j==3
i=i+1 //i == 2
print i                 2
i = i+2 //i==4
print a[i*j]//a[4*3]    12
i--; print a[i*j]       9 !
print i                 3
print a[9]              9
```

Algol 60 "Copy Rule"

- **4.7.3.2.** Name replacement (call by name)
  Any formal parameter not quoted in the **value** list is replaced, throughout the procedure body, by the corresponding actual parameter . . . Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved . . . Finally the procedure body, modified as above, is inserted in place of the procedure statement [the call] and executed . . .

  –Report on the Algorithmic Language ALGOL 60,
  CACM, May 1960

Algol 60 “Copy Rule” (cont.)

```
integer procedure f(x);
  integer x;
  begin  integer y;
      y := 1;
      f := x + y;
  end;
begin integer y, w;
  y := 2; w := 10;
  w := f(y + w + 1);
end;
```

Copy rule

```
begin integer y,w;
  y :=2; w := 10;
  begin  integer z;
      z := 1;
      w := (y+w+1)+z;
  end;
end;
```

Algol 60 "Copy Rule" (cont.): Thunks

- *thunk:* a 0-arg. Procedure encapsulating actual argument plus environment of caller, called at site of each formal.
- Used to implement CBN and <u>*delayed evaluation*</u> in general

```
integer procedure f(x);
  integer x;
  begin  integer y;
      y := 1;
      f := x + y;
  end;
begin integer y, w;
  y := 2; w := 10;
  w := f(y + w + 1);
end;
```

```
int function f(int x);
  { int y;
      y = 1;
      f = *x() + y;
  }
{ int y,w;
int& function x()
{int t=y+w+1;return &t;}
  y = 2; w = 10;
  w = f(y+w+1);
}
```

Call by name—implemented by thunk

```
i = 1
j = 3
P( i , a[i*j] )
   X ,     Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

- *thunk*: 0-ary function encapsulating argument text

```
int& X(){return &i}
int& Y(){return &a[i*j]}
```

```
j = 2 // j'==2  j==3
i=i+1 // i == 2
print *X()                  2
X()=*X()+2 //i==4
print *Y()
//a[i*j]==a[4*3]           12
i--; print *Y()            9 !
print i                     3
print a[9]                  9
```

Call by reference

```
i = 1
j = 3
P( i , a[i*j] )
   X ,     Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

aliased

lval(X) == *lval*(i)
lval(Y) == *lval*(a[1*3])

```
j = 2 //j'==2  j==3
i=i+1 //i== 2
print i                2
i = i+2  //i==4
print a[1*3]           3
i--; print a[1*3]      3
print i                3
print a[9]             9
```

Call by value (copy-in)

```
i = 1
j = 3
P( i , a[i*j] )
    X ,     Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

```
X = i       //X==1
Y = a[i*j])//Y== a[1*3]
j = 2 //j'==2  j==3
i=i+1 //i == 2
print X                 1
X = X+2 //i==2 X==3
print Y                 3
i--; print Y            3
print i                 1
print a[9]              9
```

Call by value–result (Algol W)

```
i = 1
j = 3
P( i , a[i*j] )
    X ,      Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

```
X = i
Y = a[i*j])//Y==a[1*3]
j = 2 // j'==2   j==3
i=i+1 // i == 2
print X                  1
X = X+2 //i==2 X==3
print Y                  3
i--; print Y             3
```

Use names of actuals for copy-out

```
i = X         //i==3
a[i*j] = Y   //a[3*3]==3
print i                  3
print a[9]               3
```

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

```
i = 1
j = 3
P( i , a[i*j] )
   X ,     Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

```
px=lvalue(i)          X=*px
py=lvalue(a[1*3]) Y=*py
j = 2 // j'==2  j==3
i=i+1 // i == 2
print X                   1
X = X+2 //i==2 X==3
print Y                   3
i--; print Y              3
```

Use original lvals for copy-out

```
*px = X      //i==3
*py = Y      //a[3]==3
print i                   3
print a[9]                9
```

Call by need, r-value ("normal", lazy)

```
i = 1
j = 3
P( i , a[i*j] )
   X ,     Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

- Use thunk *once* on 1st ref to assign local; then use local

```
int& X(){return &i}
int& Y(){return &a[i*j]}
```

```
j = 2 // j'==2  j==3
i=i+1 // i == 2
print X = *X()          2
 //X==2
X = X+2 //i==2 X==4
print Y = *Y()
//Y==a[i*j]==a[2*3]     6
i--; print Y            6
print i                 1
print a[9]              9
```

Call by need, l-value

```
i = 1
j = 3
P( i , a[i*j] )
    X ,     Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

- At 1st ref, alias actual lval to formal

```
int& X(){return &i}
int& Y(){return &a[i*j]}
j = 2 // j'==2  j==3
i=i+1 // i == 2
lval(X)== X() //X aliases i
print X  //X==2          2
X = X+2 //i==4 X==4
lval(Y)== Y()//Y aliases a[4*3]
print Y
//Y==a[12]}              12
i--; print Y             12
print i                  3
print a[9]               9
```

Call by Name vs. Call by Need

- Assume **X** is the formal and *e* the corresponding actual expression
- CBN
  - Delays evaluation of arguments past call until a reference to the formal
  - <u>*Re-evaluates*</u> argument *e* on each reference to **X** *in environment of caller*
  - No local variable **X** is allocated
  - Implemented by call to thunk
- CB Need
  - Delays evaluation of arguments past call until a reference to the formal
  - Evaluates *e* on 1st reference *in environment of caller* & loads local variable **X**; <u>*no re-evaluation*</u>: subsequent references use local **X**
  - Implemented by call to "memoized thunk"

Call by constant

```
i = 1
j = 3
P( i , a[i*j] )
    X ,      Y
j = 2
i = i + 1
print X
X = X + 2
print Y
i--; print Y
print i
print a[9]
```

```
const X = i
const Y = a[1*3]
j = 2 // j'==2    j==3
i=i+1 // i == 2
print X              1
X = X+2          error
print Y            (3)
i--; print Y       (3)
print i            (1)
print a[9]         (9)
```

Modes Differ

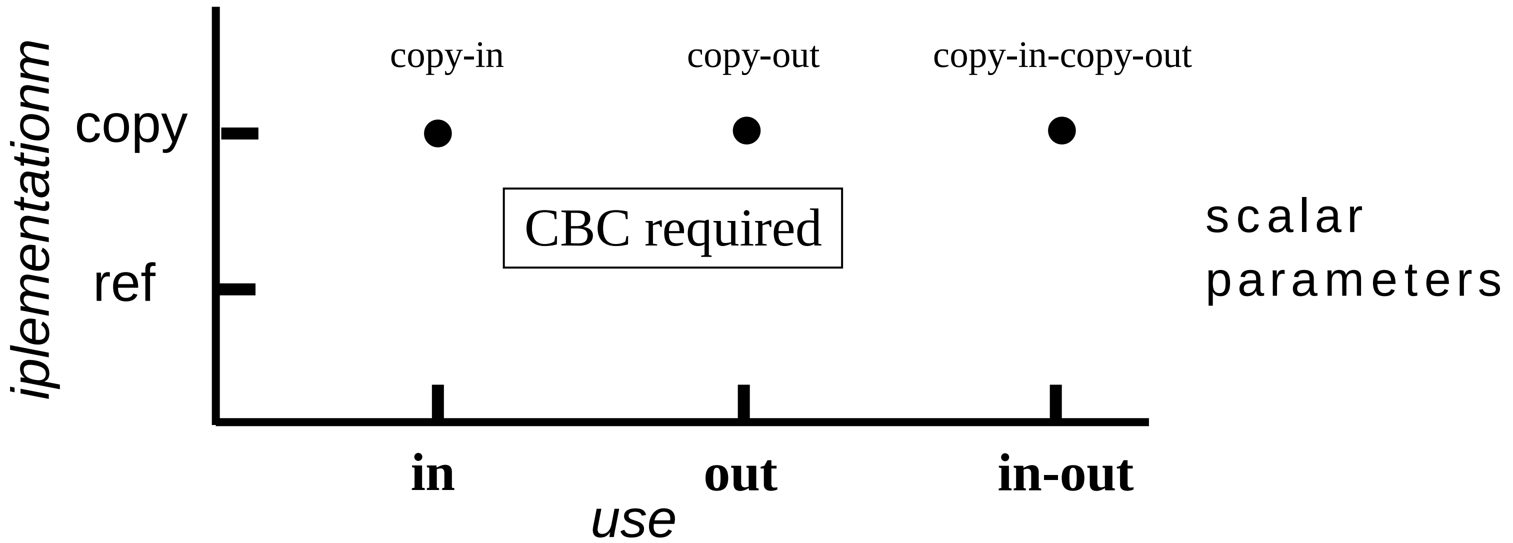
- Definitional Modes (call time binding) — example
  - Call as constant — CAC — 1 error
  - Call by reference — CBR — 2 3 3 3 9
- Copying Modes (call time passing)
  - Call by value — CBV — 1 3 3 1 9
  - Call by copy — CBC — 1 3 3 3 9
    - Copy-in/copy-out
  - Call by value-result — CBVR — 1 3 3 3 3
- Delay Modes (reference time passing)
  - Call by text — CBT — 2 8 6 3 9
  - Call by name — CBN — 2 12 9 3 9
  - Call by need
    - R-value — CBNeed-R — 2 6 6 1 9
    - L-Value — CBNeed-L — 2 12 12 3 9

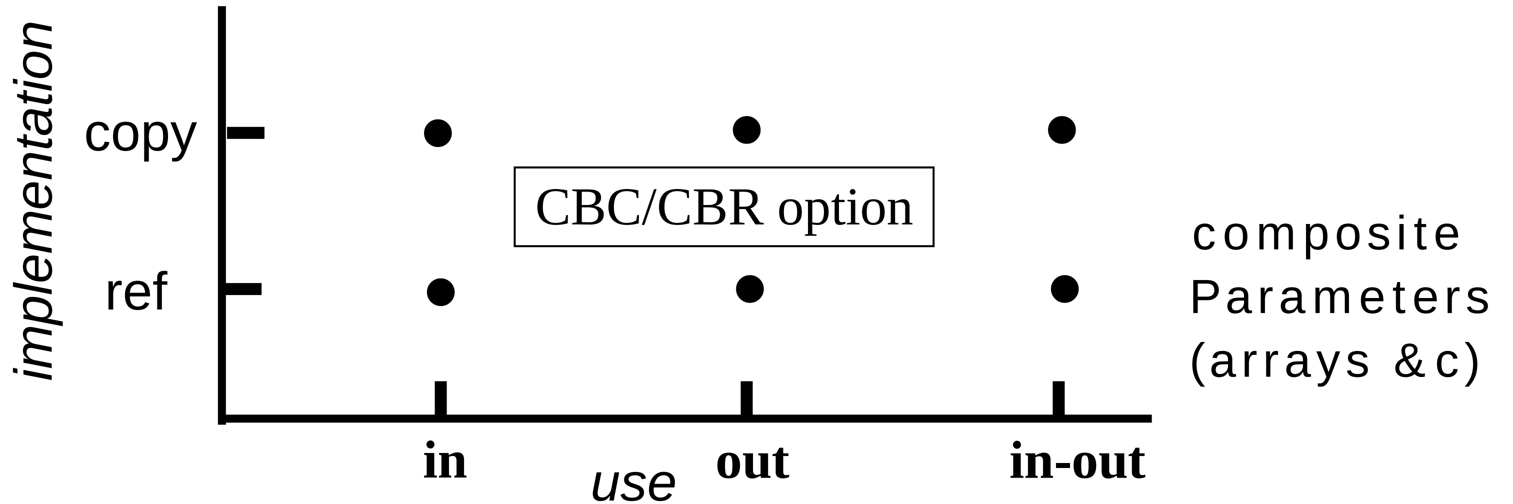
Modes in Programming Languages

• Call as constant	Algol 68, C/C++ **`const`**
• Call by reference	Pascal **var,** Fortran, Cobol, PL/I, Ada arrays, C arrays
• Call by value	Algol 60, Algol 68 (has **ref** *x*), Simula, C, C++ (has **&**x), Pascal, PL/I, APL, LISP, Snobol4, Ada (scalar **in** variables)
• Call by copy	Ada (scalar **out** or **in-out**)
• Call by value-result	Algol W
• Call by text	LISP Macros
• Call by name	Algol 60, Simula
• Call by need, R-value	Haskell, Scheme **`(delay x)`** **`&(force x)`**
• Call by need, L-value	imperative + lazy eval?

Ada Parameters

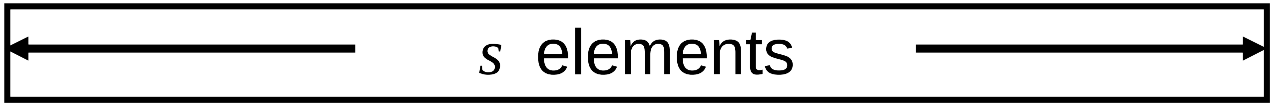
- Attempt at orthogonality of *use* vs *implementation*





Arrays: by ref or by copy?

a[]



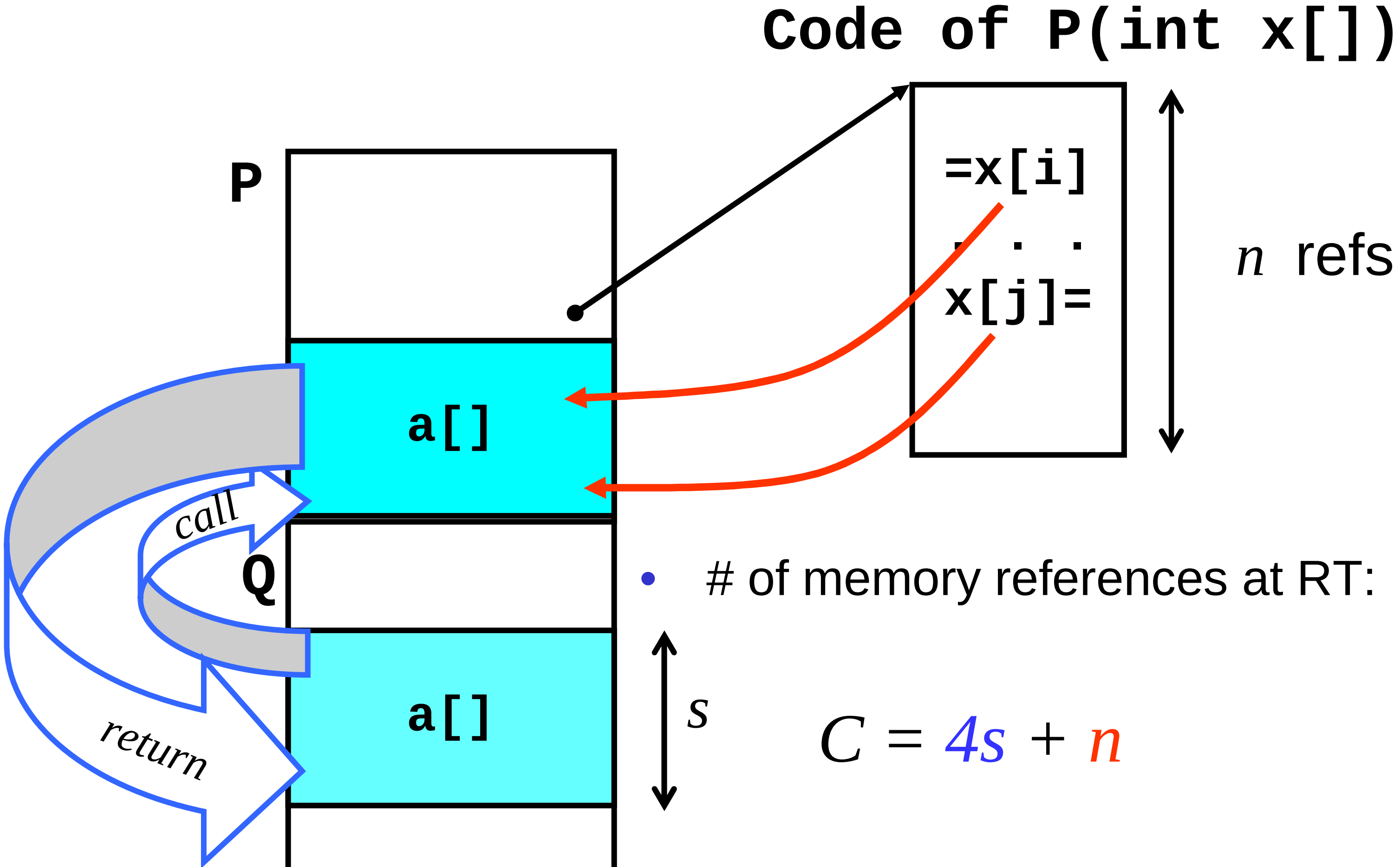

```
void function P(int x[])
 { … x[i1] … x[i2] . . . x[in] …  }
```

n references to elements of formal **x[]** at RT (*n* known at RT)

- Call by reference: copy a pointer to **a[]** into AR for **P(a[])**
  - Each reference **x[i]** becomes indirect address through pointer to original argument **a[]** in caller; loads and stores reflected immediately in original array
- Call by copy: copy argument **a[]** into AR for **P(a[])** & overwrite original **a[]** from copy on return
  - Each reference **x[i]** directly addresses copy**;** stores and loads made to copy

by ref or copy? (cont.)

- **`Q`** calls **`P(a[])`** <u>by copy</u>:

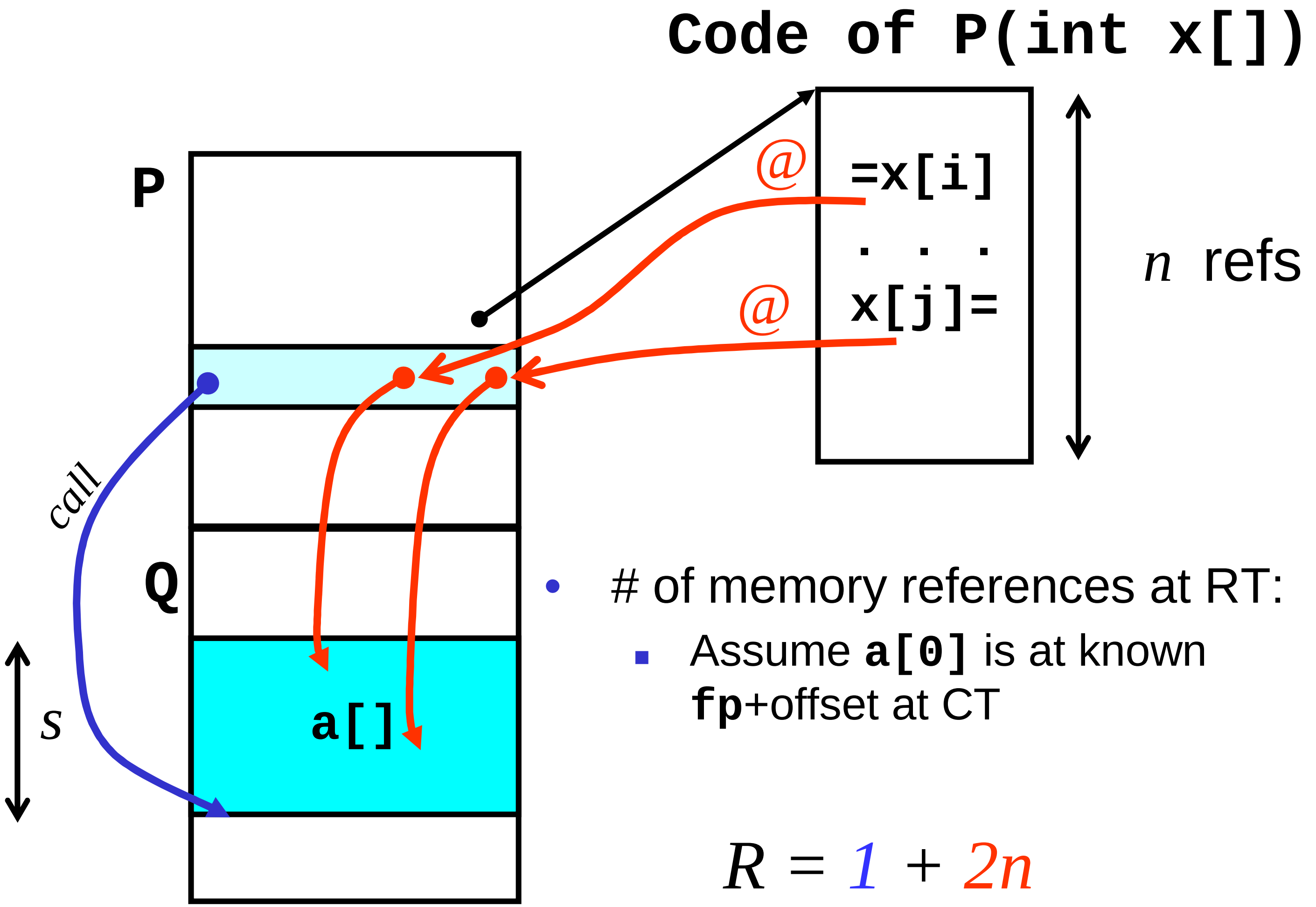


- # of memory references at RT:

$$C = 4s + n$$

by ref or copy? (cont.)

- **`Q`** calls **`P(a[])`** <u>by reference</u>:

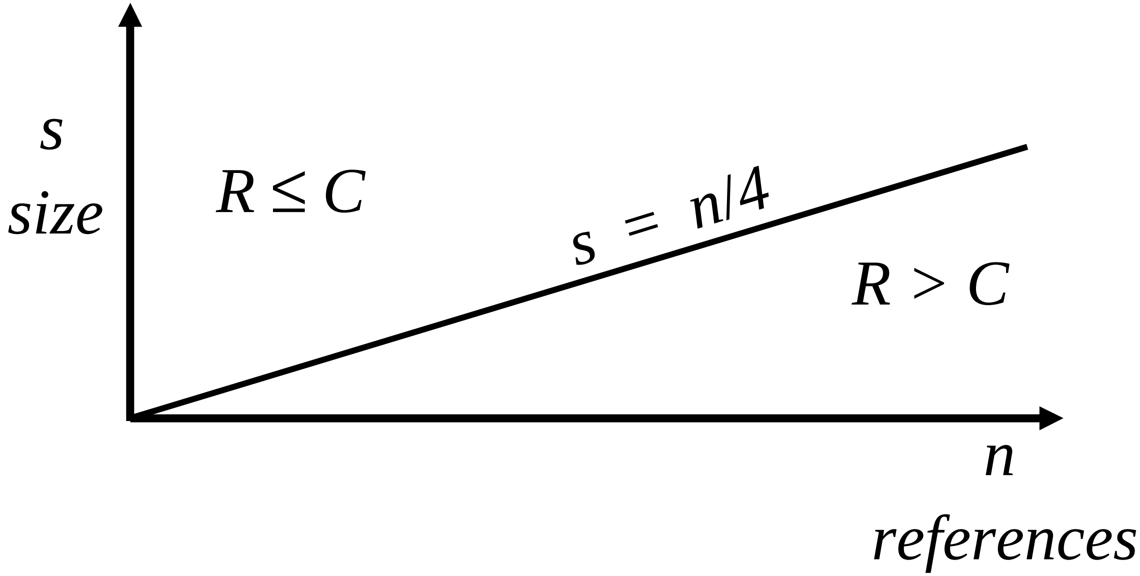


- # of memory references at RT:
  - Assume **`a[0]`** is at known **`fp`**+offset at CT

$$R = 1 + 2n$$

by ref or copy? (cont.)

- $R > C \iff 1 + 2n > 4s + n \iff n > 4s - 1 \iff n \geq 4s$



- Reference density: n / s = *ave. # references per element*
- Copying less expensive than reference:

$$R > C \iff n / s \geq 4$$

More on Call By Name

Most curious mode ever invented. Pros and Cons.

- CBN's main characteristics:
  - Defers evaluation of actual argument expressions
  - Re-evaluates the argument expression at each use of formal
    - So arg value can actually change from one ref to another
- Can simulate lazy evaluation

  example: short circuit booleans

```
boolean procedure cand(p,q);
  boolean p, q;
  if p then cand := q else cand := false;
end
```

- Cannot be written as a procedure in an "eager" programming language (like Scheme)

CBN (cont.)

- “Jensen’s Device” –weird sort of polymorphism

Example:

```
real procedure sigma(x, j, n);
  value n; real x; integer j, n;
  begin real s;
    s := 0;
    for j := 1 step 1 until n do s := s + x;
    sigma := s
  end
```

  - `sigma( a(i), i, 3)` ⇒ `a(1) + a(2) + a(3)`
  - `sigma( a(k)*b(k), k, 2)` ⇒ `a(1)*b(1) + a(2)*b(2)`

CBN (cont.)

Example:

```
begin comment zero vector
procedure zero(elt, j, lo, hi);
  value lo, hi; integer elt, j, lo, hi;
  begin
    for j := lo step 1 until hi do
       elt := 0;
  end
integer i; integer array a[1:10], b[1:5, 1:5];
zero(a[i], i, 1, 10);
zero(b[i, i], i, 1, 5);
end
```

CBN (cont.)

- Limitations of CBN: the obvious "swap" algorithm fails

```
begin
  procedure swap(a, b);
   integer a, b;
  begin integer t;
    t := b; b := a;
    a := t;
  end;
  integer array a[1:10];
   integer i;
  i := 1; a[i] := 2;
   // i=1 a[1]=2
  swap(a[i], i);
   //(a[1],i)=(2,1)
end
```

```
swap(a[i],i) //i==2 &
             //a[1]==2
 t:= i    // t==1
 i:=a[i]  // i==2
 a[i]:=t  // a[2]==1
        //(a[1],i)==(2,2)
```

- Reordering code doesn't help; `swap(i, a[i])` fails

CBN (cont.)

- Proposed "solution": assume in `t:=s` that the target l-value is computed before the source r-value

```
procedure swap(a, b); integer a, b;
 begin
 integer procedure testset(t,s); integer t,s;
  begin
    testset := t;     //ret. old r-val of target
    t := s;     //assign source r-val to target
  end;
 a := testset(b, a);
 end
swap(a[i], i) ⟹  a[i]:=testset(i,a[i]) ⟹
 ts:=i; i:=a[i]; a[i]:=ts
```

CBN (cont.)

- Problem: proposed “solution” has a bug.
  - Difficulty is the re-evaluation of actual on each use of formal
  - Can arrange an actual that yields a distinct l-value on each use

```
begin integer array a[1:10]; integer j;
   integer procedure i; comment return next integer;
     begin j:= j + 1; i := j end;
   j := 0; swap(a[i], a[i])
end
```

-
```
a[i] := testset(a[i], a[i])
ts :=a[i]; a[i]:=a[i]; a[i]:=ts;
ts :=a[1]; a[2]:=a[3]; a[4]:=ts;
```