Subprograms: Parameter Passing

Function: returns a value

Procedure: performs operations but do not return a value

Cf. Macro – code substitution

```
int  fred (a, b, c) {                    // formal parameters (parameters)
     a: in-out integer;
     b: in only float;
     c: out only Boolean;
       …
}

x = 2 * fred (x, y+z, a[3])              // actual parameters (arguments)
```

Parameter Passing Methods

① in-only methods
   call by value
② out-only methods
   call by result
③ in-out methods
   call by reference
   call by value-result
④ other methods
   call by name

[1] In-Only Methods: Call by Value

x	2
y	6
a[0]	5
a[1]	3
a[2]	1

call fred (x, a[0], a[2]+y)

subprogram fred (a: call by value,
b: call by value,
c: call by value)

fred's memory

a	2
b	5
c	7

When fred ends,

x	2
y	6
a[0]	5
a[1]	3
a[2]	1

Implemetnation cost:
memory allocation, parameter copy, memory deallocation

Note. C: call-by-value only
Array name is a pointer to the beginning of the array.
Passing array name is in effect a call by reference for the array.

[2] Out-Only Methods: Call by Result

- No copy of parameters
- At termination, formal parameters are copied into actual parameters

x	2
y	undef
a[0]	5
a[1]	undef
a[2]	1

call fred (x, y, a[2])

subprogram fred (a: call by result,
b: call by result,
c: call by result)

{ a = 6; b = 7; c = 8 }

When fred is called

x	2
y	undef
a[0]	5
a[1]	undef
a[2]	1

a	undef
b	undef
c	undef

When fred ends

x	2
y	undef
a[0]	5
a[1]	undef
a[2]	1

a	6
b	7
c	8

Final result:

x	6
y	7
a[0]	5
a[1]	undef
a[2]	8

Note.

If two of the actual parameters are the same, then call by result becomes unpredictable.

E.g. call fred (x, x)

```
subprogram fred (a: call by result,
                 b: call by result)
{
    a = 2;
    b = 3;
}
```

X will get either 2 or 3 depend on whether a or b is written back first.

[3] In-Out Methods: Call by Value-Result

x	2
y	6
a[0]	5
a[1]	3
a[2]	1

call fred (x, y, a[2])

subprogram fred (a: call by value-result,
b: call by value-result,
c: call by value-result)

{ a = b + c; c = 2 * a }

When fred is called

x	2
y	6
a[0]	5
a[1]	3
a[2]	1

fred's memory

a	2
b	6
c	1

When fred ends

x	2
y	6
a[0]	5
a[1]	3
a[2]	1

fred's memory

a	7
b	6
c	14

Final result:

x	7
y	6
a[0]	5
a[1]	3
a[2]	14

[4] In-Out Methods: Call by Reference

x	2
y	6
a[0]	5
a[1]	3
a[2]	1

call fred (x, y, a[2])

subprogram fred (a: call by reference,
b: call by reference,
c: call by reference)

{ a = b + c; c = 2 * a }

When fred is called

x, a	2
y, b	6
a[0]	5
a[1]	3
a[2], c	1

When fred ends

x, a	7
y, b	6
a[0]	5
a[1]	3
a[2], c	14

Final result:

x	7
y	6
a[0]	5
a[1]	3
a[2]	14

Note.

(1) Early Fortran - call by reference

```
call  fred (3, x)
```

Constant 3 could be changed to other value.

(2) Later Fortran - call by value-result

```
call  fred (3, x)
```

Constant 3 is not changed.

[5] Call by Name (Algol)

x	2
y	6
a[0]	5
a[1]	3
a[2]	1

```
call fred (x, y, a[x])

subprogram fred ( a:  call by name,
                  b:  call by name,
                  c:  call by  name)

{
    c = 3;
    a = 0;
    b = c;
}
```

Memory unchanged; Code changed.

```
Code:   {
            a[x] = 3;
            x = 0;
            y = a[x];
        }
```

Result:

x	0
y	5
a[0]	5
a[1]	3
a[2]	3

Implementation.

"Thunk" – parameterless procedure

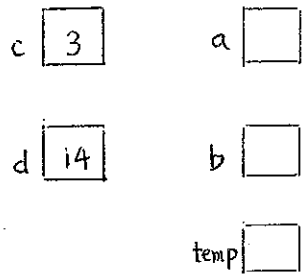
Parameter Passing in C (pass-by-value)

Problem: Swap c and d

(1)
```
void swap1 (int a, int b){
    int temp = a;
        a = b;
        b = temp;
}

swap1 (c, d);
```

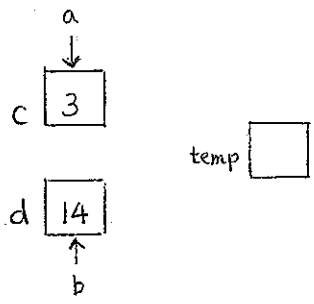
a = c
b = d
temp = a
a = b
b = temp



(2)
```
void swap2 (int *a, int *b){
    int temp = *a;
        *a = *b;
        *b = temp;
}

swap2 (&c, &d);
```

a = &c
b = &d
temp = *a
*a = *b
*b = temp



Function as a Parameter in Another Subprogram

Newton's formula for finding a root of $f(x) = 0$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

call Newtons (**f**, **fprime**, x_0) // x_0: initial guess

C: pass pointers to the function names.

Ex. $f(x) = x^2 - 2$ with $x_0 = 2$ // The root is the value of $\sqrt{2}$

```
#include <stdio.h>
#include <math.h>

float sqrt2(float x) {return(x * x - 2);}
float sqrt2prime(float x) {return(2 * x);}

float Newtons(float f(float), float fprime(float),
                                    float x0)          {
     float xnew = x0,
           xold;
     do  {
           xold = xnew;
           xnew = xold - f(xold) / fprime (xold);
           printf("%f\n", xnew);
     }  while (fabs(xnew - xold) > 0.000001);
     return (xnew);
}    /* end of Newton's method */

int main () {
    float root;
    root = Newtons(&sqrt2, &sqrt2prime, 2.0);
    printf("\nRoot is : %f\n", root);
}
```

```
1.500000
1.416667
1.414216
1.414214
1.414214
```

Generic Subprograms

Motivation.

Template

```
-------------------------------------------------
   subname(a: in out generic type T,
           b: out only generic type T,
           c: reference generic type S)
   {
       int x;
       generic type T: y;
       ...
   }
-------------------------------------------------
```

call subname (p, q, r)

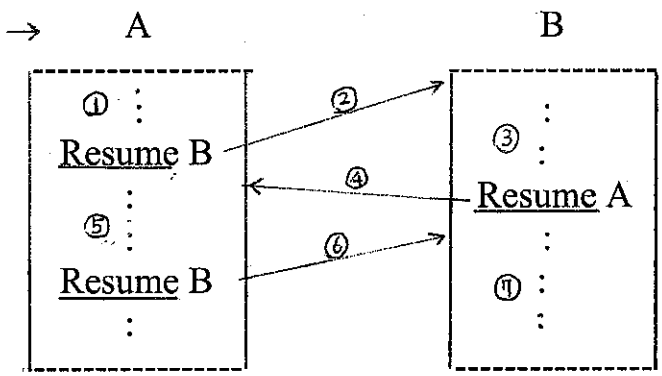
if p and q are integers, a,b,y: integer
if p and q are floats, a,b,y: float
if p is integer, q is float → error

Ada, C++, Java 5.0, C#

Coroutine

- Melvin Conway (1963) cf. threads

cooperating equals – not master/slave
interleaved execution
multiple entry points
history-sensitive – static local variables
"resume" instead of "call"
can be simulated by storing resume point on activation record
Simula-67, Modula-2, Python, Ruby



Example.

1. Card games – Blackjack, Poker, Bridge
2. Producer-Consumer

```
var q := new queue

coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```