UNIT-3
Subprograms and Blocks

Introduction

- Two fundamental abstraction facilities
  - Process abstraction
- Emphasized from early days
  - Data abstraction
- Emphasized in the1980s

3.1 Fundamentals of Subprograms – CO4

- 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

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram‘s parameter profile and, if it is a function, its return type
- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

- Positional
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
- Keyword
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  - Parameters can appear in any order

Formal Parameter Default Values

- In certain languages (**e.g.**, C++, Ada), formal parameters can have default values (if not actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated
- C# methods can accept a variable number of parameters as long as they are of the same type

Procedures and Functions

- There are two categories of subprograms
  - *Procedures* are collection of statements that define parameterized computations
  - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
- They are expected to produce no side effects
- In practice, program functions have side effects

3.2 Design Issues for Subprograms -CO3

- What parameter passing methods are provided?
- Are parameter types checked?
- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- Can subprograms be overloaded?
- Can subprogram be generic?

Scope and Lifetime

- The scope of a variable is the range of statements over which it is visible
- The nonlocal variables of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables
- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **static** variable in a C or C++ function

Static scope

  - Based on program text
  - To connect a name reference to a variable, you (or the compiler) must find the declaration
  - Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
  - Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent
- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
  - In Ada: **unit.name**
  - In C++: **class_name::name**
- Blocks
  - A method of creating static scopes inside program units--from ALGOL 60
  - Examples:

C and C++:
```
for (...)
{ int index;
  ...
}
```

Ada:
```
declare LCL : FLOAT;
begin
 ...
end
```

- Evaluation of Static Scoping
- Consider the example:

  Assume MAIN calls A and B

A calls C and D
B calls A and E

- Suppose the spec is changed so that D must now access some data in B
- Solutions:
  - Put D in B (but then C can no longer call it and D cannot access A's variables)
  - Move the data from B that D needs to MAIN (but then all procedures can access them)
- Same problem for procedure access
- Overall: static scoping often encourages many globals

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Scope Example

```
MAIN
    - declaration of x
        SUB1
            - declaration of x -
            ...
            call SUB2
            ...
        SUB2
            ...
            - reference to x -
            ...
    ...
call SUB1
...
```

- Static scoping
  - Reference to x is to MAIN's x
- Dynamic scoping
  - Reference to x is to SUB1's x
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability

Local Referencing Environments

- Def: The referencing environment of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms
- Local variables can be stack-dynamic (bound to storage)
  - Advantages
- Support for recursion
- Storage for locals is shared among some subprograms
  - Disadvantages
- Allocation/de-allocation, initialization time

- Indirect addressing
- Subprograms cannot be history sensitive
- Local variables can be static
  - More efficient (no indirection)
  - No run-time overhead

3.3 Parameter Passing Methods – CO3

- Ways in which parameters are transmitted to and/or from called subprograms
  - Pass-by-value
  - Pass-by-result
  - Pass-by-value-result
  - Pass-by-reference
  - Pass-by-name

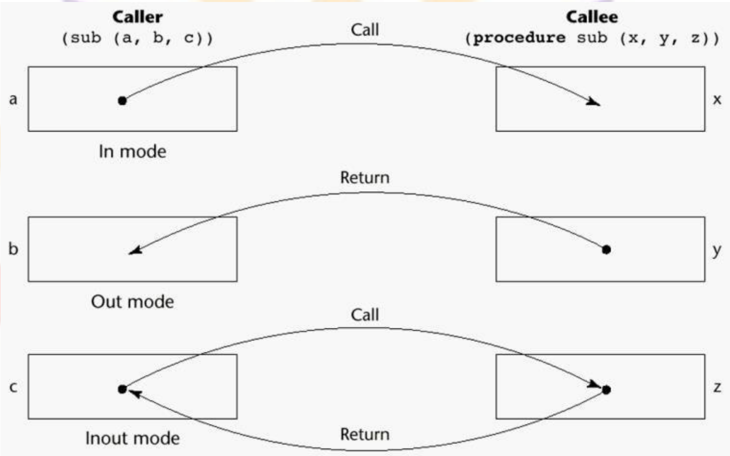

Figure 5.1 Models of Parameter Passing

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - When copies are used, additional storage is required
  - Storage and copy operations can be costly

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller‘s actual parameter when control is returned to the caller
  - Require extra storage location and copy operation
- Potential problem: sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1

Pass-by-Value-Result (Inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for un-wanted side effects
  - Un-wanted aliases (access broadened)

Pass-by-Name (Inout Mode)

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding

Implementing Parameter-Passing Methods

- In most language parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack
- A subtle but fatal error can occur with pass-by-reference and pass-by-value result: a formal parameter corresponding to a constant can mistakenly be changed

Parameter Passing Methods of Major Languages

- Fortran
  - Always used the inout semantics model
  - Before Fortran 77: pass-by-reference
  - Fortran 77 and later: scalar variables are often passed by value-result
- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All parameters are passed are passed by value
  - Object parameters are passed by reference
- Ada
  - Three semantics modes of parameter transmission: in, out, in out; in is the default mode
  - Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; in out parameters can be referenced and assigned
- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref
- PHP: very similar to C#
- Perl: all actual parameters are implicitly placed in a predefined array named @_

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal, FORTRAN 90, Java, and Ada: it is always required

- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking

Multidimensional Arrays as Parameters

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function

Multidimensional Arrays as Parameters: C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

Multidimensional Arrays as Parameters: Pascal and Ada

- Pascal
  - Not a problem; declared size is part of the array's type
- Ada
  - Constrained arrays - like Pascal
  - Unconstrained arrays - declared size is part of the object declaration

Multidimensional Arrays as Parameters: Fortran

- Formal parameter that are arrays have a declaration after the header
  - For single-dimension arrays, the subscript is irrelevant
  - For multi-dimensional arrays, the subscripts allow the storage-mapping function

Multidimensional Arrays as Parameters: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (length in Java, Length in C#) that is set to the length of the array when the array object is created

Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

3.4 Parameters Subprograms as parameters – CO3

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  - Are parameter types checked?
  - What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Parameter Type Checking

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed; parameters can be type checked
- FORTRAN 95 type checks

- Later versions of Pascal and
- Ada does not allow subprogram parameters; a similar alternative is provided via Ada's generic facility

Parameters that are Subprogram Names: Referencing Environment

- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
- *Deep binding*: The environment of the definition of the passed subprogram
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

3.5 Overloaded Subprograms – CO3

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

3.6 Generic Subprograms – CO3

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- Overloaded subprograms provide ad hoc polymorphism
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*

Examples of parametric polymorphism: C++

```
template <class Type>
Type max(Type first, Type second) {
return first > second ? first : second;
}
```

- The above template can be instantiated for any type for which operator > is defined

```
int max (int first, int second) {
return first > second? first : second;
}
```

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Ada allows any type
  - Java and C# do not have functions but methods can have any type

User-Defined Overloaded Operators

- Operators can be overloaded in Ada and C++
- An Ada example

```
Function "*"(A,B: in Vec_Type): return Integer is
Sum: Integer := 0;
begin
for Index in A'range loop
Sum := Sum + A(Index) * B(Index)
end loop
```