RGPVNOTES.IN

Subject Name: **Principles of Programming Languages**

Subject Code: **IT-5002**

Semester: **5th**

Unit 3:

Subprograms Introduction:

- Subprograms are the most important concepts in programming language design.
- Two fundamental abstraction facilities can be included in programming language: process abstraction and data abstraction.

Blocks:

In computer programming, a block or code block is a lexical structure of source code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called a block-structured programming language. Blocks are fundamental to structured programming, where control structures are formed from blocks.

Fundamentals of Subprograms

Subprograms include the following characteristics:

- Each subprogram has a single entry point
- There is only one subprogram execution at any given time
- Control always returns to the caller when the subprogram execution terminates.

Basic Definitions

- A subprogram definition describes the interface to and the actions of the subprogram abstraction
- A subprogram call is the explicit request that the subprogram be executed.
- A subprogram is active if after having been called, it has begun execution but has not yet completed that execution.
- Two fundamental kinds of subprograms are procedures and functions.
- A subprogram header, which is the first line of the definition, serves several purposes. It specifies that the following syntactic unit is a subprogram definition. And it provides the name for the subprogram. And, it may optionally specify list of parameters.
- The parameter profile of a subprogram is the number, order and types of its formal parameters.
- The protocol of a subprogram is its parameter profile plus, if it is a function, its return types
- Subprograms declarations are common in C programs where they are called prototypes.

Parameters

- Subprograms usually describe computations.
- There are 2 ways that a subprogram can gain access to the data that is to process: through direct access to nonlocal variables or through parameter passing.
- Data passed through parameters are accessed through names that are local to the subprogram. Parameter passing is more flexible than direct access to nonlocal variables
- The parameters in the subprogram header are called formal parameters
- Subprograms 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 binding of actual parameters to formal parameters – is done by simple position: the first actual parameter is bound to the first formal parameter and so forth. Such parameters are called positional parameters.
- When lists are long, it is easy for the program writer to make mistakes in the order of parameters in the list – one solution to this problem is to provide keyword parameters, in which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.

Scope

The scope of a program variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.

Static Scope

Binding names to non-local variables is called static scoping. Static scoping is thus named because the scope of the variable can be statically determined, that is prior to execution

Evaluation of Static Scoping

- It is convenient to view the structure of program as a tree in which each node represents a procedure and thus a scope.
- A programmer could by mistake call a subprogram that should not have been callable, which would not be detected as an error by the compiler.

Dynamic Scope

Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. But the scope can be determined only at run time.

Design Issues of Subprograms And Operations

- Subprograms are complex structures in programming languages
- An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment.
- A generic subprogram is one whose computation can be done on data of different types with different calls

Local Referencing Environments

- Subprograms are generally allowed to define their own variables, thereby defining local referencing environments. Variables that are defined inside subprograms are called local variables because access to them is usually restricted to the subprogram in which they are defined.
- If local variables are stack dynamic, they are bound to storage when the subprogram begins execution and unbound from storage when that execution terminates. An advantage of this is flexibility.
- It is important that recursive subprograms have stack dynamic local variables.
- Another advantage is that some of the storage for local variables of all subprograms can be shared
- Main disadvantages of stack dynamic local variables are:
- Cost of time required to allocate, initialize and de-allocate for each activation
- Accesses of stack dynamic local variables must be indirect, where accesses to static can be direct
- Stack dynamic local variables, the subprograms cannot retain data values of local variables between calls.
- The primary advantage of static local variables is that they are very efficient because of no indirection

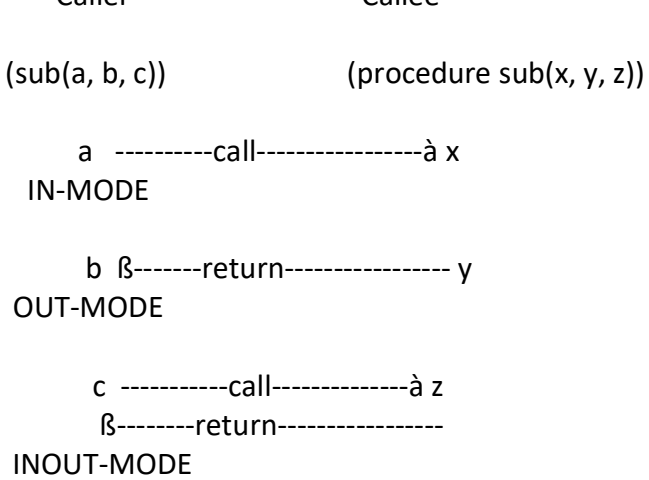
Parameter-Passing Methods

- Parameter-passing methods are the ways in which parameters are transmitted to and / or from called programs
- Formal parameters are characterized by one of three semantics models:
- They can receive data from the corresponding actual parameter
- They can transmit data to the actual parameter, OR
- They can do both.
- These three semantics models are called in mode, out mode and inout mode, respectively.
- There are 2 conceptual models of how data transfers take place in parameter transmission: either an actual value is physically moved (to the caller, to the callee, or both ways), or an access path is transmitted.

- Most commonly the access path is a simple pointer.

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 – this implements in-mode semantics.



- Pass-by-value is implemented by actual data transfer
- The main disadvantage of pass by value is that if physical moves are done, the additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram.

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
- One problem with the pass by result is that there can be an actual parameter collision such as the one created with the call **sub(p1, p2)**
- sub here assumes 2 formal parameters with different names, then whichever of the 2 is assigned to their corresponding actual parameter last becomes the value of p1.
- The order in which the actual parameters are assigned determines their value.
- Another problem with pass by result is that the implementer may be able to choose between 2 different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return.

Pass By Value Result

- Pass by value result is an implementation model for in-out mode parameters in which actual values are moved.
- It is a combination of pass by value and pass by result.

Pass By Reference

- Pass by reference is a second implementation of in-out mode parameters
- Rather than transmitting data values back and forth, 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.
- The advantage of pass by reference is efficiency in both time and space.
- The disadvantages are:
- Access to formal parameters is slow

- Inadvertent and erroneous changes may be made to the actual parameter
- Aliases can be created.

Pass By Name

- Pass by name is an in-out mode parameter transmission method that does not correspond to a single implementation model.
- When parameters are passed by name, the actual parameter is textually substituted for the corresponding formal parameter in all its occurrences in the subprogram.

Multidimensional Arrays As Parameters

- In some languages like C or C++, when a multidimensional array is passed as a parameter to a subprogram, the compiler must be able to build the mapping function for that array while seeing only the text of the subprogram. This is true because the subprograms can be compiled separately from the programs that call them.
- The problem with this method of passing matrices as parameters is that it does not allow the programmer t write a function that can accept matrices with different numbers of columns – a new function must be written for every matrix with a different number of columns. This disallows writing flexible functions that may be effectively reusable if the functions deal with multidimensional arrays.

Overloaded Subprograms

- An overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment
- Every version of an overloaded subprogram must have a unique protocol, that is, it must be different from the others in the number, order, or types of its parameters, or in its return types if it is a function
- C++, Java, and Ada include predefined overloaded subprograms

Generic Subprograms

- A generic or polymorphic subprogram takes parameters of different types on different activations.
- Overloaded subprograms provide a particular kind of polymorphism called ad hoc polymorphism.
- Parametric polymorphism is provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameter of the subprogram.
- Ada and C++ provide a kind of compile-time parametric polymorphism

Design Issues for Functions

The following two design issues are specific to function

- Are side effects allowed?
- What types of values can be returned?

Functional Side Effects

Because of the problems of side effects of functions that are called in expressions, as described in Chapter 4, parameters to functions should always be in mode. Some languages, in fact, require this; for example Ada functions can have only in-mode formal parameters. This effectively prevents a function from causing side effects through its parameters and aliasing of parameters and globals. In Pascal, function can have either pass-by-value or pass-by-reference parameters, thus allowing functions that cause side effects and aliasing. In C, the same is true.

Types of Returned Values

Most imperative programming languages restrict the types can be returned by their functions. FORTRAN 77, Pascal and Modula-2 functions allow only unstructured types to be returned. C allows any type to be returned by its functions except arrays and functions. Both of these can be handled by pointer type return values. C++ is

like also allows user-defined types, or classes, to be returned from its functions. The Ada language is alone among current imperative languages in that functions can return values of any type

Co-routines

Co-routines are computer-program components that generalize subroutines for non-preemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations. Co-routines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.

Overloaded Operators

Operator overloading is a specific case of polymorphism (part of the Object Oriented nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments. Operator overloading is usually only syntactic sugar. It can easily be emulated using function calls.

Arithmetic overloadable operators

- + (addition)
- - (subtraction)
- (multiplication)
- / (division)
- % (modulus)

Bitwise overloadable operators

- ^ (XOR)
- | (OR)
- & (AND)
- ~ (complement)
- << (shift left, insertion to stream)
- >> (shift right, extraction from stream)

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in