Chapter 9

Subprograms
Chapter 9 Topics

• Introduction
• Fundamentals of Subprograms
• Design Issues for Subprograms
• Local Referencing Environments
• Parameter-Passing Methods
• Parameters That Are Subprogram Names
• Overloaded Subprograms
• Generic Subprograms
• Coroutines
Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
  - Data abstraction
    - Emphasized in the 1980s
Fundamentals of Subprograms

• Except to coroutines (described in 9.11), all subprograms discussed in Ch9 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
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
- 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**
  - It provides protocol, but not the body, of the subprogram
Basic Definitions (continued)

- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram.  
  - It is called dummy because they are not bound to storage till the subprogram is called.

- An *actual parameter* represents a value or address used in the subprogram call statement.
Actual/Formal Parameter Correspondence

• **Positional** (most language use this)
  – 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 (but easy to make mistakes if the list are long)

• **Keyword**
  – The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.
  – E.g., Ada: `sum( length=> my_length, list=>my_array, sum=> my_sum);`
  – Parameters can appear in any order

• **Mixed**
  – Ada: `sum( my_length, sum => my_sum, list =>my_array);`
  – Positional parameters must appear before keyword parameters.
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
  – In most languages, the number of actual parameters must match the number of formal parameters (if no default values).
    • Not required for C, C++, Perl, and Javascript
    • This allows a variable number of parameters. (e.g., printf)
• C# methods can accept a variable number of parameters as long as they are of the same type
  e.g., public void DisplayList(params int[] list) {
    foreach(int next in list) {
      console.WriteLine(“Next value{0}”, next);
    }
  }
  – It can be called with both of following
    myObject.DisplayList( myList ); //assume mylist={1,2,3}
    MyObject.DisplayList(1, 2, 3, 4, 5);
Design Issues for Subprograms

• 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?
Local Referencing Environments

• 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
  – Cannot support recursion
Models of Parameter Passing

**Caller**
(sub (a, b, c))

In mode

**Callee**
(procedure sub (x, y, z))

Out mode

**Inout mode**

Return

Copyright © 2006 Addison-Wesley. All rights reserved.
Parameter Passing Methods

- 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
Pass-by-Value (In Mode)

• The value of the actual parameter is used to initialize the corresponding formal parameter, which acts as a local variable in the subprogram
  – 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.

• The value can be returned by copy
  – Require extra storage location and copy operation.

• Potential problem: `sub(p1, p1);`
  – Assuming the two formal parameters have different names and be assigned two different values.
  – Whichever formal parameter is copied last will represent the value of `p1` (copy order determine the 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 (address of actual parameters)
- 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 (harmful to readability and reliability)

```c
void fun( int &first, int &second);
...
fun(total, total);  // the first and second in fun will be alias
```
Pass-by-Name (Inout Mode)

• By textual substitution
  – Actual parameter names will replace the corresponding names of formal parameters in all its occurrences in the subprogram.

• 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 passing takes place thru the run-time stack
  – Pass-by-copy: value are copied to stack locations, which then serve as storage for the corresponding formal/actual parameters.
  – 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
• Main call sub(w, x, y, z)
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
    ```
    void fun(int &p1, int p2)  
    p1 is pass-by-reference,  
    p2 is pass-by-value
    ```
• Java
  – All parameters are passed by value
  – Object parameters are passed by reference
Parameter Passing Methods of Major Languages (continued)

• Ada
  – Three parameter passing modes used in formal parameter declaration: in, out, in out; in is default mode
    • out can be assigned but not referenced;
    • in can be referenced but not assigned;
    • in out parameters can be referenced and assigned
  – All scalars are passed by copy; all structured parameters are passed by reference.

• C#
  – Default method: pass-by-value
  – Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref
    • void sum(ref int oldsum)
    • And then it is called by sum(ref mysum);

• PHP: very similar to C#
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 C89: choice is made by the user
  - **No type checking**: name of formal parameters are listed in parentheses, and the type declarations for them follow.
    \[
    \text{double } \sin( \ x ) \\
    \text{double } x; \\
    \{ \ ... \ }
    \]
  - **Support type checking**: using Prototypes, where the formal parameter types are included in the list inside parentheses
    \[
    \text{double } \sin( \text{double} \ x )
    \]
- C99 and C++: all functions must use prototype forms. But type checking can be avoided by using ellipsis as below
  \[
  \text{int } \text{printf}( \text{const char}* \text{format_string}, \ ... \ );
  \]
- Perl, JavaScript, and PHP do not require type checking
Multidimensional Arrays as Parameters

• If a multidimensional array is passed to a subprogram, the compiler needs to know the declared size of the array
  – For subprogram to build the storage mapping function because the subprogram may be separately compiled.
  – For example, the storage mapping function for row-major order matrices with low bound of all indices are 0 and element size is 1.
    \[
    \text{address(mat[i, j])} = \text{address(mat[0,0])} + i \times \text{num_of_cols} + j
    \]
• Note, this storage mapping function need the number of columns. So, the formal parameter must include the number of columns.
  ```c
  void fun( int matrix[ ][10] ) { ... }
  void main() {
    int mat[5][10];
    ...;
    fun( mat );
  }
  ```
Multidimensional Arrays as Parameters: C and C++

• Disallows writing flexible subprograms
  – In previous example, the function cannot accept matrixes with different number of columns
  – e.g., `int mat[5][8];`
    `// cannot be passed to fun`

• Solution: pass a pointer of the array and the sizes of the dimensions as parameters;
  – the user must include the storage mapping function in terms of the size parameters
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
Parameters that are Subprogram Names

• It is sometimes convenient to pass subprogram names as parameters

• Issues:
  
  What is the correct referencing environment for a subprogram that was sent as a parameter?
Parameters that are Subprogram Names: Referencing Environment

• What referencing environment should be used for executing the passed subprogram?
  
  – *Shallow binding*: The environment of the call statement that *executes* the passed subprogram
    • Dynamic scoped language normally uses shallow binding
  
  – *Deep binding*: The environment of the *definition* of the passed subprogram
    • Static scoped languages with nested subprograms is more logical to use deep binding
  
  – *Ad hoc binding*: The environment of the call statement that *passed* the subprogram
    • Never been used
Parameters that are Subprogram Names: Referencing Environment

Example

```javascript
function sub1() {
    var x;
    function sub2() {
        showVar( x );
    };
    function sub3() {
        var x;
        x = 3;
        sub4( sub2 );
    };
    function sub4( subx ) {
        var x;
        x = 4;
        subx();
    };
    x = 1;
    sub3();
};
```

What the value of x is?

1. **shallow-binding is used**
2. **deep-binding is used**
3. **ad hoc binding is used**
Overloaded Subprograms

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.
- Ada, Java, C++, and C# allow overloaded functions with the same name but different parameter profiles.
- In Ada, two overloaded functions can have the same parameter profile. The return type of an overloaded function is used to disambiguate calls.
- It works since Ada disallows mixed-mode expressions.

```plaintext
A, B, : Integer;
...
A := B + Fun(7);
```

Assume two functions named Fun, one returns an int and one returns a float

The one return int
Generic Subprograms

• A *generic* or *polymorphic subprogram* takes parameters of different types on different activations

• 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++

```cpp
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
  
  E.g., `int a, b, c;`  
  ```
  ... 
  c = max(a, b);
  ```

- Instantiated with `int` as parameter
  ```
  int max(int first, int second) { 
      return first > second ? first : second; 
  }
  ```
Coroutines

• A coroutine is a subprogram that has multiple entries and controls them itself
• Also called symmetric control: caller and called coroutines are on a more equal basis
• A coroutine call is named a resume
  – The first call of a coroutine is to its beginning
  – Rather than executing to the end, the coroutine partially execute and then transfer control to some other coroutines.
  – subsequent calls to this coroutine enter at the point just after the last executed statement in the coroutine
• Coroutines repeatedly resume each other, possibly forever
• Coroutines provide quasi-concurrent execution of program units (the coroutines); their execution is interleaved, but not overlapped
Coroutines Illustrated: Possible Execution Controls

(a)
Coroutines Illustrated: Possible Execution Controls

(b)
Coroutines Illustrated: Possible Execution Controls with Loops
Summary

• A subprogram definition describes the actions represented by the subprogram
• Subprograms can be either functions or procedures
• Local variables in subprograms can be stack-dynamic or static
• Three models of parameter passing: in mode, out mode, and inout mode
• Some languages allow operator overloading
• Subprograms can be generic
• A coroutine is a special subprogram with multiple entries