Subprograms

In Text: Chapter 9

Outline

- Definitions
- Design issues for subroutines
- Parameter passing modes and mechanisms
- Advanced subroutine issues

Subroutine

- A sequence of program instructions that perform a specific task, packaged as a unit
- The unit can be used in programs whenever the particular task should be performed

Subroutine

- Subroutines are the fundamental building blocks of programs
- They may be defined within programs, or separately in libraries that can be used by multiple programs
- In different programming languages, a subroutine may be called a **procedure**, a **routine**, a **method**, or a **subprogram**

Characteristics of Subroutines/Subprograms

- Each subroutine has **a single entry point**
- **The caller is suspended** during the execution of the callee subroutine
- **Control always returns to the caller** when callee subroutine's execution terminates

Parameters

- A subroutine may be written to expect one or more data values from the calling program
- The expected values are called **parameters** or **formal parameters**
- The actual values provided by the calling program are called **arguments** or **actual parameters**

Actual/Formal Parameter Correspondence

- Two options
  - Positional parameters
    - In nearly all programming languages, the binding is done by position
    - E.g., the first actual parameter is bound to the first formal parameter
  - Keyword parameters
    - Each formal parameter and the corresponding actual parameter are specified together
    - E.g., Sort (List => A, Length => N)

Keyword Parameters

- Advantages
  - Order is irrelevant
  - When a parameter list is long, developers won't make the mistake of wrongly ordered parameters
- Disadvantages
  - Users must know and specify the names of formal parameters

Default Parameter

- A parameter that has a default value provided to it
- If the user does not supply a value for this parameter, the default value will be used
- If the user does supply a value for the default parameter, the user-specified value is used

An Example in Ada

```
procedure sort (list   : List_Type;
                length : Integer := 100);
...
sort (list => A);
```

Design issues for subroutines

- What parameter passing methods are provided?
- Are parameter types checked?
- What is the **referencing environment** of a passed subroutine?
- Can subroutine definitions be nested?
- Can subroutines be overloaded?
- Are subroutines allowed to be generic?
- Is separate/independent compilation supported?

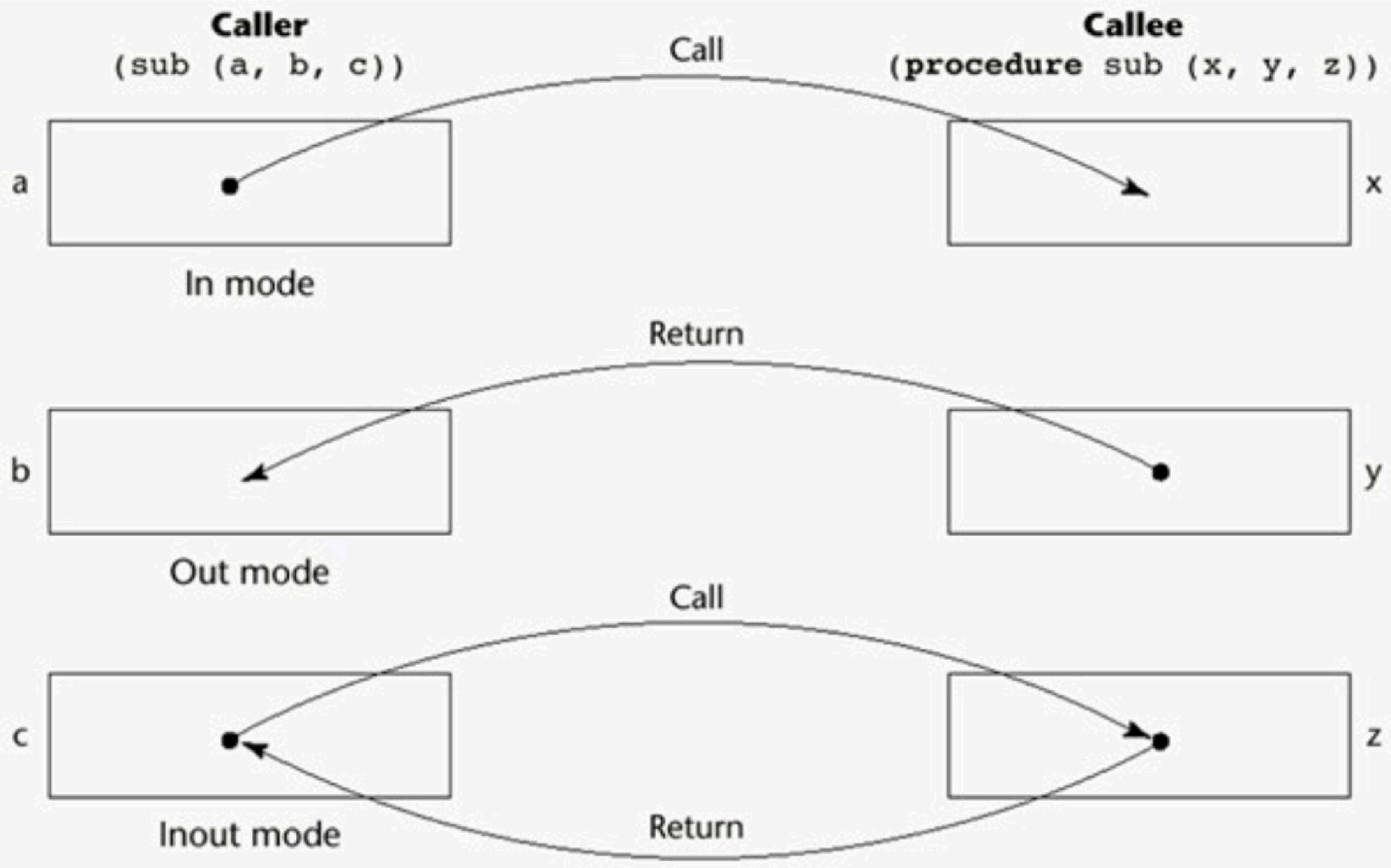
Parameter-Passing Methods

- Ways in which parameters are transmitted to and/or from callee subroutines
  - Semantic models
  - Implementation models

Semantic Models

- Formal parameters are characterized by one of three distinct semantic models
  - **In mode**: They can receive data from the corresponding actual parameters
  - **Out mode**: they can transmit data to the actual parameters
  - **Inout mode**: they can do both

Models of Parameter Passing



An Example

```
public int[] merge(int[] arr1, int[] arr2) {
    int[] arr = new int[arr1.length + arr2.length];
    for (int i = 0; i < arr2.length; i++) {
        arr[i] = arr1[i];
        arr2[i] = arr1[i] + arr2[i];
        arr[i + arr1.length] = arr2[i];
    }
    return arr;
}
```

Which parameter is in mode, out mode, or inout mode?

Implementation Models

- A variety of models have been developed by language designers to guide the implementation of the three basic parameter transmission modes
  - Pass-by-value
  - Pass-by-result
  - Pass-by-value-result
  - Pass-by-reference
  - Pass-by-name

Pass-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
- Implement in-mode semantics
- Implemented by copy

Pros and Cons

- Pros
  - Fast for scalars, in both linkage cost and access time
  - No side effects to the parameters
- Cons
  - Require extra storage for copying data
  - The storage and copy operations can be costly if the parameter is large, such as an array with many elements

Pass-by-Result

- No value is transmitted to a subroutine
- The corresponding formal parameter acts as a local variable, whose value is transmitted back to the caller's actual parameter
  - E.g.,

```
void Fixer(out int x, out int y) {
        x = 17;
        y = 35;
      }
```

- Implement out-mode parameters

Pros and Cons

- Pros
  - Same as pass-by-value
- Cons
  - The same cons of pass-by-value
  - Parameter collision
    - E.g., Fixer(x, x), what will happen?
    - If the assignment statements inside Fixer() can be reordered, what will happen?

Pass-by-Value-Result

- A combination of pass-by-value and pass-by-result, also called **pass-by-copy**
- Implement inout-mode parameters
- Two steps
  - The value of the actual parameter is used to initialize the corresponding formal parameter
  - The formal parameter acts as a local variable, and at subroutine termination, its value is transmitted back to the actual parameter

Pros and Cons

- Pros
  - Same as pass-by-reference, which is to be discussed next
- Cons
  - Same as pass-by-result

Pass-by-Reference

- A second implementation model for inout-mode parameters
- Rather than copying data values back and forth, it shares an access path, usually an address, with the caller
  - E.g., void fun(int &first, int &second)

Pros and Cons

- Pros
  - Passing process is efficient in terms of time and space
- Cons
  - Access to the formal parameters is slower than pass-by-value parameters due to indirect access via reference
  - Side effects to parameters
  - Aliases can be created

An Example: pass-by-value-result vs. pass-by-reference

```
program foo;
var x: int;
     procedure p(y: int);
     begin
          y := y + 1;
          y := y * x;
      end
begin
     x := 2;
     p(x);
     print(x);
end
```

	pass-by-value-result		**pass-by-reference**	
	x	**y**	**x**	**y**
(entry to p)	2	2	2	2
(after y:= y + 1)	2	3	3	3
	6	6	9	9
(at p’s return)				

Aliases can be created due to pass-by-reference

- Given void fun(int &first, int &second),
  - Actual parameter collisions
    - E.g., fun(total, total) makes first and second to be aliases
  - Array element collisions
    - E.g., fun(list[i], list[j]) can cause first and second to be aliases if i == j
  - Collisions between formals and globals
    - E.g., int* global;
      void main() { … sub(global); … }
      void sub(int* param) { … }
    - Inside sub, param and global are aliases

Pass-by-Name

- Implement an inout-mode parameter transition method
- The body of a function is interpreted at call time after textually substituting the actual parameters into the function body
- The evaluation method is similar to C preprocessor macros

An Example in Algol

```
procedure double(x);
   real x;
begin
   x := x * 2;
end;
```

Therefore, double(C[j]) is interpreted as
C[j] = C[j] * 2

Another Example

- Assume k is a global variable,

```
procedure sub2(x: int; y: int; z: int);
begin
    k := 1;
    y := x;
    k := 5;
    z := x;
end;
```

- How is the function call sub2(k+1, j, i) interpreted?

Disadvantages of Pass-by-Name

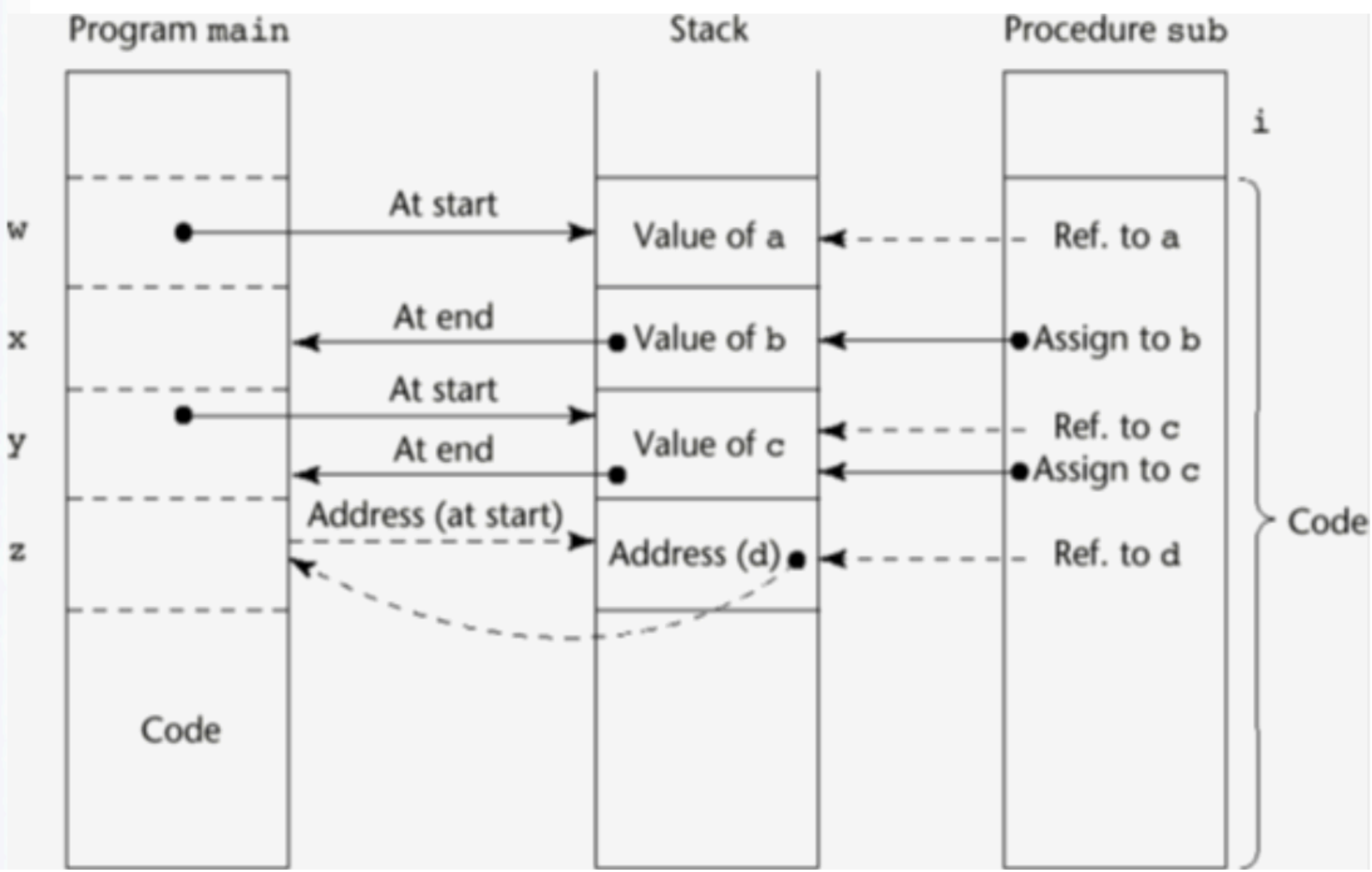
- Very inefficient references
- Too tricky; hard to read and understand

Implementing Parameter-Passing Methods

- Most languages use the runtime stack to pass parameters
  - Pass-by-value
    - Values are copied into stack locations
  - Pass-by-result
    - Values assigned to the actual parameters are placed in the stack
  - Pass-by-value-result
    - A combination of pass-by-value and pass-by-result
  - Pass-by-reference
    - Parameter addresses are put in the stack

An Example

- Function header: void sub (int a, int b, int c, int d)
  - a: pass by value
  - b: pass by result
  - c: pass by value result
  - d: pass by reference
- Function call: main() calls sub(w, x, y, z)


Program main
Stack
Procedure sub
i
w
x
y
z
At start
At end
At start
At end
Address (at start)
Value of a
Value of b
Value of c
Address (d)
Ref. to a
Assign to b
Ref. to c
Assign to c
Ref. to d
Code
Code

Design Considerations for Parameter Passing

- Efficiency
- Whether one-way or two-way data transfer is needed

One Software Engineering Principle

- Access by subroutine code to data outside the subroutine should be minimized
  - In-mode parameters are used whenever no data is returned to the caller
  - Out-mode parameters are used when no data is transferred to the callee but the subroutine must transmit data back to the caller
  - Inout-mode parameters are used only when data must move in both directions between the caller and callee

A practical consideration in conflict with the principle

- Pass-by-reference is the fastest way to pass structures of significant size