Lecture 5: Subprograms

Marriette Katarahweire and John Kizito

Outline

- Chapter 9, Concepts of Programming Languages by Sebesta
- Defining a subprogram
- Role of subprograms
- Subprogram Parameters
- Subprogram Design Issues
- Parameter-passing models and methods
- Polymorphism and Overloading

Subprogams

- Why use subprograms?
  - Give a name to a task.
  - We no longer care how the task is done.
- The Subprogam call is an expression
- Subprogams take arguments (in the formal parameters)
- Values are placed into variables (actual parameters/arguments)
- A value is (usually) returned

Subprogams

- Subprogram: is a method, function or procedure. It may have a name, parameters. Procedures have no return values
- Subprogram characteristics
  - Each subprogram has a single entry point
  - The calling program unit is suspended during the execution of the called subprogram, there is only one subprogram in execution at any given time
  - Control always returns to the caller when the subprogram execution terminates

Subprograms

- aid reuse, process abstraction
- save memory space, coding time
- increase readability of a program
- describe computation in a group of statements

Subprogram Parameters

- A subprogram can gain access to the data that it is to process through:
  - direct access to nonlocal variables (declared elsewhere but visible in the subprogram)
  - parameter passing.
- Data passed through parameters are accessed through names that are local to the subprogram
- The parameters in the subprogram header are called formal parameters
- 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.

Subprogram Parameters

- In nearly all programming languages, the correspondence between actual and formal parameters—or the binding of actual parameters to formal parameters—is done by position. Such parameters are called positional parameters
- When parameter lists are long, it is easy for a programmer to make mistakes in the order of actual parameters in the list. One solution to this problem is to provide keyword parameters
- Keyword parameters: the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call.
- The advantage of keyword parameters is that they can appear in any order in the actual parameter list

Keyword Parameters

Ada function

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

- formal parameters: *length*, *list*, *sum*
- What is the disadvantage of having keyword parameters?
- Find out about programming languages:
  - whose subprograms' formal parameters accept default values.
  - which allow a variable number of parameters.

Design Issues for Subprograms

- Are local variables statically or dynamically allocated?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter-passing method or methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprograms be generic?
- If the language allows nested subprograms, are closures supported?

Design Issues for Subprograms

- 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 in different calls.
- A closure is a nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program

Parameter-Passing Methods

- One of the design issues of subprograms
- Parameter passing methods: the ways in which parameters are transmitted to and/or from called subprograms
- The call by reference method allows the procedure to change the value of the parameter, whereas call by value method guarantees that the procedure will not change the value of the parameter.

Semantics Models of Parameter passing

Formal parameters are characterized by one of three distinct semantics models:

- they can receive data from the corresponding actual parameter: **in mode**
- they can transmit data to the actual parameter: **out mode**
- they can do both: **inout mode**

Example

consider a subprogram that takes two arrays of int values as parameters—list1 and list2. The subprogram must add list1 to list2 and return the result as a revised version of list2. Furthermore, the subprogram must create a new array from the two given arrays and return it.

- list1 should be in mode, because it is not to be changed by the subprogram
- list2 must be inout mode, because the subprogram needs the given value of the array and must return its new value
- the third array should be out mode, because there is no initial value for this array and its computed value must be returned to the caller

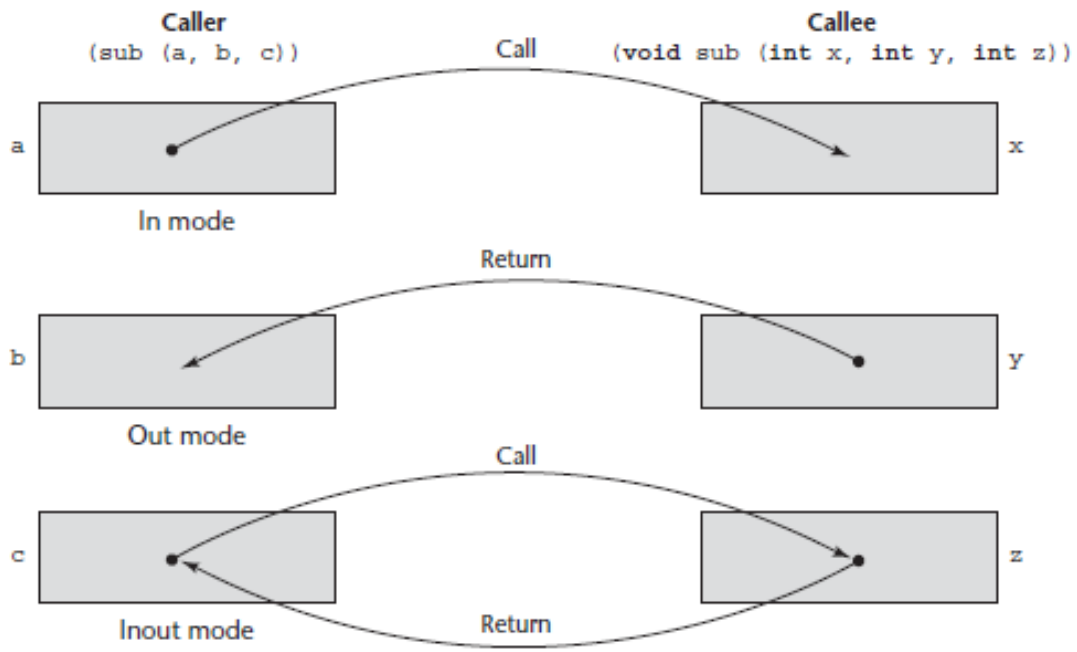
Conceptual Model of Data transfer

two conceptual models of how data transfers take place in parameter transmission:

- an actual value is copied (to the caller, to the called, or both ways)
- an access path is transmitted. Most commonly, the access path is a simple pointer or reference

Illustration

the figure illustrates the three semantics models of parameter passing when values are copied



Implementation Models of Parameter Passing

- Pass-by-value
- Pass-by-result
- Pass-by-value-result
- Pass-by-reference
- Pass-by-name

Pass by value

- implements in-mode semantics
- the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram
- Pass-by-value is normally implemented by copy. Alternative is by transmitting an access path to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell (one that can only be read).
- Disadvantage: if copies are used, additional storage is required for the formal parameter. The storage and the copy operations can be costly if the parameter is large
- Advantage: is fast for scalars in both linkage cost and access time

Pass by result

- an implementation model for out-mode parameters
- no value is transmitted to the subprogram.
- the corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the caller's actual parameter, which must be a variable.

Pass-by-value-result

- an implementation model for inout-mode parameters in which actual values are copied
- 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
- 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

Pass-by-reference

- is a second implementation model for inout-mode parameters.
- rather than copying data values back and forth,like in pass-by-value-result, the pass-by-reference method transmits an access path, usually just an address, to the called subprogram
- provides the access path to the cell storing the actual parameter.
- the called subprogram is allowed to access the actual parameter in the calling program unit.
- the actual parameter is shared with the called subprogram.

Pass by Name

- an inout-mode parameter transmission method that does not correspond to a single implementation model
- the actual parameter is textually substituted for the corresponding formal parameter in all its occurrences in the subprograms
- different from those discussed so far; formal parameters are bound to actual values or addresses at the time of the subprogram call
- a 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

Operator Overloading

- An overloaded operator is one that has multiple meanings. The types of its operands determine the meaning of a particular instance of an overloaded operator.
- For example, if the * operator has two floating-point operands in a Java program, it specifies floating-point multiplication.
- But if the same operator has two integer operands, it specifies integer multiplication.

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 if it is a function.
- The meaning of a call to an overloaded subprogram is determined by the actual parameter list.
- Users are also allowed to write multiple versions of subprograms with the same name in Ada, Java, C++, and C#

Signatures

- In any programming language, a signature is what distinguishes one function or method from another
- In C, every function has to have a different name
- In Java, two methods have to differ in their names or in the number or types of their parameters
  foo(int i) and foo(int i, int j) are different
  foo(int i) and foo(int k) are the same
  foo(int i, double d) and foo(double d, int i) are different
- In C++, the signature also includes the return type but not in Java!

Polymorphism

- Polymorphism means many (poly) shapes (morph)
- In Java, polymorphism refers to the fact that you can have multiple methods with the same name in the same class
- There are two kinds of polymorphism:
  - Overloading: Two or more methods with different signatures
  - Overriding: Replacing an inherited method with another having the same signature

Overloading

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}
```

Overriding

```
class A {
   int i, j;
   A(int a, int b) {i = a; j = b; }
   void show() {
     System.out.print("i and j:  "+ i +" "+ j);
   }
}

class B extends A {
   int k;
   // ...
   void show() {
     super.show(); // call superclass version
     System.out.print(" k:  "+ k);
   }
}
```

Generic Subprograms

- A programmer should not need to write four different sort subprograms to sort four arrays that differ only in element type.
- A generic subprogram is one whose computation can be done on data of different types in different calls.
- 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 parameters of the subprogram.

Exercise

Review Questions: 8, 10, 15, 16
Problem Set 5, 7, 11, 12, 15