3. What are different methods for providing synchronization ? Explain in detail.
4. Explain the concept of semaphores in detail.
5. Explain the concept of monitors in detail.
6. Explain the concept of message passing in detail.
7. Write short notes :
   (a) Concurrency and parallelism
   (b) Cooperation and competition synchronization
   (c) Message passing models
   (d) Concurrency levels and subprogram levels concurrency.

CHAPTER

# 8 DATA CONTRO

## 8.1 INTRODUCTION

A program regardless of the language used specifies a set of operations applie on certain data in a certain sequence. The organization of data and operations into complex executable programs involves following two aspects :

(i) Sequence control, and

(ii) Data control

The term sequence control has already been discussed in detail in chapter. The various aspects of data control are discussed in this chapter.

## 8.2 DATA CONTROL

The term **data control** is defined as the control of the transmission of data for each operation of a program. The data control features of a programming language are concerned with following aspects :

1. The accessibility of data at different points during program execution.
2. The determination of how data may be provided for each operation, and how a result of one operation may be saved and retrieved for later use as an operand by a subsequent operation.

The term data control differs from sequence control in the following aspects as shown in table 8.1.

| Data Control | Sequence Control |
|---|---|
| 1. The control of the transmission of data among the subprograms of a program is termed as data control. | 1. The control of the order of the execution of operations, both primitive and user defined is termed as sequence control. |
| 2. Data control is ruled by the dynamic and static scope rules for an identifier. | 2. Sequence control is ruled by notations in expressions and hierarchy of operations. |

| Data Control | Sequence Control |
|---|---|
| 3. Data object can be made available through two methods.<br>(i) Direct Transmission and<br>(ii) Transmission through Reference. | 3. Sequence control structures may be either explicit or implicit. Implicit sequence control structures are those defined by the language and explicit are those that the programmer may optionally use. |
| 4. Data control structures may be categorized according to the referencing environment of data. | 4. Sequence control structures may be conveniently categorized in three groups.<br>(i) Structures used in expressions.<br>(ii) Structures used between statements, and<br>(iii) Structures used between subprograms. |
| 5. Data control is concerned with the binding of identifiers to particular data objects and subprogram. | 5. Sequence control is concerned with decoding the instructions and expressions into executable form. |

**Table 8.1 : Difference between Data control and Sequence control**

## 8.3 NAMES AND REFERENCING ENVIRONMENTS

There are two ways to make a data object available as an operand for an operation.

1. **Direction transmission,** used for data control with in expressions. In direct transmission, a data object computed at one point as the result of an cperation may be directly transmitted to another operation as an operand *e.g* in the computation of

$$a = 2 * b + c;$$

The result of multiplication is transmitted directly as an operand of the addition operation.

2. **Referencing through a named object,** used for most data controls outside of expressions. In this method, a data object may be given a name



when it is created, the name may then be used to designate it as an operand of an operation.

### 8.3.1 NAMES

A name is a string of characters used to identify some entity (or data object) in a program. The earliest programming languages used single-character names. Most of the languages today use multi-character names. Some languages such as C + +, do not specify a length limit on names, although implementers of those languages sometimes do. In a given program, there are following program elements that may be named.

1. Variable names
2. Formal parameters
3. Subprograms
4. Defined types
5. Defined constants
6. Labels
7. Exception names
8. Primitive operations
9. Literal constants

A name in a program may be

(a) Simple name, that designates an entire data structure *e.g.* an identifier name.

(b) Composite name, that designates a component of a data structure *e.g.* student [7] . firstname.

### 8.3.2 ASSOCIATIONS

An association is defined as the binding of identifiers (simple name) to particular data objects and subprograms. During program execution :

(i) At the beginning of main program, execution, associations are made (variables and subprograms).

(ii) During execution of main program, referencing operations (to be defined shortly) are invoked.

(iii) A new set of associations is created when a new subprogram is called.

(iv) During the execution of a subprogram, referencing operations are also invoked.

(v) The set of associations are destroyed when subprogram returns.

(vi) When the control returns back to main program, the main program continues execution as before.

### 8.3.2.1 VISIBILITY OF ASSOCIATIONS

Associations are visible if they are part of referencing environment, otherwise associations are hidden.

### 8.3.2.2 DYNAMIC SCOPE OF ASSOCIATIONS

The dynamic scope of an association includes the set of subprogram activations within which the association is visible.

## 8.3.3 REFERENCING ENVIRONMENTS

For each program or subprogram, a set of identifier associations available for use in referencing during execution is termed as **referencing environment** of the subprogram (or program). The referencing environment of a subprogram is :

(i) Set up when the subprogram activation is created, and is

(ii) Invariant during one activation.

The referencing environment of a subprogram may have several components.

1. **Local Referencing environment :** The local referencing environment of a subprogram is the set of identifier associations available for use in referencing, that may be determined without going outside the subprogram activation. The local referencing environment of a subprogram is the set of associations created on entry to a subprograms and includes.
   (a) Local variables, defined inside the subprograms *i.e.* access to local variables is restricted to subprograms in which they are defined.
   (b) Formal parameters
   (c) Subprograms defined only with in that subprograms.
2. **Non-Local Referencing Environment :** The non-local referencing environment of a subprogram is the set of identifier associations that may be used with in a subprogram but that are not created on entry to it. The non-local variables of a subprogram are those that are visible with in the subprogram but are not locally declared. It can be global or predefined. The non-local referencing environment includes.
   (a) Non-local variables
   (b) Subprograms enclosing given subprogram.
3. **Global referencing environment :** The global referencing environment of a sub-program is the set of identifier associations created at the start of the execution of the main program, available to be used in a subprogram.



The global referencing environment is the part of the non local referencing environment and includes :

(a) Global variables *i.e.* the variables that are visible in all program units.

(b) The subprograms declared in the main program.

4. **Predefined referencing environment :** The predefined referencing environment of a subprogram is set of identifier associations defined directly in the language definition that may be used without explicitly creating them. The example of a predefined environment in Pascal consist of constants such as MAXINT (the maximum integer value) and subprograms such as read, write and sqrt. Any of these predefined identifiers may be given a new association through an explicit program declaration, and thus the predefined association may become hidden for part of the program.

In order to illustrate the concept of local, non-local and global referencing environment consider the following program segment shown in figure 8.1.

```
void main ( )
    {
        float a, b, c ;
        .....
        void sub 1 (float a)
            {
                float d ;
                .....
                void sub 2 (float c)
                    {
                        float d ;
                        .....
                        c = c + b ;
                        .....
                    }
                .....
                sub 2 (b) ;
                .....
            }
        .....
        sub (a) ;
        .....
    }
```

**Fig 8.1 : A Sample Program in C to Illustrate Referencing Environment.**

From the figure 8.1, we observe the following referencing environments.

(i) **For Sub 2 :** (a) Local referencing environment consist of *c*, *d*.

(b) Non-local referencing environment consist of *a*, sub 2 in sub 1, *b*, sub 1 in main.

(*ii*) **For Sub 1 :** (*a*) Local referencing environment consist of *a*, *d*, sub 2.
(*b*) Non-local referencing environment consist of *b*, *c*, sub 1 in main.

(*iii*) **For main :** (*a*) Local referencing environment consist of *a*, *b*, *c*, sub 1.

### 8.3.3.1 REFERENCING ENVIRONMENT OF A STATEMENT

The **referencing environment** of a statement is the set of identifier associations (names) that are visible in the statement. The referencing environment of a statement in a static-scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible. In such a language, the referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.

In order to illustrate the referencing environment of statements consider the following Pascal program segments.

```
program main ;
    var a, b : integer ;
    .....
    procedure sub 1 ;
        var x, y : integer ;
        begin {sub 1}
        ..... ⟷ 1
        end ; {sub 1}
    procedure sub 2;
        var x : integer ;
        .....
        procedure sub 3 ;
        var x : integer ;
        begin {sub 3}
        ... ⟷ 2
        end ; {sub 3}
        begin {sub 2}
        ..... ⟷ 3
        end ; {sub 2}
    begin { main }
    ..... ⟷ 4
    end ; { main }
```

Fig. 8.2. A Sample Pascal Program Segment to Illustrate the Referencing Environments



From the figure 8.2, we observe the following referencing environments.

(*i*) At point 1, the referencing environment consist of *x* and *y* of sub 1, *a* and *b* of main.

(*ii*) At point 2, the referencing environment consist of *x* of sub 3 (*x* of sub 2 is hidden), *a* and *b* of main.

(*iii*) At point 3, the referencing environment consist of *x* of sub 2, *a* and *b* of main.

(*iv*) At point 4, the referencing environment consist of *a* and *b* of main,.

### 8.3.4 REFERENCING OPERATIONS AND REFERENCES

Given an identifier and a referencing environment, a **referencing operation** returns the associated data object or subprogram definition *i.e.* a referencing operation finds an appropriate association for an identifier in a given referencing environment. The signatures of a referencing operation is given as :

ref_op : id × ref_environment → data object or subprogram

#### 8.3.4.1 REFERENCES

A reference to an identifier involves finding the association in a particular referencing environment. A reference is said to be :

(*i*) **A local reference,** if the referencing identified operation finds the association in the local environment.

(*ii*) **A non-local reference,** if the referencing operation finds the association in the non-local environment.

(*iii*) **A global reference,** if the referencing operation finds the association in the global environment.

## 8.4 SCOPE

One of the most important factors having an effect on the understanding of variables is **scope**. The scope of a program variable is the range of the statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.

The scope rules of a language determine how a particular occurrence of a name is associated with a variable. In particular, scope rule determine how references to variables declared outside the currently executing subprogram or block are associated with their declaration and thus their attributes. A complete knowledge of

these rules for a language is therefore essential to our ability to write or read programs in that language.

Sometimes the scope and lifetime of a variable appear to be related. The lifetime of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell. In order to differentiate between scope and lifetime, consider a variable that is declared in a Pascal procedure that contains no subprogram calls. The scope of such a variable is from its declaration to the end reserved word of the procedure. The lifetime of that variable is the period of time beginning when the procedure is entered and ending when execution of the procedure reaches the end ; Although the scope and lifetime of the variable are clearly not the same because static scope is a textual, or spatial, concept where lifetime is a temporal concept, they atleast appear to be related in this case.

In this section we discuss two types of scope rules.

(*i*) Static scope, and

(*ii*) Dynamic scope

These scope rules are described as given below :

### 8.4.1 STATIC SCOPE

ALGOL 60 introduced the method of binding names to non-local variables, called **static scoping**. The **static scope** of a declaration is the part of the program text where a use of the identifier is a reference to that particular declaration of the identifier. The static scoping is thus named because the scope of a variable can be statically determined, *i.e.* prior to execution.

**Static Scope Rule :** A **static scope rule** is a rule determining the static scope of a declaration. The static scope rules relate references with declarations of names in the program text. The important of static scope rule lies in recording the information about a variable during translation.

In all common static-scoped languages except C, C++, Java and FORTRAN, subprogram can be nested inside other subprograms, which can create a hierarchy of scopes in a program. When a reference to a variable is found by a compiler for a static scoped language, the attributes of the variable are determined by finding the statement in which it is declared. In static scoped languages with nested subprograms, this process can be thought of in the following way.

Suppose a reference is made to a variable *a* in subprogram sub 1. The correct declaration is found by first searching the declarations of subprogram sub 1. If no declaration is found for the variable there, the search continues in the declarations



of the subprogram that declared subprogram sub 1, which is called its static parent. If a declaration of *a* is not found there, the search continues to the next larger enclosing unit and so on, until a declaration for *a* is found or the largest unit is declarations have been searched without success. In that case, an undeclared variable error has been detected. The static parent of subprogram sub 1, and its static parent, and so on up to and including the main program, are called the static ancestors of sub 1.

For example, consider the following Pascal procedure ;

```
procedure main ;
        var a : integer ;
        procedure sub 1 ;
        begin { sub 1 }
        ..... a .....
        end ; { sub 1 }
        procedure sub 2 ;
        var a : integer ;
        begin { sub 2 }
        .....
        end ; { sub 2 }
begin { main }
.....
end ; { main }
```

**Fig 8.3 :** A Skeleton Pascal Procedure.

Under static scoping, the reference to the variable *a* in sub 1 is to the a declared the procedure main, because the search for a begins in the procedure in which the ference occurs, sub 1, but no declaration for *a* is found there. The search thus ntinues in the static parent of sub 1, main whose the declaration of *a* is found.

### 8.1.1 IMPORTANCE OF STATIC SCOPE

The importance of the static scoping is clearly highlighted in the following ints.

1. The scope of a variable can be determined prior to execution (*i.e.* at compile time).
2. In the language that use static scoping, some variable declarations can be hidden from some subprograms.

3. The static type checking makes the program execution faster and more reliable.
4. The static scoping makes program easier to read and understand.
5. Static scope rules play an important part in the design and implementation of most programming languages *e.g.* Ada, C, FORTRAN, Pascal and COBOL.
6. The static scoping provides a method of accessing non-local variables.
7. The static scoping provides a different set of simplifications during translation that make execution of the program more efficient.

## 8.4.2 DYNAMIC SCOPE

The scope of variable in APL, SNOBOL4, and the early versions of LISP is dynamic. The **dynamic scoping** is based on the calling sequence of subprograms, not on their spatial relationship to each other. The **dynamic scope** of an association for an identifier is that set of subprogram activations in which the association is visible during execution. The dynamic scoping is thus named because the scope can be determined only at runt time.

**Dynamic Scope rule :** A **dynamic scope rule** defines the dynamic scope of each association in terms of the dynamic course of program execution. The dynamic scope rules relate references with associations for names during program execution. We can define it according to the dynamic chain of subprogram activations.

For example, consider again the Pascal procedure shown in figure 8.3.

Under dynamic scoping rules, the meaning of an identifier a determined in sub 1 is dynamic *i.e.* it cannot be determined at compile time. It may reference the variable from either declaration of *a*, depending on the calling sequence.

The correct meaning of a can be determined at run time by beginning the search with the local declarations. When the search of local declarations fails, the declarations of the dynamic parent, or calling procedure are searched. If a declaration for **a** is not found there, the search continues in that procedure's dynamic parent and so on, forming a **dynamic chain**, until a declaration for a is found. If none is found in any dynamic ancestor, it is a run-time error.

Consider the two different call sequences for sub 1 in the example above. First, main calls **sub 2**, which calls **sub 1**. In this case, the search proceeds from the local procedure, sub 1, to its caller, sub 2, where a declaration for a is found. So, a reference to a in **sub 1** in this case is to the a declared in sub 2. Next, sub 1 is called



directly from **main**. In this case, the dynamic parent of **sub 1** is **main**, and the reference is to the a declared in **main**.

### 8.4.2.1 ADVANTAGES AND DISADVANTAGES OF DYNAMIC SCOPE

The dynamic scope provides following advantages.

1. There is no need to determine the scope at compile time.
2. The correct attributes of non-local variables can be easily determined at run time.
3. A statement in a subprogram can refer to different non-local variables during different executions of the subprogram.
4. In some cases, the parameters passed from one subprogram to another are simply variables that are defined in the caller. None of these need to be passed in a dynamically scoped language, because they are implicitly visible in the called subprogram.

However, the dynamic scope has following problems

1. There is no way to protect local variables from being accessed during the execution of the subprograms.
2. The dynamic scoping results in less reliable programs than static scoping.
3. The dynamic scoping is unable to statically type check references to non-locals.
4. Dynamic scoping makes programs much more difficult to read.
5. The accesses to non-local variables in dynamic scoped languages take for longer than access to non-locals when static scoping is used.
6. The implementation of dynamic scope rule is costly.
7. It is not possible to statically determine the declaration for a variable referenced as a non-local.

## 8.5 BLOCK STRUCTURE

The concept of block structured languages was originated in ALGOL 60, one of the most important early languages. Because of their elegance and effect on implementation efficiency, they have been adopted in other languages like Pascal, PL/I, and Ada etc.

The **Block structured languages** have a characteristic program structure and associated set of static scope rules. The block structured program is organized as a set of blocks. A **block** is a section of code having its own local variables whose scope is minimized. In general a **block** consist of :

1. The beginning of a block consist of a set of declarations for names (or variables). Such variables are typically stack dynamic, so they have their storage allocated when the block is entered and deallocated when the block is exited. The declarations in a block define its local referencing environment.
2. The variable declarations are followed by a set of statements in which those names may be referenced.

The term **block-structure paradigm** is characterized by

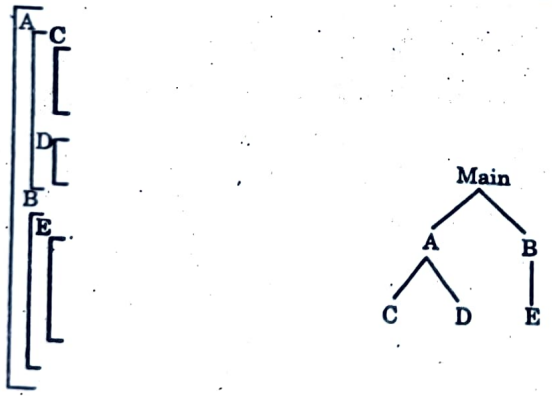
(*a*) Nested blocks

(*b*) Procedures

(*c*) Recursion

These are described as given below :

(*a*) **Nested blocks :** In a block structured languages, the blocks may be nested within other blocks, and may contain their own variables. The nesting of blocks is accomplished by allowing the definition of one block to entirely contain the definition of other blocks. At the outermost level, a program consists of a single block, defining the main program. With in this block are other blocks defining subprograms callable from the main program. With in these blocks may be other blocks defining subprograms callable from which the first-level subprograms, and so on. Figure 8.4 (*a*) illustrates typical layout of a block-structured program. The corresponding tree representation of a block-structured program is shown in figure 8.4 (*b*).




(*a*) (*b*)

**Fig. 8.4 (*a*) A Typical Layout of a Block Structured Program. (*b*) The Tree Representation of Block Structured Program.**

(*b*) **Procedures :** Once blocks have been implemented, procedures naturally as named blocks that can be called from other parts of a prog and that facilitate explicit information exchange between the calling the called blocks through parameters.

(*c*) **Recursion :** The implementation model for blocks is the stack. Only one block can be active at any one time, and its allocated storage occupies the top of the run-time stack. When a block terminates, its memory allocation will be popped, and memory for the calling block reactivated. The stack implementation supports recursion, as successive invocations of a recursive procedure can be pushed onto the run-time stack and popped in reverse order, passing values back down the stack.

### 8.5.1 STATIC-SCOPE RULES FOR BLOCK STRUCTURED LANGUAGES

The static scope rules associated with a block-structured program are as follows :

1. The declarations at the beginning of a block defines the local referencing environment for the block.
2. If for an identifier referenced with in the body of block no local declarations exists, then the reference is considered as a reference to a declaration with in the block that immediately encloses the first block and so on.
3. Finally, if no declaration is found, the declaration in the predefined language environment is used, else the reference is taken as an error.
4. If a block contain another block definition, then any local declarations with in the inner block or any blocks it contains are completely hidden from the outer block and cannot be referenced from it *i.e.* inner blocks encapsulates declarations so that they are hidden from outer blocks.
5. The block name becomes the part of the local referencing environment of the containing block.

The static scope rules are clearly illustrated in figure 8.5.

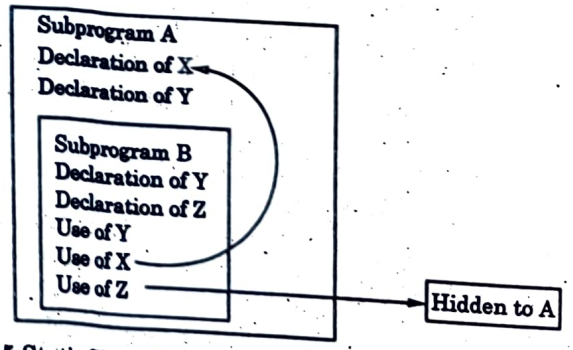



**Fig. 8.5 Static Slope Rules for a Block Structured Program.**

declaration is found for the

### 8.5.2 ADVANTAGES

The block structured language programs have following advantages.

1. A block introduces a new local referencing environment.
2. The program design is highly structured.
3. The variables declared in a block are stack dynamic.
4. Using the static scope rules, a declaration for the same identifier may occur in many different blocks ; but a declaration in an outer block is always hidden with in an inner block if the inner block gives a new declaration for the same identifier.
5. The static scope rules allow every reference to a name in a block to be associated with a unique declaration at compile time.
6. The compiler for the language may provide static-type checking.
7. Many simplifications of the run-time structure are based on the use of static scope rules.

## 8.6 LOCAL DATA AND LOCAL REFERENCING ENVIRONMENT

Subprograms are generally allowed to define their own data, called local data. The local data is defined inside the subprogram and access to the local data is usually restricted to the subprogram in which they are defined. Therefore the local data of subprogram defines the **local referencing environment**. The local **referencing environment** of a subprogram is the set of identifier associations available for use in referencing that may be determined without going outside the subprogram activation. The local data defined by the local referencing environment consist of

(*i*) **Local Variables**, declared in the head of the subprogram. These variables are called local because access to them is usually restricted to the subprogram in which they are defined. The local variables can be either static or stack dynamic. If the local variables are stack dynamic, they are bound to storage when the subprogram begins execution and unbound from storage when that execution terminates. The stack dynamic variables provides following advantages :

(*a*) Stack dynamic variables provides the subprogram flexibility.

(*b*) It is essential that recursive subprograms have stack-dynamic local variables.

(*c*) Some of the storage for local variables of all subprograms can be shared.



However the main disadvantages of stack-dynamic local variables are

(*a*) There is the cost of the time required to allocate, initialize and deallocates such variables for each activation.

(*b*) Access to stack-dynamic local variables must be indirect, whereas access to static variables can be direct.

(*c*) With stack-dynamic local variables, subprograms cannot be history sensitive *i.e.* they cannot retain data values of local variables between calls.

The static local variables have following advantages.

(*a*) The static local variables are very efficient.

(*b*) There is no indirection.

(*c*) Tney can be accessed much faster.

(*d*) They allow the subprograms to be history- sensitive.

The greatest disadvantage of static local variables is the inability to support recursion.

(*i*) Formal parameters (to be discussed in section 8.8)

(*ii*) The subprogram names defined locally with in that subprogram *i.e.* subprograms whose definitions are nested with in that subprogram.

### 8.6.1 SCOPE RULES FOR LOCAL REFERENCING ENVIRONMENT

For local environments, static and dynamic scope rules are easily made consistent :

**Static scope rule** : The static scope rule specifies that a reference to an identifier (or name) in the body of the subprogram is related to the local declaration for that identifier in the head of subprogram (assuming one exists). The static scope rules are implemented by means of a table of the local the entries are pairs, each containing an identifier and the associated data object. The name is only used so that later references to that variable will be able to determine where that variables will reside in memory during execution.

**Dynamic scope rule** : The dynamic scope rule specifies that a reference to an identifier (or name) during the execution of a subprogram refers to the association for that identifier in the current activation of the subprogram. Implementation of dynamic scope rule is done by following two methods.

(*i*) **Retention** : The identifier associations and the bound values are retained after execution. The implementation of the retention approach to local environment is straight forward using local environment tables. The table

is kept as the part of the code segment. Since the code segment is allocated storage statically and remains in existence throughout execution, any variables in the local environment part of the code segments are also retained. With this implementation of retention for the local environment, no special action is needed to retain the values of the data objects ; the values stored at the end of one call of subprogram will still be there when the next call begins. Also, no special action is needed to change from one local environment to another as one subprogram calls another. Since the code and local data for each subprogram are part of the same code segment, a transfer of control to the code for another subprogram automatically results in a transfer to the local environment for that subprogram as well. The languages like COBOL and many versions of FORTRAN use retention approach.

(*ii*) **Deletion :** In this approach, the identifier associations are deleted. The languages like C, Ada, LISP, APL and SNOBOL4 use the deletion approach. The implementation of the deletion approach to local approach to local environment is straight forward and using local environment tables. The table is kept as part of the activation record, destroyed after each execution. Assuming the activation record is created on a central stack on entry to subprogram, and deleted from the stack on exit, the deletion of the local environment follows automatically. Assuming each deleted local variable is declared at the start of the definition of subprogram, the compiler again can determine the number of variables and the size of each in the local environment table and may compute the offset of the start of each data object from the start of the activation record.

### 8.6.2 ADVANTAGES AND DISADVANTAGES

Both retention and deletion are used in a substantial number of important languages. These both approaches provides following advantages and disadvantages :

(*i*) The Retention approach allows the programmer to history sensitive subprograms.

(*ii*) The Retention approach consumes more storage space.

(*iii*) The Deletion approach is more natural strategy for recursive subprograms.

(*iv*) The Deletion approach provides a savings in storage space.

(*v*) The Deletion approach does not allow any local data to be carried over from one call to the next, so a variable that must be retained between calls must be declared as non-local to the subprogram.



## 8.7 SHARED DATA

The local data objects can be used only with in a single local referencing environment (*i.e.* with in a single subprogram). However, data objects are generally shared among several subprograms. The operations in each of subprograms may use the shared data. There are two ways that a subprogram can gain access to the shared data objects.

(*i*) Direct sharing through parameter transmission (to be discussed in section 8.8).

(*ii*) Through direct access to non-local referencing environments.

Although much of the required communication among subprograms can be accomplished through parameters, most languages provide direct access to non-local variables from external environments.

The **non-local variables** of a subprogram are those that are visible with in the subprogram but are not locally declared. **Global variables** are those that are visible in all program units. The global variables also form the part of non-local referencing environment.

There are four basic approaches to non-local environments that are used in programming languages :

(*i*) Explicit common environments and implicit non-local environments.

(*ii*) Dynamic scope

(*iii*) Static scope

(*iv*) Inheritance

These are four basic approaches are discussed as given below :

(*i*) **Explicit common environment and implicit non-local environments:** The explicit common environment method is most straight forward method of sharing data. A set of data objects to be shared among a set of subprograms is allocated storage in a separate named block. Each subprogram contains a declaration that explicitly names the shared block. The data objects with in the block are then visible with in the subprogram and may be referenced by name in the usual way. Such a shared block forms the common environment.

**Specification :** A common environment is similar to a local environment, however it is not a part of any single subprogram.

- Definition of variables
- Constants, and
- types

A common environment cannot contain

- Subprograms
- Formal parameters

**Implementation :** The common environment is implemented as a separate block of memory storage. Special keywords are used to specify the variables to be shared. The data objects may then be referred by names in the usual way. The implicit non-local environment has already been discussed in section 8.3.3.

**(ii) Dynamic Scope :** An alternative to the use of explicit common environments for shared data is the association of a non-local environment with each executing subprogram. A simpler, alternative is the use of the local environment for subprograms in the current dynamic chain.

**Specification :** The specification of dynamic scope has already been discussed in detail in section 8.4.2.

**Implementation :** An identifier association in the dynamic chain of subprogram calls is termed as **most-recent-association rule**. It is a referencing rule based on dynamic scope. The implementation of the most-recent-association rule for non-local references is straight forward, given the central-stack implementation for storing subprogram activation records. On entry to the subprogram, the activation record is created ; in return, the activation record is deleted.

**(iii) Static Scope :** In block-structured languages like Pascal and Ada, the handling of non-local references to shared data is done by the static scope rules used during translation.

**Specification :** The specification of static scope has already been discussed in detail in section 8.41 and 8.5.1.

**Implementation :** The static scope rules are straight forward to implement in the compiler. The static scope rules can be implemented by using the following two techniques :

**(a) Static Chain Implementation :** The static chain technique is the most direct technique for implementation of the correct referencing environment. A special entry in the beginning of the local environment table is called **static chain pointer.** This static chain pointer always contain the base address of another local table further down the stack. The table pointed to is the table representing the local environment of the statically enclosing block or subprogram in the original program. These static chain pointers forms the basis for a simple referencing scheme.



**Advantages :**

1. Fairly efficient implementation.
2. Allows straight forward entry and exit of subprograms.

**Disadvantages :** To determine an appropriate static chain pointer to install on entry to a subprogram.

**(b) Display Implementation :** The display implementation provides improvement upon the static chain implementation for non-local variable accessing. Instead of explicitly searching down the static chain for an identifier, we need only skip down the chain a fixed number of tables, and then use the base-address-plus-offset computation to pick out the appropriate entry in the table. We represent an identifier in the form of a pair (chain position, offset), during execution.

In this implementation, the current static chain is copied into a separate vector, termed the **display** on entry to each subprogram. The display is separate from the central stack and is often represented in a set of high-speed registers. At any given point during execution, the display contains the same sequence of pointers that occur in the static chain of the subprogram currently being executed.

**Advantages :**

1. Referencing using a display is particularly simple.
2. If the display is represented in high-speed registers during execution, then only one memory access per identifier reference is required.

**Disadvantages :** Subprogram entry and exit are more difficult.

**(iv) Inheritance :** Often, the information is passed among program components implicitly. We call the passing of such information **inheritance.** The inheritance is the receiving in one program component of properties or characteristics of another component according to a special relationship that exists between the two components. We have often used inheritance in programming language design. The details of further concept of inheritance are beyond the scope of this text.

## 8.8 PARAMETERS AND PARAMETER TRANSMISSION SCHEMES

Subprograms typically describe computations. There are two ways that a subprogram can gain access to the data that it is to process.

(i) Through direct access to non-local variables (discussed in section 8.7).

(ii) Through parameter passing.

The data objects passed through parameters are accessed through names that are local to the subprogram. Parameter passing is more flexible than direct access to non-local variables. In essence, a subprogram with parameter access to the data it is to process is a parameterized computation. It can perform its computation on whatever data it receives through its parameter.

### 8.8.1 PARAMETERS

An argument is the term used for a data object (or value) sent to a subprogram for processing. An argument can be obtained through :

(*i*) Parameters

(*ii*) Non-local references

The term **result** refers to the data object (or value) delivered by the subprogram returned through

(*i*) Parameters

(*ii*) Assignment to non-local variables

(*iii*) Explicit function values

Thus the terms **argument** and **result** apply to data spent to and returned from the subprogram through a variety of language mechanisms.

The parameters are associated with subprograms, specifying the form or pattern of data objects with which they will work. The parameters provide mechanisms for passing the information among subprograms. The different types of parameters are :

1. **Formal parameters :** A formal parameter is a particular kind of local data object with in a subprogram. The formal parameters are present in the subprogram header. It has a name and the declaration which specifies its attributes. They are sometimes called as dummy variables because they are not variables in the usual sense. In some case, they are bound to storage when the subprogram is called, and that binding is often through some other program variables. For example consider the following scenario as shown in figure 8.6.

```
void main ( )
        {
            int a, b ;
            .....
            .....
            sum (a, b) ;
            .....
            .....
        }
        sum (int x, int y)
            {
            .....
            .....
            }
```

**Fig 8.6** A C Program Fragment.



In figure 8.6 the subprogram sum defines two formal parameters $x$ and $y$ and declares the type of each.

2. **Actual parameters :** An actual parameter is a data object that is shared with the caller subprogram. The 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. Actual parameters must be distinguished from the formal parameters because the two can have different restrictions on their forms, and of course their uses are quite different. An actual parameter may be :

(*i*) A local data object belonging to the caller,

(*ii*) A formal parameter of the caller.

(*iii*) A non-local data object visible to the caller.

(*iv*) A result returned by a function invoked by the caller and immediately transmitted to the called subprogram.

For example in figure 8.6 the subprogram defines two actual parameters $a$ and $b$ and declares the type of each.

### 8.8.2 ESTABLISHING THE CORRESPONDENCE BETWEEN PARAMETERS

In nearly all programming languages, the correspondence between actual and formal parameters or the binding of actual parameters to formal parameters needs to be established when a subprogram is called with a list of actual parameters. There are following two methods of establishing this correspondence.

1. **Positional Correspondence :** The simplest possible way to establish a correspondence between actual and formal parameters is by position. The actual and formal parameters are paired based on their respective positions in the actual and formal parameter lists *e.g.* the first actual parameter is bound to the first formal parameter and so forth. This is good method for relatively short parameter lists. Such parameters are called **positional parameters.**

2. **Correspondence by explicit name :** When the parameter lists are long, however, it is easy for the program writer to make mistakes in the order of parameters in the list. One solution 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 explicitly. The advantage of the keyword parameters is that they can appear in any order in the actual parameter list. For example in Ada

sum ( $x \Rightarrow a$, $y \Rightarrow b$ ) ;

pairs the formal parameter $x$ with an actual parameter $a$ and a formal parameter $y$ with an actual parameter $b$. The main disadvantage to keyword parameters is that the user of the subprogram must know the names of the formal parameters.

Some programming languages like Ada and FORTRAN 90 allow the keyword parameters to be mixed with the positional parameters in a call.

### 8.8.3 PARAMETER PASSING METHODS

The term **parameter passing** refers to the matching of actual parameters with formal parameters when a subprogram call occurs. The parameter – passing methods are the ways in which the parameters are transmitted to and/or from called subprograms. The two approaches are often used :

(*i*) Semantic models of parameter passing

(*ii*) Implementation models of parameter passing

These approaches are discussed in detail as given below.

#### 8.8.3.1 SEMANTIC MODELS OF PARAMETER PASSING

To define the behaviour of the formal parameter, we use three distinct semantic models (called parameter modes).

(*i*) **IN mode :** An IN parameter mode lets you pass values to the subprogram being called *i.e.* formal parameters can receive data from the corresponding actual parameter. Inside the subprogram, an IN parameter acts like a constant. Therefore, it cannot be assigned a value. Figure 8.7 shows the IN mode of parameter passing.

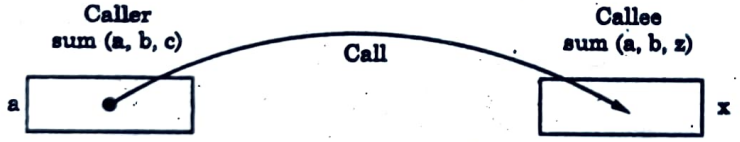



**Fig. 8.7 An IN Mode of Parameter Passing**

(*ii*) **OUT mode :** An OUT parameter mode lets you return values to the caller of a subprogram *i.e.* the formal parameters can transmit data to the actual parameter. Inside the subprogram, an OUT parameter acts like a variable. That means you can use an OUT formal parameter as if it were a local



variable. You can change its value or reference the value in any way. Figure 8.8 shows the OUT mode of parameter passing.


**Fig. 8.8 An OUT Mode of Parameter Passing.**

(*iii*) **IN OUT mode :** An IN OUT parameter mode lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, and IN OUT parameter acts like an initialized variable. Therefore, it can be assigned a value and its value can be assigned to another variable. he actual parameter that corresponds to an IN OUT formal parameter must be a variable ; it cannot be a constant or an expression.

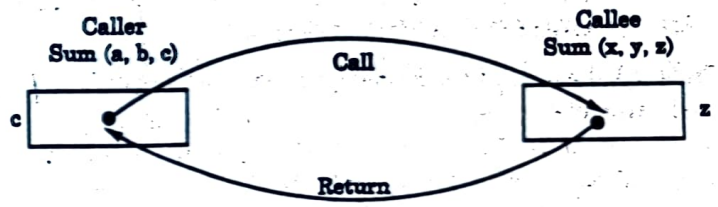



**Fig. 8.9 An IN OUT Mode of Parameter Passing.**

The table 8.2 summarizes the differences between three parameter modes described above.

| IN mode | OUT mode | IN OUT mode |
|---|---|---|
| 1. The default. | 1. Must be specified. | 1. Must be specified. |
| 2. Passes values to a subprogram. | 2. Returns values to caller. | 2. Passes initial values to a subprogram and returns updated values to the caller. |
| 3. Formal parameter acts like a constant. | 3. Formal parameters act like a variable. | 3. Formal parameter acts like an initialized variable. |

| IN mode | OUT mode | IN OUT mode |
|---|---|---|
| 4. Formal parameter cannot be assigned a value. | 4. Formal parameter must be assigned a value. | 4. Formal parameter should be assigned a value. |
| 5. Actual parameter can be a constant, initialized variable, literal, or expression. | 5. Actual parameter must be a variable. | 5. Actual parameter must be a variable. |
| 6. Actual parameter is passed by reference (a pointer to the value is passed in) | 6. Actual parameter is passed by value (a copy of the value is passed out) unless No copy is specified. | 6. Actual parameter is passed by value (a copy of the value is passed in and out) unless No copy is specified. |

**Table 8.2 :** Differences between IN mode, OUT mode and IN OUT mode

### 8.8.3.2 IMPLEMENTATION MODELS OF PARAMETER PASSING

Several models has been developed by language designers to guide the implementation of the three basic parameter transmission modes. The various methods ( or modes) for transmitting parameters are

(*i*) Call by value
(*ii*) Call by result
(*iii*) call by value-result
(*iv*) Call by reference
(*v*) Call by name
(*vi*) call by constant value

These methods of parameter passing are discussed in detail as given below.

**(*i*) Call 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 act as a local variable in the subprogram, thus implementing IN-mode semantics. In call by value, the actual parameter is copied in the location of the formal parameter.



The call by value is normally implemented by actual data transfer because accesses are usually more efficient with this method. In general a call-by-value implementation :

(*a*) Passes its *r*-value

(*b*) The formal parameter contains the value that is used.

The main disadvantage of the pass-by-value method if physical moves are done is that the additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and called subprogram. Also, the changes made in the formal parameter values during execution of the subprogram are lost when the subprogram terminates.

As an example consider the following program skeleton shown in figure 8.10

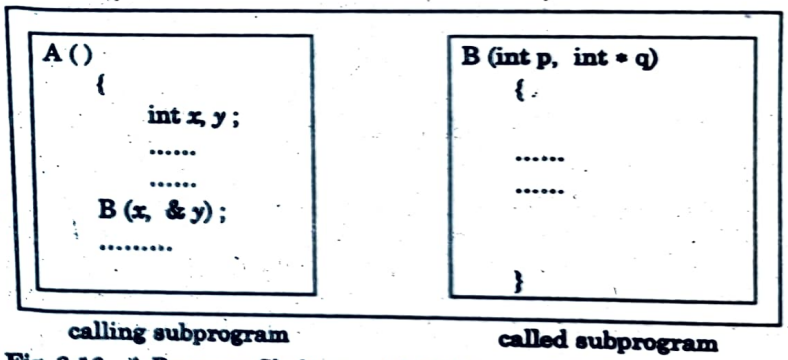



**Fig. 8.10** A Program Skeleton with Calling and Called Subprogram

In figure 8.10, the subprogram B has two formal parameters *p* and *q*. The parameter *p* is transmitted by value here.

**(*ii*) Call by result :** The call-by-result is an implementation model for OUT mode parameters. A parameter transmitted by result is used only to transmit a result back from a subprogram. The formal parameter is a local variable with no initial value when the subprogram terminates, the final value of formal parameter is assigned as the new value of the actual parameter. For example in figure 8.10, the final value of *p* and *q* may be assigned as the new values of the actual parameters *x* and *y*. However the call by result transmission method has following disadvantages.

(*a*) It requires extra storage.

(*b*) The difficulty of implementing call-by-result by transmitting an access path usually result in it being implemented by data transfer.

(c) The problem of ensuring that the initial value of the actual parameter is not used in the called subprogram.

(d) There can be an actual parameter collision.

(e) The portability problems that are difficult to diagnose.

(f) A problem that an implement may be able to choose between two different times to evaluate the addresses of the actual parameters.

**(iii) Call-by-value-result :** The call-by-value-result is an implementation model for IN OUT mode parameters in which actual values are moved. 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, call-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.

The call-by-value result has following disadvantages.

(a) Multiple storage for parameter

(b) Time for copying values

(c) The order in which actual parameters are assigned.

**(iv) Call-by-reference :** The call-by-reference is an implementation model for IN mode parameters. To transmit a data object as a call-by-reference parameter means that a pointer to the location of the data object is made available to the subprogram. The data object itself does not change position in memory. In the beginning of subprogram execution, the *l*-values of actual parameters are used to initialize local storage locations for the formal parameters. In general, a call-by-reference implementation.

(a) Passes its *l*-value.

(b) A call-by-reference parameter uses the *l*-value stored in the formal parameter to access the actual data object.

(c) There is no copying of values.

In figure 8.10, the parameter *q* is passed by reference.

**(v) Call-by-name :** The call-by-name is an IN OUT mode parameter transmission method that does not correspond to a single implementation model. When the parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. A call-by-name formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.



The call-by-name parameter passing method has advantages of the flexibility it affords the programmer. However, the call-by-name parameters passing method has following disadvantages.

(a) The process is slow relative to other parameter passing methods.

(b) High cost in terms of execution efficiency.

(c) Difficult to implement.

(d) Some simple operations are not possible with call-by-name parameters.

As an example of call-by-name parameter passing method in figure 8.10 the effect is to substitute the parameter *x* in place of *p* and *y* in place of *q*.

**(vi) Call-by-constant value :** If a parameter is transmitted by constant values then no change in the value of the formal parameter is allowed during program execution *i.e.* no assignment of a new value or other modification of the value of the parameter is allowed, and the formal parameter may not be transmitted to another subprogram except as a constant-value parameter. The formal parameter thus acts as a local constant during execution of the subprogram. The call-by-constant value has following advantages.

(a) Two implementations are possible.

(b) Protecting the calling program from changes in the actual parameter.

(c) The value of actual parameters cannot be modified by the subprogram.

### 8.8.3.3 IMPLEMENTATION OF PARAMETER TRANSMISSION

Each time when a subprogram is called a new activation of a subprogram receives a different set of parameters. Therefore, the storage for formal parameters is allocated as the part of activation record. Each formal parameter is a local data object in the subprogram. A formal parameter P can be treated as

(a) A Local data object of type T (*e.g.* in parameters transmitted by value-result, by value, and by result).

(b) A local data object of type pointer to T (*e.g* in parameters transmitted by reference).

The various actions associated with parameter transmission are split into two groups.

(a) Actions associated with the point of subprogram call in each calling subprogram.

(b) Actions associated with the entry and exit in the subprogram itself.

At each point of call, actual parameters are evaluated and the control is transferred to the called subprogram. After the control transfer, the prologue for the subprogram completes the actions associated with parameter transmission. Before the subprogram termination, the epilogue for the subprogram must copy the result values into the actual parameters transmitted by result or value result. The function values must also be copied into registers or into temporary storage provided by the calling program. The subprogram then terminates and activation record is lost, so all results must be copied out of the activation record before termination.

The compiler has two main tasks in the implementation of parameter transmission.

(a) It must generate the correct executable code for parameter transmission, for return of results and for each reference to a formal-parameter name.

(b) To perform the necessary static type checking to ensure that the type of each actual-parameter data object matches that declared for the corresponding formal parameter.

## KEY POINTS TO REMEMBER

- The term **data control** is defined as the control of the transmission of data for each operation of a program.
- The two ways to make a data object available as an operand for an operation are direct transmission and referencing through a named object.
- A **name** is a string of characters used to identify some entity (or data object) in a program.
- An **association** is defined as the binding of identifiers (simple names) to particular data objects and subprograms.
- The **referencing environment** of the subprogram (or program) is defined as the set of identifier associations available for use in referencing during execution.
- The basic component of a referencing environment includes local referencing environment, Non local referencing environment, global referencing environment and predefined referencing environment.
- The referencing environment of a statement is the set of identifier associations (names) that are visible in the statement.
- The scope of a program variable is the range of the statements in which the variable is visible.



- A variable is visible in a statement if it can be referenced in th statement.
- The two types of scope rules are **static and dynamic** scope rules.
- The **static scope** of a declaration is the part of the program text where a use of the identifier is a reference to that particular declaration of the identifier.
- The **dynamic scoping** is based on the calling sequence of subprograms, and not on their spatial relationship to each other.
- The **block structured program** is organized as a set of blocks.
- A **block** is a section of code having its own local variables whose scope is minimized.
- The nesting of blocks is accomplished by allowing the definition of one block to entirely contain the definition of other blocks.
- **Subprograms** are generally allowed to define their own data, called local data.
- The local data of subprogram defines the local referencing environment.
- A subprogram can gain access to the shared data objects through :
  - Direct sharing through parameter transmission.
  - Through direct access to non-local referencing environments.
- An **argument** is the term used for a data object (or value) sent to a subprogram for processing and **result** refers to the data object (or value) delivered by the subprogram returned through parameters, assignment to non-local variable and explicit function values.
- The two different types of parameters are formal and actual parameters.
- The correspondence between parameters can be obtained by position or by explicit name.
- The three different parameter model are IN, OUT and INOUT modes.
- The various methods for parameter transmission are :
  - Call by value
  - Call by result
  - Call by value-result
  - Call by reference
  - Call by name
  - Call by constant value.