Typology of programming languages

Routines

Fundamentals of Subprograms

- 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

Subprograms

- At the origin, snippets copied and pasted from other sources
  - Impact on memory management;
  - Impact on separated compilation;
  - Modular programming: first level of interface/abstraction.
- First impact on Software Engineering: “top-down” conception, by refinements.
- Generalizations: modules and/or objects.

Procedures vs. Functions (1/3)

Procedure Collection of statements that define parameterized computations.
Subprograms with no return value.
Procedures have side effects

Function Structurally resemble procedures but are semantically modeled on mathematical functions.
Subprograms that return something.
(Pure) Functions do not have side effects

Procedures vs. Functions (2/3)

Ada, Pascal, ...have two reserved keywords **procedure** and **function**
BUT functions generally describe subprograms with return values, while procedures do not return values

Distinction sometimes blurred by the language:
(e.g., using `void` ALGOL, C, Tiger...).

Procedures vs. Functions (3/3)

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

Functions in **Pascal**

```
Procedure finish(name: String);
Begin
  WriteLn('Goodbye ', name);
End;
```

Procedures in **Pascal**

Nested subprograms (1/2)

Organize your programs in a cleaner fashion

It allows to share state easily in a controlled fashion, because the nested subprograms have access to the parameters, as well as any local variables, declared in the outer scope

Nested subprograms (2/2)

```
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;
```

Vocabulary

Formal Argument Arguments of a subprogram declaration.

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

Effective Argument Arguments of a call to a subprogram.

```
sum (40, 12)
```

Parameter Please reserve it for templates.

Hybridation: Procedure/Functions

Using functions with side effects is very dangerous. For instance:

```
foo = getc () +
      getc () *
      getc ();
```

is undefined ($\neq$ nondeterministic). *On purpose*!

Default arguments

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

Default Arguments in C++

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

Named Argument (Some sugar)

In Ada, named arguments and/or default values:

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

Some Ada function declaration

```
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);
```

Possible invocations

Named Arguments

Named parameters are availables in many languages: Perl, Python, C#, Fortran95, Go, Haskell, Lua, Ocaml, Lisp, Scala, Swift/ObjectiveC (**fixed order of named parameters**!), ...

- No need to remember the order of parameters
- No need to guess specific **default's** values
- More Flexible
- Clarity

Named Arguments

Can we simulate **named arguments** in C++ or Java?

Yes : **Named parameter idiom** uses a proxy object for passing the parameters.

Named Parameter Idiom 1/2

```
class foo_param{
private:
  int a = 0, b = 0;
  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 2/2

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

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

Works ... but require one specific class per function

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

Easter Egg

```
def f(list = []):
    list.append(1)
    print(list)
m = []
f()
f()
```

Summary