SUBPROGRAMS
Introduction

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

• 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
  - In Python, function definitions are executable; in all other languages, they are non-executable
- 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
- 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
More Definitions

• Function declarations in C and C++ are often called *prototypes*
• A *subprogram declaration* provides the protocol, but not the body, of the subprogram
• 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
Actual/Formal Parameter Correspondence

• 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
Parameter Default Values

• In certain languages (e.g., C++, Python, Ruby, Ada, 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

• Variable numbers of parameters
  – 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.
  – In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk
  – In Lua, a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods
Procedures vs. 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 can 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?
- Can subprograms be overloaded?
- Can subprogram be generic?
Local Variables

• 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
Models of Parameter Passing

[Diagram of parameter passing models: In mode, Out mode, and Inout mode.]
Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)
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, by physical move
  – Require extra storage location and copy operation

• Potential problem: \textit{sub(p1, p1)}; whichever formal parameter is copied back will represent the current value of \textit{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
- Also called pass-by-sharing
- Advantage: Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)
Pass-by-Name (Inout Mode)

• By textual substitution
• 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 communication takes place through the run-time stack
• 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
In Practice

- **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
- **Ada**
  - Three semantics modes of parameter transmission: **in**, **out**, **in out**; **in** is the default mode
  - Formal parameters declared **out** can be assigned but not referenced; those declared **in** can be referenced but not assigned; **in out** parameters can be referenced and assigned
In Practice

- 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#

- Perl: all actual parameters are implicitly placed in a predefined array named `@`

- Python and Ruby use pass-by-assignment (all data values are objects)
Parameter Passing Example

Main
{
    A = 2;
    B = 5;
    C = 8;
    D = 9;
    P(A, B, C, D);
    write(A, B, C, D);
}

P(U, V, W, X) {
    V = U+A;
    W = A+B;
    A = A+1;
    X = A+2;
    write(U, V, W, X)
}

**Parameter mode**

<table>
<thead>
<tr>
<th>Parameter mode</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass by value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass by Reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass by Value Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>