Subprograms

Lecture 16

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Fundamentals of Subprograms: Procedures and Functions

- There are two distinct categories of subprograms—***procedures*** and ***functions***—both of which can be viewed as approaches to extending the language.
- All subprograms are collections of statements that define parameterized computations.
- ***Functions*** return values and ***procedures*** do not.
- In most languages that do not include procedures as a separate form of subprogram, functions can be defined not to return values and they can be used as procedures.

Fundamentals of Subprograms: Procedures and Functions

- Procedures can produce results in the calling program unit by two methods:
  - (1) If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them;
  - (2) If the procedure has formal parameters that allow the transfer of data to the caller, those parameters can be changed.

Fundamentals of Subprograms: Procedures and Functions

- Functions are called by appearances of their names in expressions, along with the required actual parameters.
- The value produced by a function's execution is returned to the calling code, effectively replacing the call itself.
- For example, the value of the expression f(x) is whatever value f produces when called with the parameter x.
- For a function that does not produce ***side effects***, the returned value is its only effect.

Fundamentals of Subprograms: Procedures and Functions

- Functions define new user-defined "operators".
- For example,
  - if a language does not have an exponentiation operator, a function can be written that returns the value of one of its parameters raised to the power of another parameter.
- Its header in C++ could be

```
float power(float base, float exp)
```

which could be called with

```
result = 3.4 * power(10.0, x)
```

Local Referencing Environments: Local Variables

- In most contemporary languages, local variables in a subprogram are by default ***stack dynamic***.

```
int adder(int list[], int listlen) {
    int sum = 0;
    int count;
    for (count = 0; count < listlen; count++)
        sum += list [count];
    return sum;
}
```

Local Referencing Environments: Local Variables

- In C and C++ functions, locals are stack dynamic unless specifically declared to be ***static***.

```
int adder(int list[], int listlen) {
    static int sum = 0;
    int count;
    for (count = 0; count < listlen; count++)
        sum += list [count];
    return sum;
}
```

Local Referencing Environments: Local Variables

- Subprograms can define their own variables, thereby defining local referencing environments.

- Variables that are defined inside subprograms are called ***local variables***, because their scope is usually the body of the subprogram in which they are defined.

Parameter-Passing Methods: Semantics Models of Parameter Passing

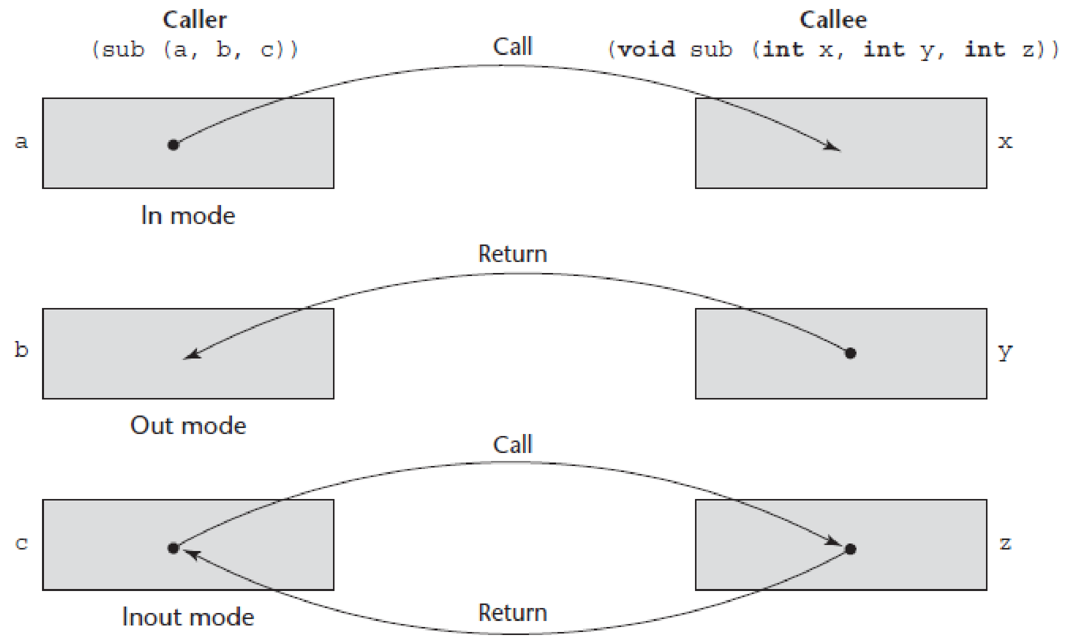
- Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms.
- Formal parameters are characterized by one of three distinct semantics models:
  - (1) They can receive data from the corresponding actual parameter; (***in mode***)
  - (2) They can transmit data to the actual parameter; (***out mode***)
  - (3) They can do both. (***inout mode***)

Parameter-Passing Methods: Semantics Models of Parameter Passing

- For 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.
    - For this subprogram, 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.

Parameter-Passing Methods: Implementation Models of Parameter Passing

- A variety of models have been developed by language designers to guide the implementation of the three basic parameter transmission modes.
- The three semantics models of parameter passing when physical moves are used



Parameter-Passing Methods: Semantics Models of Parameter Passing

- There are 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.

Parameter-Passing Methods: Pass-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 acts as a local variable in the subprogram, thus implementing in-mode semantics.

- ***Pass-by-value*** is normally implemented by ***copy***, because accesses often are more efficient with this approach.

Parameter-Passing Methods: Pass-by-Value

- Example:

```
void swap(int a, int b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
}

int main() {
  int num1 = 10, num2 = 20;
  printf("Before swapping num1 = %d num2 = %d\n", num1, num2);
  swap(num1, num2);
  printf("After swapping num1 = %d num2 = %d\n", num1, num2);
  return 0;
}
```

Parameter-Passing Methods: Pass-by-Reference

- ***Pass-by-reference*** is a second implementation model for inout-mode parameters.

- ***Pass-by-reference*** method transmits an ***access path***, usually just an ***address***, to the called subprogram.

- This provides the ***access path*** to the cell storing the ***actual parameter***.
  - Thus, the called subprogram is allowed to *access the actual parameter* in the calling program unit.

- In effect, the actual parameter is *shared* with the called subprogram.

Parameter-Passing Methods: Pass-by-Reference

- Example:

```
void swap(int *a, int *b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
  int num1 = 10, num2 = 20;
  printf("Before swapping num1 = %d num2 = %d\n", num1, num2);
  swap(&num1, &num2);
  printf("After swapping num1 = %d num2 = %d\n", num1, num2);
  return 0;
}
```