Chapter 9 Subprograms

We now explore the design of subprograms, including parameter-passing methods, local referencing environment, overloaded subprograms, generic subprograms, and the aliasing and problematic side effects.

9.1 Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
    - Discussed in this chapter
  - Data abstraction
    - Emphasized in the1980s
    - Discussed at length in Chapter 11

9.2 Fundamentals of Subprograms

- All subprograms have the following characteristics
  - 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

9.2.2 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
  - Python: def adder (parameters):
  - JavaScript: function
  - C: void adder (parameters)
- In Python, function def statements are executable.
  - Until a function's def has been execution, the function cannot be called

9.2.2 Basic Definitions

- The *parameter profile* 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
- Subprograms can have declarations as well as definitions

9.2.2 Basic Definitions (continued)

- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram

```
main(){
    int foo(int);
    …
    i=foo(j);
 }
 int foo(int x)
 {
   …
 }
```

9.2.3 Parameters

- There are two ways that a non-method subprogram can gain access to the data that it is to process
  - Direct access to nonlocal variables
  - Parameter passing
- 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

9.2.3 Parameters

- 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
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter's names

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)

- Variable numbers of parameters
  - C and C++: Ellipsis (…)
  - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**
  - In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.

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

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprogram be generic?
- If the language allows nested subprograms, are closures supported?

Local Referencing Environments

- Local variables can be stack-dynamic
  - 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
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Local Referencing Environments: Examples

- In most contemporary languages, locals are stack dynamic
- In C-based languages, locals are by default stack dynamic, but can be declared **`static`**
- The methods of C++, Java, Python, and C# only have stack dynamic locals
- In Lua, all implicitly declared variables are global; local variables are declared with **`local`** and are stack dynamic

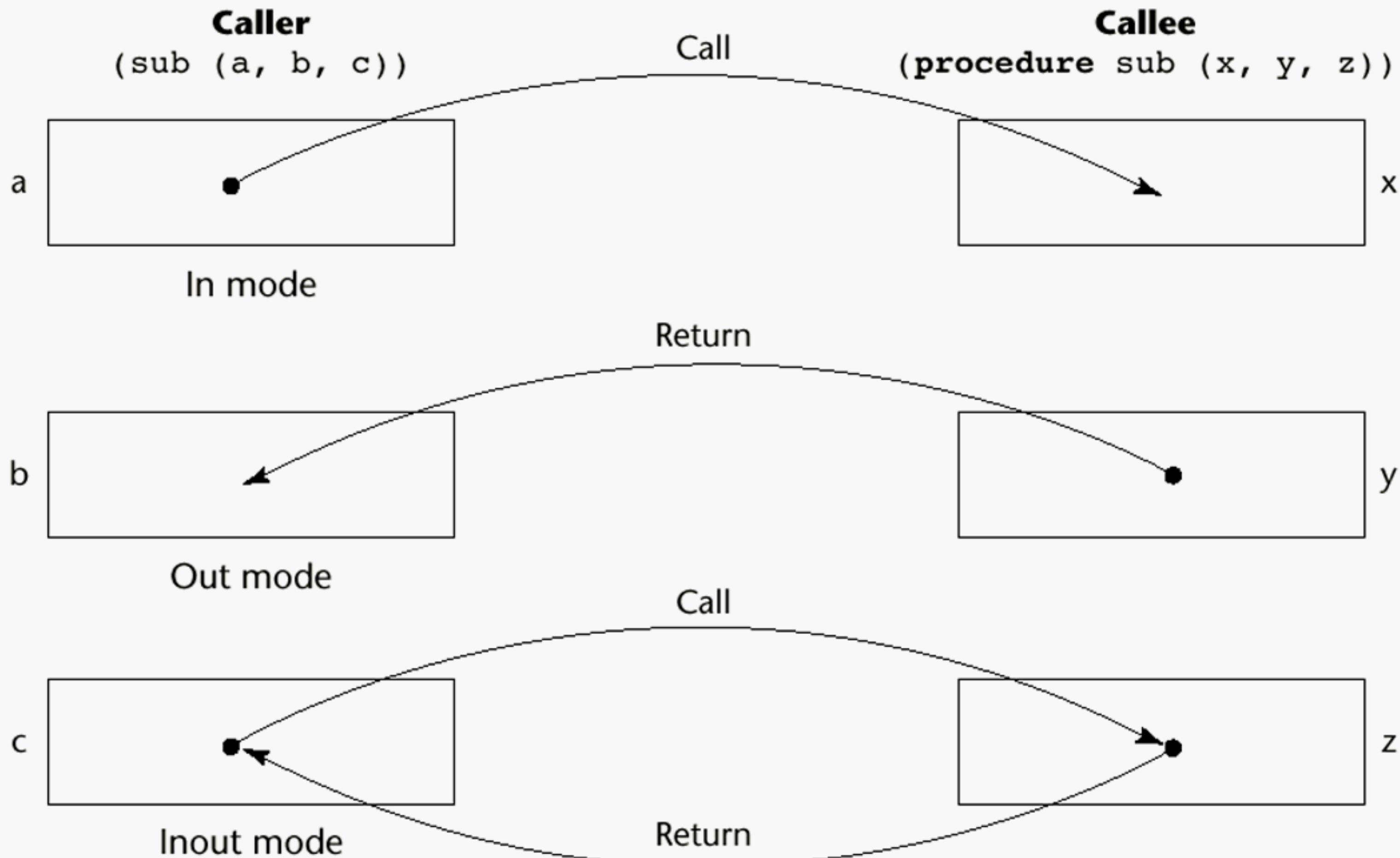
9.5 Parameter-Passing Methods

- Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms
- Issues:
  - Semantics models
  - Implementation models
  - Design choices

9.5.1 Semantics Models of Parameters Passing

- Three distinct semantics models
  - (1) Receive data from the corresponding actual parameter
    - In mode
  - (2) Transmit data to the actual parameter
    - Our mode
  - (3) Both
    - Inout mode

Models of Parameter Passing



9.5.2 Implementation Models of Parameter Passing

- Pass-by-value
  - In mode
  - Normally implemented by copy
  - Or, implemented by transmitting an access path to the value of the actual parameter in the caller
    - Write-protected is necessary

9.5.2 Implementation Models of Parameter Passing

- Pass-by-result
  - Out mode
  - No value is transmitted to the subprogram
  - Need extra copy
  - Problems
    - Actual parameter collision
    - How to choose between two different time to evaluate the addresses of the actual parameters

9.5.2 Implementation Models of Parameter Passing

- Pass-by-Value-Result
  - Inout mode
  - Sometimes called pass-by-copy
  - It shares with pass-by-value and pass-by-result the disadvantages of them.
  - The advantages of pass-by-value-result are relative to pass-by-reference.

9.5.2 Implementation Models of Parameter Passing

- Inout mode
- It transmits an access path (usually an address) to the called subprogram
- Advantage:
  - Efficient in terms of both time and space
- Disadvantage:
  - Slow
  - Unreliable
  - Alias may be created.

9.5.2 Implementation Models of Parameter Passing

- Pass-by-Name
  - Inout mode
  - The actually parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subrpograms
    - Consider the following program in pass-by-name

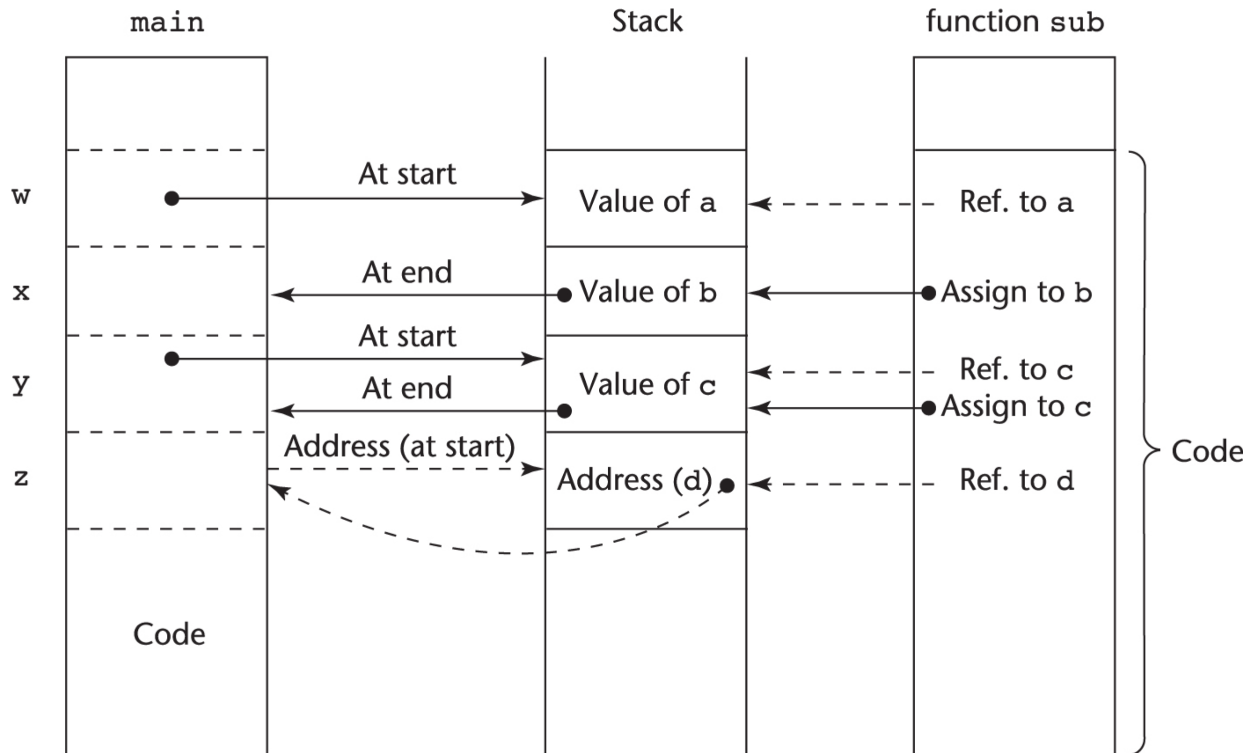
```
void swap(int a, int b){
    int temp;
    temp=a;
    a=b;
    b=temp;}
main(){
    int value=2, list[5]={1,3,5,7,9};

    swap(value, list[value]);
}
```

9.5.3 Implementing Parameter-Passing Methods

- In most languages 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
- See next slice.

Implementing Parameter-Passing Methods



Function header: `void sub(int a, int b, int c, int d)`

Function call in main: `sub(w, x, y, z)`

(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)

Parameter Passing Methods of Major Languages

- 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

Parameter Passing Methods of Major Languages (continued)

- Fortran 95+
  - Parameters can be declared to be in, out, or inout mode
- 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#, except that either the actual or the formal parameter can specify ref
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal
  - Check the text book for details.

9.5.5 Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: 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
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

9.5.6 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: 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