Typology of programming languages

Evaluation strategy

Argument passing

From a naive point of view (and for **strict evaluation**), three possible modes: `in`, `out`, `in-out`. But there are different flavors.

	Val	ValConst	RefConst	Res	Ref	ValRes	Name
ALGOL 60	*						*
Fortran					?	?	
PL/1					?	?	
ALGOL 68		*			*		
Pascal	*				*		
C	*	?			?		
Modula 2	*				?		
Ada (simple types)		*		*		*	
Ada (others)		?	?	?	?	?	
Alphard		*	*		*		

Table of Contents

Call by Value – definition

Passing arguments to a function copies the actual value of an argument into the formal parameter of the function.

In this case, changes made to the parameter inside the function have no effect on the argument.

```
def foo(val):
    val = 1
i = 12
foo(i)
print (i)
```

Call by value in Python – **output: 12**

Pros & Cons

- **Safer**: variables cannot be accidentally modified
- **Copy**: variables are copied into formal parameter *even for huge data*
- **Evaluation before call**: resolution of formal parameters must be done before a call
  - Left-to-right: Java, Common Lisp, Effeil, C#, Forth
  - Right-to-left: Caml, Pascal
  - Unspecified: C, C++, Delphi, , Ruby

Table of Contents

Call by Reference – definition

Passing arguments to a function copies the actual address of an argument into the formal parameter.
In this case, changes made to the parameter inside the function will have effect on the argument.

```
void swap(int &x, int &y){
  int t = x; x = y;  y = t;
}
int main() {
  int x = 2, y = 3;
  swap(a, b);
  printf("%d, %d\n", x, y);
}
```

Call by reference in C++ – **output: 3 2**

Pros & Cons

- **Faster** than call-by-value if data structure have a large size.
- **Readability & Undesirable behavior**: a special attention may be considered when doing operations on multiple references **since they can all refer to the same object**

```
void xor_swap(int &x, int &y) {
  x = x ^ y;
  y = y ^ x;
  x = x ^ y;
}
```

Undesirable behavior when x and y refers the same object (zeroing x and y)

Note on call-by-reference

$swap(foo, foo)$ is forbidden in Pascal but what about $swap(foo[bar], foo[baz])$ ...

Table of Contents

Call by Value-Result – definition

Passing arguments to a function copies the argument into the formal parameter of the function.
The values are then copied back when exiting the function

In this case, changes made to the parameter inside the function will only reflect on the argument at the end of the function.

Call by Value-Result – Example

```
procedure Tryit is
   procedure swap (i1, i2: in out integer) is
      tmp: integer;
   begin
       tmp := i1;   i1  := i2;   i2  := tmp;
   end swap;

a : integer := 1;   b : integer := 2;
begin
  swap(a, b);
  Put_Line(Integer'Image (a) & " " &
           Integer'Image (b)) ;
end Tryit;
```

Call by Value-result in Ada – **output: 2 1**

Pros & Cons

- **Safety** other thread will only see consistent values since changes made will not show up until after the end of the function.

- **Local copies**: but they can be sometimes avoided by the compiler

Notes on call-by-value-result

- Also called: **Call by copy-restore**, **Call by copy-in copy-out**
- If the reference is passed to the callee uninitialized, this evaluation strategy is called **call by result**.
- Used in multiprocessing contexts.
- Multiple interpretations:
  - Ada: Evaluates arguments once, during function call
  - AlgolW: Evaluates arguments during call **AND** when exiting the function

Table of Contents

An outsider: call by name

(In ALGOL 60) It behaves as a macro would, including with name captures: the argument is evaluated *at each use*.

- Try to write some code which results in a completely different result had SWAP been a function.

```
#define SWAP(Foo, Bar) \
  do {                 \
    int tmp_ = (Foo);  \
    (Foo) = (Bar);     \
    (Bar) = tmp_;      \
  } while (0)
```

- ALGOL 60 introduced “thunks” : snippets of code that return the l-value when evaluated

An application of call by name: Jensen's Device

- General computation of a sum of a series $\sum_{k=l}^{u} a_k$:

```
real procedure Sum(k, l, u, ak)
      value l, u;
      integer k, l, u;
      real ak;
      comment `k' and `ak' are passed by name;
   begin
      real s;
      s := 0;
      for k := l step 1 until u do
         s := s + ak;
      Sum := s
   end;
```

- Computing the first 100 terms of a real array V[]:

```
Sum(i, 1, 100, V[i])
```

Table of Contents

Call by Need

Call by need is a memoized variant of call by name where, if the function argument is evaluated, that value is stored for subsequent uses.

The argument is then evaluated only once, during its first use.

What if $y = 0$ in the following code?

```
let function loop (z: int):int =
        if z > 0 then z else loop (z)
    function f    (x: int):int =
        if y > 8 then x else -y
in
  // if y > 8 then loop (y) else -y ?
  f (loop (y))
end
```

Call by name vs. Call by need

Call by name

Don't pass the evaluation of the expression, but a "thunk" computing it:

```
let var a := 5 + 7 in
    a + 10
end
==>
let function a () := 5 + 7 in
   a () + 10
end
```

Call by need

The thunk is evaluated once and only once. Add a "memo" field.

Lazy evaluation

```
easydiff f x h = (f (x + h) - f (x)) / h

repeat f a = a : repeat f (f a)
halve x = x / 2
differentiate h0 f x = map (easydiff f x) (repeat halve h0)

within eps (a : b : rest)
  | abs (b - a) <= eps = b
  | otherwise          = within eps (b : rest)

relative eps (a : b : rest)) =
  | abs (b - a) <= eps * abs b = b
  | otherwise                  = relative eps (b : rest)
within eps (differentiate h0 f x)
```

Slow convergence...Suppose the existence of an error term:

```
a (i)     = A + B * (2 ** n) * (h ** n)
a (i + 1) = A + B * (h ** n)
```

Table of Contents

Exhibit the differences (Explicit lyrics...)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Exhibit the differences (Explicit lyrics...)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val			

Exhibit the differences (Explicit lyrics...)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)			

Exhibit the differences (Explicit lyrics...)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)			

Exhibit the differences (Explicit lyrics...)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Mode	`foo[1]`	`foo[2]`	`t`
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)	4	2	2
Ref			

Exhibit the differences (Explicit lyrics...)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)	4	2	2
Ref	9	2	2
Name			

Exhibit the differences (Explicit lyrics...)

```
var t   : integer
    foo: array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 6;
  t      := 2;
  x      := x + 3;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo[t]);
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	2
Val-Res (ALGOL W)	6	4	2
Val-Res (Ada)	4	2	2
Ref	9	2	2
Name	6	5	2

Table of Contents

Call by Sharing – definition

Call by sharing implies that values in the language are based on objects rather than primitive types, i.e. that all values are ”boxed”

Differs from both call-by-value and call-by-reference.

call by sharing is not in common use; the terminology is inconsistent across different sources.

```
def f(list):
    list.append(1)

m = []
f(m)
print(m)
```

Call by sharing in Python – **output: [1]**

```
def f(list):
    list = [1]

m = []
f(m)
print(m)
```

Call by sharing in Python – **output: []**

Notes on Call-by-sharing

Mutations of arguments perfors mall by
the called routine
will be visible to the caller.

Notes on Call-by-sharing

Mutations of arguments perfors mall by the called routine
will be visible to the caller.

Access is not given to the variables of the caller,
but merely to certain objects

Notes on Call-by-sharing

Mutations of arguments perfors mall by the called routine
will be visible to the caller.

Access is not given to the variables of the caller,
but merely to certain objects

Can be seen as ”call by value” in the case where the value is an object reference

Notes on Call-by-sharing

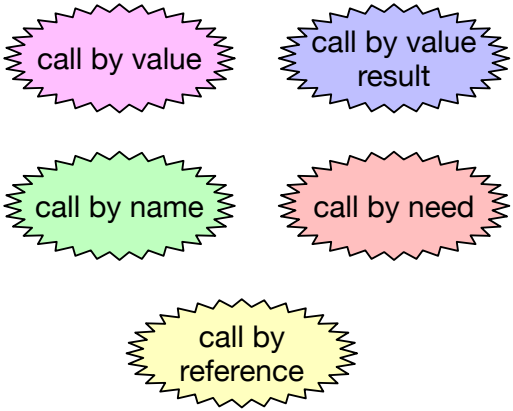
Mutations of arguments perfors mall by the called routine
will be visible to the caller.

Access is not given to the variables of the caller,
but merely to certain objects

Can be seen as ”call by value” in the case where the value is an object reference

- First introduced by Barbara Liskov for CLU language (1974)
- Widely used by: Python, Java, Ruby, JavaScript, Scheme, OCaml, ...

Summary


call by value
call by value result
call by name
call by need
call by reference