Typology of Programming Languages

Subprograms

May 2025

Section 1

What is a subprogram?

Exercise

Why are functions useful for the programmer?

Exercise

Why are functions useful for the programmer?

- Good software engineering
  - “DRY”
  - No duplication of errors
  - Easier to read
  - …
- Modular programming
- Separated compilation

Fundamentals

Definition

A **subprogram** is a *callable* unit of code with a defined interface and behavior.

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates

Synonyms: *routine*, *subroutine*.

Procedures vs Functions

Definition

Procedure:

- Collection of statements that define parameterized computations
- Subprograms with no return value
- **Procedures (necessarily) have side effects**

Definition

Function:

- Structurally resemble procedures but semantically modeled on mathematical functions
- Subprograms with a return value
- **(Pure) Functions do not have side effects**

Languages may or may not distinguish functions and procedures.

Procedures vs Functions in Pascal

Ada and Pascal have two reserved keywords `procedure` and `function` where functions have return values while procedures do not.

```
// Pascal
Function Add(A, B : Integer) : Integer;
Begin
  Add := A + B;
End;

Procedure Finish(Name : String);
Begin
  WriteLn('Goodbye ', Name);
End;
```

Procedures vs Functions in C/C++

C and C++ have the same syntax for functions and procedures (`void` functions).

```cpp
// C++
int add(int a, int b) {
  return a + b;
}

void finish(std::string name) {
  std::cout << "Goodbye " << name << std::endl;
}
```

`finish` is still a procedure because it doesn't have a return value.

Procedures vs Functions in Rust/OCaml

Rust, OCaml, Python… arguably have no procedures, only functions returning a “null value” (the `unit`/`None` type).

```ocaml
(* OCaml *)
let add a b = a + b


let finish name =
  Format.printf "Goodbye %s@." name

let () : unit =
  let x : unit = finish "Tigrou" in
  x
```

```rust
// Rust
fn add(a: i64, b: i64) -> i64 {
  a + b
}
fn finish(name: &str) {
  println!("Goodbye {}", name)
}
fn main() {
  let x : () = finish("Tigrou");
}
```

Hybridation: Procedures/Functions

Using functions with side effects can be dangerous and counter-intuitive.

Exercise

```c
// C
foo = getchar() + getchar() * getchar();
```

What's the result?

Hybridation: Procedures/Functions

Using functions with side effects can be dangerous and counter-intuitive.

Exercise

```c
// C
foo = getchar() + getchar() * getchar();
```

What's the result?

Undefined ($\not=$ nondeterministic)! This is *on purpose*!

Methods

Multiple competing definitions for methods.

Definition

A **method** is a procedure associated to an entity, which it implicitly takes as argument.

- Most often encountered in OOP, but not limited to it.
  - Go, Zig, Rust… have methods but are not object-oriented.
- Often perform dynamic dispatch on the first argument (“this”, or “self” or…).
  - Multimethods are a generalization of this aspect and provide dynamic dispatch on multiple arguments.

Methods in Go

```go
// Go
type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{3, 4}
    v.Scale(10) // Implicitly take as pointer
    fmt.Println((&v).Abs()) // Implicitly dereference
}
```

Anonymous functions

Languages may or may not require functions to have a name.

- A function with no name is **anonymous**.
- Languages with anonymous functions may consider *named functions* as a simple sugar.

```lua
-- Lua
local function f()
  print("f!")
end
```

```lua
-- Lua
local f = function ()
  print("f!")
end
```

Function pointers

Von Neumann architecture: both data and program are stored in the memory.

- …so the code for a function is itself in memory and thus has an address.
- A function is, at its core, simply an address *i.e.* a pointer!

Function pointers

Von Neumann architecture: both data and program are stored in the memory.

- …so the code for a function is itself in memory and thus has an address.
- A function is, at its core, simply an address *i.e.* a pointer!

```c
// C
void f(void) { puts("f!"); }
void g(void) { puts("g!"); }

int main(int argc, char** argv) {
  (void) argv;
  printf("&f:    %p\n", &f);
  printf("g:     %p\n", g); // Same as &g
  printf("main:  %p\n", main); // Why not!
  printf("heap:  %p\n", malloc(42));
  printf("&argc: %p\n", &argc);

  return 0;
}
```

Function pointers

Von Neumann architecture: both data and program are stored in the memory.

- …so the code for a function is itself in memory and thus has an address.
- A function is, at its core, simply an address *i.e.* a pointer!

```c
// C
void f(void) { puts("f!"); }
void g(void) { puts("g!"); }

int main(int argc, char** argv) {
  (void) argv;
  printf("&f:    %p\n", &f);
  printf("g:     %p\n", g); // Same as &g
  printf("main:  %p\n", main); // Why not!
  printf("heap:  %p\n", malloc(42));
  printf("&argc: %p\n", &argc);

  return 0;
}
```

```
f:     0x5d2e7c3fd189
g:     0x5d2e7c3fd1a3
main:  0x5d2e7c3fd1bd
heap:  0x5d2eaf8936b0
&argc: 0x7ffe685dc97c
```

Function objects

Definition

A **function object** is an object that can behave like a function, *i.e.* it is callable.

This allows for “functions” with *state*, preserved from one call to another.

```
// C++
struct Adder {
  int x;

  int operator()(int y) {
    return ++x + y;
  }
};

Adder add{0};
add(20); // returns 21
add(40); // returns 42
```

Nested subprograms

```ada
-- Ada
procedure one is
  A, B : Integer;

  function two(I : Integer)
      return Integer is
    function three(I : Integer)
      return Integer is
    begin
      return I;
    end three;
  begin
    return three(I);
  end two;
begin
  -- main code here
end one;
```

A subprogram defined *within* another subprogram.

- Organize your programs hierarchically.
  - Nested subprograms cannot be accessed outside their enclosing subprogram.
- Share state easily and precisely.
  - Nested subprograms have access to the parameters, local variables… declared in the outer subprogram(s)

Un-nesting subprograms

Exercise

Translate this OCaml function to C:

```
(* OCaml *)
let print_to x =
  let rec aux start limit =
    if start < end then (
      print_int start;
      aux (start + 1) end
    )
  in
  aux 0 (x + 1)
```

Un-nesting subprograms

Exercise

Translate this OCaml function to C:

```
(* OCaml *)
let print_to x =
  let rec aux start limit =
    if start < end then (
      print_int start;
      aux (start + 1) end
    )
  in
  aux 0 (x + 1)
```

```
// C
void aux(int start, int limit) {
  if (start < limit) {
    print("%d", start);
    aux(start + 1, limit);

  }
}
void print_to(int x) {
  aux(0, x + 1);
}
```

Un-nesting subprograms

Exercise

What about this function:

```
(* OCaml *)
let print_to x =
  let rec aux start =
    if start < x + 1 then (
      print_int start;
      aux (start + 1)
    )
  in
  aux 0
```

If we unnest aux, how can it access x?

Un-nesting subprograms

Exercise

What about this function:

```
(* OCaml *)
let print_to x =
  let rec aux start =
    if start < x + 1 then (
      print_int start;
      aux (start + 1)
    )
  in
  aux 0
```

If we unnest aux, how can it access x?

```
// C
void aux(int start, int x) {
  if (start < x + 1) {
    print("%d", start)

  }
}
void print_to(int x) {
  aux(0, x);
}
```

- This is called *lambda lifting*.
- We might need to take x as a pointer to allow for mutability.
  - Not needed here though since variables are not mutable in OCaml.

Un-nesting subprograms

Exercise

What about this:

```
(* OCaml *)
let add a b c ... y =
  let aux z =
    a + b + c ... + y + z
  in aux

let my_add = add 1 2 3 4 ...
let the_answer = my_add 42
```

Lamdba lifting such cases would be too complex, we need something else.

Un-nesting subprograms

Exercise

What about this:

```
(* OCaml *)
let add a b c ... y =
  let aux z =
    a + b + c ... + y + z
  in aux

let my_add = add 1 2 3 4 ...
let the_answer = my_add 42
```

Lamdba lifting such cases would be too complex, we need something else.

```
// C++
int aux(env& e, int z) {
  return
    e.a + e.b + e.c +
    ... + e.y + e.z;
}

std::pair<env, int(*)(env&, int)>
add(int a, int b, ..., int y) {

  return {
    env{a, b, ..., y},
    aux,
  };
}

int main() {
  auto my_add = add(1, 2, 3, 4, ...);
  auto [my_add_env, my_add_fun] = my_add;
  int the_answer = my_add_fun(my_add_env, 42);
}
```

Closures

Definition

A **closure** is a function capturing *non-local* variables. The set of captured variables is called the *environment*.
Closures can be thought of as a pair `{function_ptr, env}`.

```python
# Python
def add(x):
  def aux(y):
    return x + y

  return aux
```

- x is defined in add but used by aux
  - "x *escapes* add"
  - "x is an *upvalue* of aux"
  - "x is *non-local* to aux"
  - "x is a *free variable* in aux"
- aux is a closure with environment {x}

Lambdas

Definition

A **lambda** often refers to a combination of anonymous function and closure.

Exercise

```cpp
// C++
int x;
double y;
auto lambda =
  [x, &y](int a) -> float {
    return a + x * y;
  }
```

C++ lambdas are actually simple syntactic sugar!
What does this desugar to?

Lambdas

Definition

A **lambda** often refers to a combination of anonymous function and closure.

Exercise

```cpp
// C++
int x;
double y;
auto lambda =
  [x, &y](int a) -> float {
    return a + x * y;
  }
```

C++ lambdas are actually simple syntactic sugar!
What does this desugar to?

```cpp
// C++
int x;
double y;
auto lambda = struct {
  int x;
  double& y;

  float operator()(int a) {
    return a + x * y;
  }
}{x, y};
```

Section 2

Arguments

Vocabulary

- **Formal Argument**: Argument in a subprogram declaration.

```
function sum(x: int, y: int): int = x + y
```

- **Effective Argument**: Argument in a call to a subprogram.

```
sum (30, 12)
```

- **Parameter**: Better reserved for templates, generics…

```
function sort<t>(x: array of t) = ...
type list<t> = { head: t, tail: list<t> }
```

Partial applications

Definition

Partial application is the process of specifying only some arguments of a function. The result is another function, which takes the remaining arguments.

```scala
// Scala
def sum(n1: Int, n2: Int, n3: Int) = n1 + n2 + n3

val partialSum: (Int, Int) => Int = sum(2, _, _)

val sum: Int = partialSum(3, 5)
println(sum) // 10

val partialSum2: (Int, Int) => Int = sum(_, 2, _)

val sum2: Int = partialSum2(3, 5)
println(sum2) // 10
```

Currying

Definition

Currying translates a function with multiple arguments to a sequence of functions with a single argument.

```
(* OCaml *)
let sum a b c = a + b + c

(* val sum : int -> int -> int -> int *)

let sum = fun a -> fun b -> fun c -> a + b + c

(* val sum : int -> int -> int -> int *)
```

Currying allows for trivial partial application.

```
let sum42 = sum 42

(* val sum42 : int -> int -> int *)
```

Default arguments

```cpp
// C++
int sum(int a, int b = 21, int c = 42, int d = 42) {
   return a + b + c + d;
}
```

- `sum(1, 2, 3, 4)` is fine.
- `sum(1, 2)` is also fine.

Default arguments

```cpp
// C++
int sum(int a, int b = 21, int c = 42, int d = 42) {
   return a + b + c + d;
}
```

- `sum(1, 2, 3, 4)` is fine.
- `sum(1, 2)` is also fine.
- But what if we want to call `sum` with `a=1` and `c=2` while *keeping `b` and `d`'s default value*?

Arguments in Ada

Default *and* named arguments.

```ada
-- Ada
put(
  number   : in float;
  before   : in integer := 2;
  after    : in integer := 2;
  exponent : in integer := 2
) ...
```

```ada
-- Ada
put(pi, 1, 2, 3);
put(pi, 1);
put(pi, 2, 2, 4);
```

Arguments in Ada

Default *and* named arguments.

```ada
-- Ada
put(
  number   : in float;
  before   : in integer := 2;
  after    : in integer := 2;
  exponent : in integer := 2
) ...
```

```ada
-- Ada
put(pi, 1, 2, 3);
put(pi, 1);
put(pi, 2, 2, 4);
put(pi, before => 2,
   after => 2,
   exponent => 4);
put(pi, exponent => 4);
```

Named Arguments

Named parameters are availables in many languages: Perl, Python, C#, Fortran95, Go, Haskell, Lua, OCaml, Lisp, Scala, Swift…

- No need to remember the order of parameters (except in Swift!)
- No need to guess specific *default* values
- More flexibility, more clarity

Named Arguments

Exercise

How can we simulate *named arguments* in C?

Named Arguments

Exercise

How can we simulate *named arguments* in C?

Use struct litterals!

```c
// C
struct Arg {
  int a;
  int b;
};

void f(struct Arg args) { ... }
```

Named Arguments

Exercise

How can we simulate *named arguments* in C?

Use struct litterals!

```c
// C
struct Arg {
  int a;
  int b;
};

void f(struct Arg args) { ... }


void call_f(void) {
  f((struct Arg){.a = 42, .b = 51});
}
```

Named Arguments

Exercise

How can we simulate *named arguments* in C?

Use struct litterals!

```c
// C
struct Arg {
  int a;
  int b;
};

void f(struct Arg args) { ... }
```

```c
void call_f(void) {
  f((struct Arg){.a = 42, .b = 51});
}
```

```c
#define F(...) f((struct Arg){__VA_ARGS__})

void call_f(void) {
  F(.b = 51, .a = 42);
}
```

Named Parameter Idiom

Exercise

How can we simulate *named arguments* in C++ or Java?

Named Parameter Idiom

Exercise

How can we simulate *named arguments* in C++ or Java?

```
// Java
class foo_param {
private:
    int a = 0, b = 0; // default values
    foo_param() = default; // make it private

public:
    foo_param& with_a(int provided) {
        a = provided; return *this;
    }
    foo_param& with_b(int provided) {
        b = provided; return *this;
    }
    static foo_param create() {
        return foo_param();
    }
};
```

Named Parameter Idiom

```
// Java
void foo(foo_param& f) {
  // ...
}

foo(
    foo_param::create()
        .with_b(1)
        .with_a(2)
);
```

Very similar to a *builder* pattern!

Works… but requires one specific class per function.

For C++, `Boost::Parameter` library also offer a generic implementation

Bitten by Python

```python
# Python
def f(data: list[int] = []):
    data.append(1)
    print(data)

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

What's the resulting output?

Bitten by Python

```python
# Python
def f(data: list[int] = []):
    data.append(1)
    print(data)

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

What's the resulting output?

"Obviously":

```
[1]
[1, 1]
```

Section 3

Evaluation Strategies

Argument passing

Exercise

```cpp
// C++
int double1(int x) {
  return x * 2;
}
int double2(int* x) {
  return *x * 2;
}
int double3(int& x) {
  return x * 2;
}
```

What's the difference between these functions?

- `double1` takes `x` *by value*
  - modifications of `x` within the function are *not* propagated to the caller function.
- `double2` and `double3` take `x` *by reference*
  - modifications of `x` within the function *are* propagated to the caller function.

Argument passing

Naively, three possible modes: `in`, `out` and `in-out`.

```
-- Ada
procedure proc_in(in x : Integer) is
begin
  print(x);
  -- x := 32; -- Error!
end;
procedure proc_out(out x : Integer) is
begin
  -- print(x); -- Warning!
  x := 32;
end;
procedure proc_inout(in out x : Integer) is
begin
  print(x); -- OK because in.
  x := 32;  -- OK because out.
end;
```

Argument passing

Exercise

Examples of in, out and in-out in C++?

Argument passing

Exercise

Examples of in, out and in-out in C++?

```
void in(int x) {
  print(x);
}

void in(const int& x) {
  print(x);
}
```

```
void out(int* x) {
  x = 32;
}

void inout(int* x) {
  print(*x);
  *x = 32;
}
```

Argument passing

Exercise

Examples of in, out and in-out in C++?

```
void in(int x) {
  print(x);
}

void in(const int& x) {
  print(x);
}
```

```
void out(int* x) {
  x = 32;
}

void inout(int* x) {
  print(*x);
  *x = 32;
}
```

- C++ does not actually support `out` arguments, but they can be emulated using `in-out`.
- Two distinct `in` examples: different flavors for these naive modes!

Call by Value – Definition

```c
// C
void foo(int val) {
    val += 1;
}

int main(void) {
    int i = 12;
    foo(i);
    printf("%d\n", i);
    return 0
}
```

Call by value in C – **output:**

Call by Value – Definition

```c
// C
void foo(int val) {
    val += 1;
}

int main(void) {
    int i = 12;
    foo(i);
    printf("%d\n", i);
    return 0
}
```

Call by value in C – **output:**
12

Call by Value – Definition

```c
// C
void foo(int val) {
    val += 1;
}

int main(void) {
    int i = 12;
    foo(i);
    printf("%d\n", i);
    return 0
}
```

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

Changes made to the parameter inside the function have no effect on the argument.

Call by value in C – **output:**
12

Call by Value – Pros & Cons

Pros:

- **Safer**: variables cannot be accidentally modified

Cons:

- **Copy**: variables can be 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

Call by Value – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t : ??? integer);
begin
  t      := 2;
  foo[t] := x + t;
  x      := t;
  x      := t;
end;

begin
  x      := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x]);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t` by **value**?

Call by Value – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t : ??? integer);
begin
  t      := 2;
  foo[t] := x + t;
  x      := t;
  x      := t;
end;

begin
  x      := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x]);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t` by **value**?

```
x      = 2
foo[1] = 1
foo[2] = 3
```

Call by Reference – Definition

```cpp
// C++
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:**

Call by Reference – Definition

```cpp
// C++
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

Call by Reference – Definition

```
// C++
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

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.

Call by Reference – Pros & Cons

Pros:

- **Faster** than call-by-value if data structure have a *large size*.
  - Please, do not pass integers by reference for "performance" reasons.

Cons:

- **Readability & Undesirable behavior**: a special attention may be considered when doing operations on multiple references *since they can all refer to the same object*

Call by Reference – Pros & Cons

Pros:

- **Faster** than call-by-value if data structure have a *large size*.
  - Please, do not pass integers by reference for "performance" reasons.

Cons:

- **Readability & Undesirable behavior**: a special attention may be considered when doing operations on multiple references *since they can all refer to the same object*

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

Call by Reference – Pros & Cons

Pros:

- **Faster** than call-by-value if data structure have a *large size*.
  - Please, do not pass integers by reference for "performance" reasons.

Cons:

- **Readability & Undesirable behavior**: a special attention may be considered when doing operations on multiple references *since they can all refer to the same object*

```
// C
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)…

Call by Reference – Pros & Cons

Pros:

- **Faster** than call-by-value if data structure have a *large size*.
  - Please, do not pass integers by reference for "performance" reasons.

Cons:

- **Readability & Undesirable behavior**: a special attention may be considered when doing operations on multiple references *since they can all refer to the same object*

```
// C
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)…

`swap(foo, foo)` is forbidden in Pascal.

Call by Reference – Pros & Cons

Pros:

- **Faster** than call-by-value if data structure have a *large size*.
  - Please, do not pass integers by reference for "performance" reasons.

Cons:

- **Readability & Undesirable behavior**: a special attention may be considered when doing operations on multiple references *since they can all refer to the same object*

```
// C
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)…

`swap(foo, foo)` is forbidden in Pascal.

But what about `swap(foo[bar], foo[baz])`…?

Call by Reference – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t : ??? integer);
begin
  t      := 2;
  foo[t] := x + t;
  x      := t;
  x      := t;
end;

begin
  x      := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x]);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t` by **reference**?

Call by Reference – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t : ??? integer);
begin
  t      := 2;
  foo[t] := x + t;
  x      := t;
  x      := t;
end;

begin
  x      := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x]);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t` by **reference**?

```
x      = 2
foo[1] = 2
foo[2] = 3
```

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.

```ada
-- Ada
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:**

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.

```ada
-- Ada
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

Call by Value-Result – Pros & Cons

Pros:

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

Cons:

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

Call by Value-Result – Pros & Cons

Pros:

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

Cons:

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

Notes on call-by-value-result

- Also called: *copy-restore*, *copy-in copy-out*
- If the reference is passed to the callee uninitialized, this is called *call by result*.
  - "`out`" mode in Ada
- Useful in multiprocessing contexts.
- Multiple interpretations:
  - Ada: Evaluates arguments once, at call site
  - AlgolW: Evaluates arguments at call site *AND* when exiting the function

Call by Value-Result – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t1, t2 : ??? integer);
begin
  t2      := 2;
  foo[t1] := x + t2;
  x       := t1;
  x       := t1;
end;

begin
  x       := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x], x);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t1, t2` by **value-result à la Ada**? (*i.e.* arguments are evaluated once, when calling the function)

Call by Value-Result – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t1, t2 : ??? integer);
begin
  t2      := 2;
  foo[t1] := x + t2;
  x       := t1;
  x       := t1;
end;

begin
  x       := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x], x);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t1, t2` by **value-result à la Ada**? (*i.e.* arguments are evaluated once, when calling the function)

```
x      = 2
foo[1] = 1
foo[2] = 2
```

An outsider: Call by Name

What is this C/C++ macro's argument passing mode?

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

An outsider: Call by Name

What is this C/C++ macro's argument passing mode?

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

```
int &a() { ... }
int &b() { ... }

SWAP(a(), b());
```

An outsider: Call by Name

What is this C/C++ macro's argument passing mode?

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

```
int &a() { ... }
int &b() { ... }

SWAP(a(), b());
```

a and b are **evaluated twice**!

With call-by-name, arguments are *evaluated every time they are used.*

Introduced in Algol 60

- Behaves similarly to macros, including name captures: the argument is evaluated *at each use.*
- Based on *"thunks"*: snippets of code that return the l-value when evaluated.

```
let var a = 5 + 7 in
    a + 10
end
```

↓

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

Call by Name – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t : ??? integer);
begin
  t      := 2;
  foo[t] := x + t;
  x      := t;
  x      := t;
end;

begin
  x      := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x]);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t` by **call-by-name**?

Call by Name – Example

```
(* Global values *)
var x: integer;
var foo: array [1..2] of integer;

procedure f(t : ??? integer);
begin
  t      := 2;
  foo[t] := x + t;
  x      := t;
  x      := t;
end;

begin
  x      := 1;
  foo[1] := 1;
  foo[2] := 2;
  f(foo[x]);
end;
```

Exercise

What are the values of `x`, `foo[1]`, `foo[2]` at the end of the program if passing `t` by **call-by-name**?

```
x      = 3
foo[1] = 2
foo[2] = 3
```

An application of call-by-name: Jensen's Device in Algol 60

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

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

An application of call-by-name: Jensen's Device in Algol 60

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

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

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

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

Call by Name – Pros & Cons

Pros:

- **Flexible & Expressive**: can be used to write elegant code somewhat easily.

Cons:

- **Poor performances**: if optimized poorly, can be costly since every variable access is now a call to a (potentially non-trivial) function.
- **Unusual**: very rare approach in languages, can be confusing.

Call by Need – Definition

Memoized variant of call by name where, if the function argument is evaluated, its value is stored for subsequent uses.

The argument is **evaluated only once**: at the site of its *first use*.

Most commonly implemented through *lazy evaluation*.

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
    f(loop(y))
end
```

(In a fictional, Tiger-like language using call-by-need)

Call by Need – Pros & Cons

Pros:

- **Equational reasoning**: every function call with specific arguments behaves the same and is only called once.
- **Memoization**: complex computations are automatically memoized.

Cons:

- **Memoization**: if optimized poorly, every computation is memoized which can have a large memory footprint.
- **Unusual**: not widespread, can be confusing in some contexts.

Call by name vs. Call by need

- **Call by name**: Don't pass the evaluation of the expression, but a thunk computing it.
- **Call by need**: The thunk is evaluated once and only once. Add a "memo" field. Allows *lazy evaluation*.

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

Mode	foo[1]	foo[2]	t
Val			

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

Mode	foo[1]	foo[2]	t
Val	6		

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

Mode	foo[1]	foo[2]	t
Val	6	2	

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

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

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

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

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

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

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

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

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
end.
```

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

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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		

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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	

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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		

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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	

Exhibit the differences (Explicit lyrics…)

```
(* Global values *)
var t: integer;
var 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]);
    (* what value do foo[1], foo[2] and t now hold? *)
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

Call by Sharing – definition

- Implies that values in the language are based on *objects* rather than primitive types.
  - (i.e. all values are “boxed”)
  - This *differs from both call-by-value and call-by-reference.*
- Not in common use; terminology is inconsistent across different sources.

```
def f(data: list[int]):
    data.append(1)

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

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

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

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

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

Notes on Call-by-sharing

- Mutations of arguments done 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 done 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…

Semantics vs implementation

Exercise

Let's imagine a call by value on large POD, which the function does not modify.
It would be expensive to copy, so it should be passed by reference instead.

Should the compiler optimize the call? Or is it the responsibility of the developper?

Semantics vs implementation

Exercise

Let's imagine a call by value on large POD, which the function does not modify.
It would be expensive to copy, so it should be passed by reference instead.

Should the compiler optimize the call? Or is it the responsibility of the developper?

No right or wrong answer here!

- No compiler optization = explicit for the developper
  - Low-level focus on implementation
- Compiler optimization = implicit for the developper
  - High-level focus on program semantics