Software II: Principles of Programming Languages

Lecture 9 – Subprograms

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

Subprogram 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
  - In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of `Object`. They can be called without an object, like a function
  - In Lua, all functions are anonymous

Basic Definitions

- 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

Basic Definitions (continued)

- 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

Actual/Formal Parameter Correspondence

- 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

Formal 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 (no keyword parameters)

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

Ruby Blocks

- Ruby includes a number of iterator functions, which are often used to process the elements of arrays
- Iterators are implemented with blocks, which can also be defined by applications
- Blocks are attached methods calls; they can have parameters (in vertical bars); they are executed when the method executes a **`yield`** statement

Ruby Blocks – An Example

```
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first, second = second, first + second
  end
end

puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}
puts
```

Procedures and 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, some program functions will 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?

Design Issues for Subprograms (continued)

- 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?
- If the language allows nested subprograms, are closures supported?

Local Referencing Environments

- 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

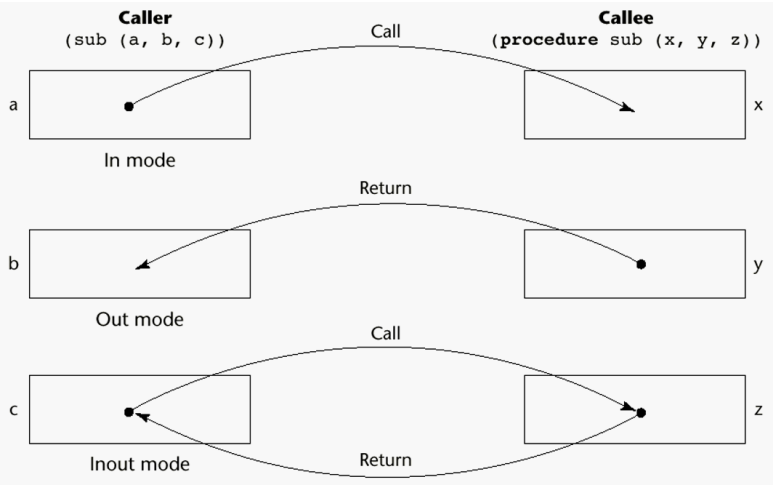
Local Referencing Environments: Examples

- In most contemporary languages, locals are stack dynamic
- In C-based languages, locals are by default stack dynamic, but can be declared `static`
- The methods of C++, Java, Python, and C# only have stack dynamic locals
- In Lua, all implicitly declared variables are global; local variables are declared with `local` and are stack dynamic

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Conceptual Models of Transfer

- Physically move a value
- Move an access path to a value

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

Pass-by-Result (Out Mode) – Potential Problems

- **`sub(p1, p1);`**
  - whichever formal parameter is copied back will represent the current value of **`p1`**
- **`sub(list[sub], sub);`**
  - Compute address of list[sub] at the beginning of the subprogram or end?

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)

```
fun(total, total);  fun(list[i], list[j];  fun(list[i], i);
```

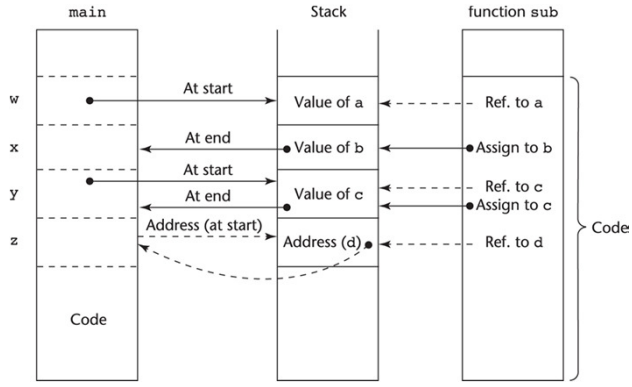
Pass-by-Name (Inout Mode)

- 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
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated

Implementing Parameter-Passing Methods

- In most languages parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack

Implementing Parameter-Passing Methods



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

Parameter Passing Methods of Major Languages

- 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

Parameter Passing Methods of Major Languages (C, C ++, Java)

- 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

Parameter Passing Methods of Major Languages in 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

Parameter Passing Methods of Major Languages (Fortran, C#)

- 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`**

Parameter Passing Methods of Major Languages (PHP, Perl, and Python)

- PHP: very similar to C#, except that either the actual or the formal parameter can specify ref
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal

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 C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

Multidimensional Arrays as Parameters

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function

Multidimensional Arrays as Parameters: C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

Multidimensional Arrays as Parameters: Ada

- Ada – not a problem
  - **Constrained arrays** – size is part of the array's type
  - **Unconstrained arrays** - declared size is part of the object declaration

Multidimensional Arrays as Parameters: Fortran

- Formal parameters that are arrays have a declaration after the header
  - **For single-dimension arrays**, the subscript is irrelevant
  - **For multidimensional arrays**, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function

Multidimensional Arrays as Parameters: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (`length` in Java, `Length` in C#) that is set to the length of the array when the array object is created

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

Design Issues for Functions

- Are side effects allowed?
- What types of return values are allowed?

Are side effects allowed?

- Parameters should always be in-mode to reduce side effect (like Ada)

What types of return values are allowed?

- Most imperative languages restrict the return types
- C allows any type except arrays and functions
- C++ is like C but also allows user-defined types
- Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
- Java and C# methods can return any type (but because methods are not types, they cannot be returned)
- Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
- Lua allows functions to return multiple values

User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- A Python example

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

Use: To compute x + y, **`x.__add__(y)`**