# Sub Programs

## Fundamentals of subprograms

### General Subprogram Characteristics.

* All subprograms discussed in this chapter, Except the coroutines described in Section have the following characteristics
  - → Each Subprogram has a Single entry Point
  - → The calling program unit is Suspended during the Execution of the called Subprogram, Which implies that there is only one subprogram in Execution at any given time.
  - → Control always returns to the caller When the Subprogram Execution terminates

Basic Definition- A subprogram describes the interface to and the actions of the subprogram abstraction. It is the Explicit request that the Called Subprogram be Executed. A Subprogram is said to be active if, after having been called, it has begun Execution but has not yet completed that Execution.

A Subprogram header, Which is the first part of the definition, Serves Several purposes. first it specifies that the following syntactic unit is a sub program definition of Some Particular Kind. The kind of sub program is often specified with a special word. Second the header provides a name for the Subprogram. Third, it may optionally Specify a list of Parameters

Consider the following header Examples:

* Subroutine Adder(Parameters)

This is the header of a fortran Subroutine Subprogram named Adder. In Ada, the header for this Subprogram would be

* procedure Adder(parameters)
* In Python, the header of a subprogram has the following form:

def adder(parameters);

Scanned by CamScanner

between calls to the subprogram.

* Extensive access to nonlocals can reduce reliability. Variables that are visible to the subprogram where access is desired often endup also being visible where access to them is not needed
* However, when a method does access nonlocal data, the reliability problems are the same as with non-method subprogram
* Also, in an Object-Oriented language, method access to class variables (those associated with the class, rather than an object) is related to the concept of nonlocal data and should be avoided whenever possible
* In this case, as well as the case of a C function accessing nonlocal data, the method can have the side Effect of changing something other than its parameters or local data.
* In some situations, it is convenient to be able to transmit computations, rather than data, as parameters to subprograms.
* The name of the subprogram that implements that computation may be used as a parameter.
* The parameters in the subprogram header are called formal parameters.
* Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called actual parameters
* The first actual parameter is bound to the first formal parameter and so forth. Such parameters is called positional parameters.
* It is easy for a programmer to make mistakes in the order of actual parameters in the list. One solution to this problem is to provide keyword parameters

Scanned by CamScanner

in Which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.

* The advantage of keyword Parameters is that they can appear in any order in the actual parameter list.
* Python functions can be called using this technique, as in

```
Sumer (length = my_length,
       list = my_array,
       Sum = my_Sum)
```

* Where the definition of Sumer has the formal parameters length, list, and Sum.
* In addition to keyword Parameters, Ada, fortran 95, and Python allow positional parameters. The two can be mixed in a call, as in

```
Sumer (my_length, Sum = my_Sum, list = my_array)
```

* This ~~Summary~~ restriction is necessary because a position may no longer be well defined after a keyword parameter has appeared.
* In python, Ruby, C++, fortran 95, Ada, and PHP formal Parameters can have default values. A default value is used if no actual parameter is passed to the formal parameter in the subprogram header.

Python function header:

```
def compute_pay (income, Exemptions = 1, tax_rate)
```

Scanned by CamScanner

## Procedures and functions

* There are two distinct categories of subprograms - procedures and functions - both of which can be viewed as approaches to extending the language.
* procedures are collections of statements that define parameterized computations. These computations are enacted by single call statements.
* In effect, procedures define new statements. for Example, because Ada does not have a sort statement, a user can build a procedure to sort arrays of data and use a call to that procedure in place of the unavailable sort statement.
* In Ada, procedures are called just that; in fortran, they are called subroutines.
* Procedures can produce result in the calling program unit by two methods. first, if there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them
* Second, if the subprogram has formal parameters that allow the transfer of data to the caller, those parameters can be changed
* functions structurally resemble procedures but are sematically modeled on mathematical functions. If a function is a faithful model, it produces no side effects, that is it modifies neither its parameters nor any variables defined outside the function.
* If a function is a faithful model, it produces no side effects that is, it modifies neither its parameters nor any variables defined outside the function
* functions are called by appearences of their names

Scanned by CamScanner

in Expressions, along with the ~~returned~~ required actual Parameters.

* Ex:- the value of the Expression f(x) is whatever value f Produces when called with the parameter x. for a function that does not produce side effects, the returned value is its only effect.

* A function can be written that returns the value of one of its parameters raised to the power of another Parameter. Its header in C++ could be

float power (float base, float Exp)

Which could be called with

result = 3.4 * power (10.0, x)

* The standard C++ library already includes a similar function named pow. Compare this with the Same operation in perl, in Which Exponentiation is a built-in operation:

result = 3.4 * 10.0 ** x.

Design Issues for Sub Programs

Subprograms are complex structures in programming languages. and it follows from this that a lengthy list of issues is involved in their design.

* One obvious issues is the choice of one or more Parameter-passing methods that will be used. The wide variety of approaches that have been used in various languages is a reflection of the diversity of opinion on the Subject.

* A closely related issue is whether the types of actual parameters will be type checked against the

Scanned by CamScanner

types of the corresponding formal parameters

* The nature of the local Environment of a Subprogram dictates to Some degree the nature of the Subprogram. The most important question here is whether local variables are Statiscally or dynamically allocated.
* Next, there is the question of the whether Subprogram definitions can be nested. Another issue is whether subprogram names can be passed as Parameters. If subprogram names can be passed as parameters and the correct referencing environment of a Subprogram that has been passed as a Parameter.
* finally, there are the questions of whether Subprograms can be over loaded or generic. An overloaded Subprogram is one that has the Same name as another Subprogram in the Same referencing Environment.
* A generic Sub program is one whose computations can be done ondata different types in different calls.

→ Are local variables statically or dynamically allocated

→ Can Subprogram definitions appear in other subprogram definitions?

→ What parameter-passing Method or Methods are used?

→ Are the types of the actual parameters checked against the types of formal Parameters?

→ If subprograms can be passed as parameters and Subprograms can be nested, What is the referencing Environment of a passed Subprogram?

→ Can subprograms be overloaded?

→ Can Subprograms be generic?

Scanned by CamScanner

# Local Referencing Environments

## Local variables:-

* Subprograms can define their own variables, thereby defining local referencing Environment.
* Variables that are defined inside subprograms are called local variables, because their scop is usually the body of the subprogram in which they are defined.
* In the terminology of chapter 5, local variables can be either static or stack-dynamic. If local variables are stack-dynamic, they are bound to storage when the subprogram begins Execution and are unbound from storage when that Execution terminates.
* There are several advantages of stack-dynamic local variables, the primary one being the flexibility they provide the subprogram.

It is essential that recursive subprograms have stack-dynamic local variables. Another advantage of stack-dynamic locals is that the storage for local variables in an active subprogram can be shared with the local variables in all inactive subprograms.

The main disadvantages of stack-dynamic local variables are the following. First there is the cost of the time required to allocate, initialize, and deallocate such variables for each call to the indirect, whereas accesses to static variables can be direct.

This indirectness is required because the place in the stack where a particular local variable will reside can be determined only during Execution.

It is sometimes convenient to be able to write history-sensitive subprograms. A common Example of a need

... sensitive subprogram is one whose task is

Scanned by CamScanner

to generate pseudorandom numbers.

* Each call to such a subprogram computes one pseudorandom numbers, using the last one it computed. It must, therefore, store the last one in a static local variable.
* The primary advantage of static local variables are over stack-dynamic local variables is that they are slightly more efficient - they require no run time overhead for allocation and deallocation.
* The greatest disadvantage of static local variables is their inability to support recursion. Also their storage cannot be shared with the local variables of other inactive subprograms
* In most contemporary languages, local variables in a subprogram are by default stack-dynamic. In C & C++ functions locals are stack-dynamic unless specifically declared to be static.
* for example in the following function, the variables sum is static and count is stack-dynamic.

```
int adder (int list[], int listlen) {
  static int sum = 0;
  int count;
  for (count = 0; count < listlen; count++)
  sum += list [count];
  return sum;
}
```

* Ada subprograms and the methods of C++, java and C# have only stack-dynamic local variables.

Nested Subprograms.

The idea of nesting subprograms originated with Algol 60. The motivation was to be able to create a hierarchy of both logic and scopes.

Scanned by CamScanner

* If a subprogram is needed only within another subprogram. Why not place it there are hide it from the rest of the program.
* Because Static Scoping is usually used in languages that allow subprograms to be nested. This also provides a highly structured way to grant access to nonlocal variables in enclosing subprograms.
* for a long time, the only languages that allowed nested Subprograms were those directly descending from Algol 60, Which were Algol 68, pascal, and Ada, Many other languages including all of the direct descendants of C, do not allow subprogram nesting.
* Recently, Some new languages again allow it. Among these are java Script, python, & Ruby.

## Parameter-Passing Methods:-

* Parameter-Passing methods are the ways in which parameters are transmitted to and from called Subprograms.
* we first focus on the different Semantics models of parameter-passing methods. Then we discuss the various implementation models invented by language designers for these Semantics models.
* Next we survey the design choices of the various imperative languages and discuss the actual methods used to implement the implementation models.

## Semantics Models of Parameter Passing

* formal parameters are characterized by one of three distinct Semantics models:

1. They can receive data from the corresponding

Scanned by CamScanner

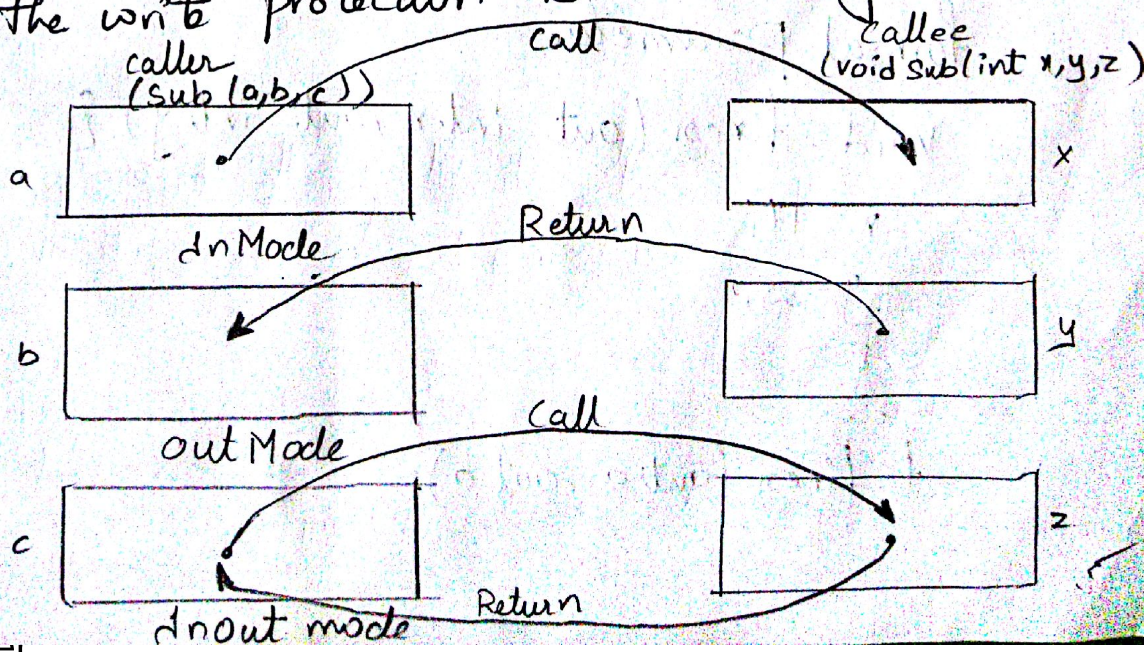
actual parameter
2. they can transmit data to the actual parameter,
3. they can do both. These three Semantics models are called in mode, out mode, & inout mode.

## Implementation Models of Parameter Passing:-

* A variety of Models have been developed by language designers to guide the implementation of the three basic parameter transmission modes.

### Pass-by-value:-

* When a parameter is passed by value, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram, thus implementing in-mode Semantics.
* pass-by-value is normally implemented by copy because accesses often are more efficient with this approach.
* It could be implemented by transmitting an access path to the value of the actual parameter in the caller but that would require that the value be in a write-protected cell (one that can only be read).
* Enforcing the write protection is not always a simple matter



Pass-by-Result

* Pass-by-result is an implementation model for out mode parameters. When a parameter is passed by result no value is transmitted to the subprogram.
* The corresponding formal parameter acts as a local variables, but just before control is transfered back to the caller.
* Its value is transmitted back to the caller's reference, the which obviously must be a variable
* The pass-by-result method has the advantages and disadvantages of pass by-value, plus some additional disadvantage.
* If values are returned by copy as they typically are pass-by-result also requires the extra storage and the copy operations that are required by pass-by-value.
* As with pass-by-value, the difficulty of implementing pass-by-result by transmitting an access path usually results in it being implemented by data copy.

for Ex:- Consider the following C# method, which specifies the pass-by-result method with the out specifier on its formal parameter.

```
void fixer (out int x, out int y) {
    x = 17;
    y = 35;
}
- - - -
f. Fixer (out a, out a);
```

* If at the end of the Execution of Fixer the formal parameter x is assigned to its corresponding actual parameter first, the value of the actual parameter a in the caller will be 35. If y is assigned first, the value of the actual parameter a in the caller will be 17.
* Yet another problem that can occur with pass-by-result is that the implementor may be able to choose between two different times to evaluate the addresses of actual parameter
* The time of the call or at the time of the return.

Ex:- Consider the following C# method & following code.

```
void DoIt(out int x, int index){
    x = 17;
    index = 42;
}
- - - -
sub = 21;
f.DoIt(list[sub], sub);
```

Pass-by-value Result:-

* It is an implementation model for inout-mode Parameters in which actual values are copied.
* It is in effect a combination of pass-by value and pass-by-result.
* The value of the actual parameter is used to initialize the corresponding formal parameter
* which then acts as a local variable. In fact, pass-by-value-result formal parameters must

have local storage associated with the called Subprogram.

* At Subprogram termination, the value of the formal parameter is transmitted back to the actual parameter

* Pass-by-value result is Sometimes called pass-by-copy, because the actual parameter is Said to the formal parameter at Subprogram Entry and then copied back at Subprogram termination.

* Pass-by-value result shares with pass-by-value and pass-by-result the disadvantages of requiring Multiple storage for parameters and time for copying values. The advantage of it is relative to reference.

Pass by Reference:-

* pass-by-reference is a Second implementation model for inout-mode parameters. Rather than copying data values back and forth. however, as in pass-by-value-result, the pass-by-reference method transmits an access path, usually just an address, to the called subprogram.

* This provides the access path to the cell storing the actual parameter. Thus the called Subprogram is allowed to access the actual parameter in the calling program unit.

* The advantage of pass by reference is that the Passing process itself is efficient, in terms of both time and Space

Scanned by CamScanner

* There are, however, several disadvantages to the pass-by-reference method. first access to the formal parameters will be slower than pass-by value parameters, because of the additional level of indirect addressing that is required.
* first, collisions can occur between actual parameters, Consider a C++ function that has two parameters that are to be passed by reference.

→ void fun (int & first, int & second)

If the call to fun happens to pass the same variable twice, as in fun (total, total) then first & list [i], list [j])

* still another way to get aliasing with pass-by-reference parameters is through collisions between formal parameters and nonlocal variables that are visible.

for example:-

```
int *global;
void main() {
---
sub(global);
---
}
void Sub (int * param) {
----
}
```

Inside sub, param, and global are aliases.

Scanned by CamScanner