9. Fundamentals of Subprograms

9.1. Introduction

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

9.2. Fundamentals of Subprograms

9.2.1. General Subprogram 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

9.2.2. 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

- Function declarations in C and C++ are often called prototypes
- A subprogram declaration provides the protocol, but not the body, of the subprogram
- Parameter: A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram
- Argument: An actual parameter represents a value or address used in the subprogram call statement

- Argument/Parameter Correspondence
  - Positional
    - The binding of actual parameters (arguments) 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 (argument) 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
9.2.3. Parameters

- **Actual/Formal Parameter Keyword Correspondence: Python Example**
  
  ```python
  sumer
  (length = my_length,
   list = my_list,
   sum = my_sum
  )
  Parameters: length, list, sum
  Arguments: my_length, my_list, my_sum
  ```

- **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
  
    - 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

9.2.4. 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
  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
  ```
9.2.5. Procedures and Functions

- There are two categories of subprograms
  - Procedures are collections 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 have side effects

9.3. 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?

Subprogram Implementation

![Figure 9.3-1 Anatomy of an Activation Record](image-url)
### Figure 9.3-2 Example function \( \text{max}(3,4) \).

9.4. **Local Referencing Environments**

9.4.1. **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
    - Cost of 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
9.4.2. Nested Subprograms

\begin{verbatim}
const
    Max=100;

type
    List=Array[0..Max-1] of integer;
procedure sort(var A:List);
    var i,iom,eol:integer;
    procedure swap(var m,w:integer);
        var b:integer;
        begin{swap}
            b:=m;m:=w;w:=b
        end{swap};
    begin{sort}
        for eol=Max downto 2 do
        begin
            for i:=1 to eol do
            begin{for}
            end{for};
            swap(A[eol],A[iom]);
        end{for}
    end{sort};
\end{verbatim}
9.5. Parameter-Passing Methods

9.5.1. Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

9.5.2. Implementation Models of Parameter Passing

- Conceptual Models of Transfer
  - Physically move a path
  - Move an access path

9.5.2.1. 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 - arrays)
- Disadvantages (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)
9.5.2.2. 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: sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1

- Pass-by-Result: Example 1
  ```c
  void Fixer(out int x, out int y)
  {
    x = 17;
    y = 35;
  }
  ...
  f.Fixer(out a, out a);
  a = 17?
  a = 35?
  ```

- Pass-by-Result: Example 2
  ```c
  void DoIt(out int x, int index)
  {
    x = 17;
    index = 42;
  }
  ...
  sub =21;
  f.DoIt(list[sub],sub);
  list[21]=17?
  list[42]=42?
  ```

9.5.2.3. Pass-by-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

9.5.2.4. Pass-by-Reference (Out 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)

9.5.2.5. 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

9.5.3. Implementing Parameter-Passing Methods

• In most language 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
• 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

• Pass-by-Reference error

```c
void Bad(int& i){i=5;}
...
int a=1;
Bad(1);
a=a+1;
cout << a;
```

9.5.4. Parameter-Passing 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
• 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)

9.5.5. 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

9.5.6. 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

9.5.6.1. 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

9.5.6.2. Multidimensional Arrays as Parameters: Ada – not a problem

• Constrained arrays – size is part of the array’s type
• Unconstrained arrays - declared size is part of the object declaration

9.5.6.3. Multidimensional Arrays as Parameters: Fortran

• Formal parameter 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
9.5.6.4. 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

9.5.7. Design Considerations

- 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

9.6. Parameters that are Subprograms

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter

9.6.1. Parameters that are Subprograms: Parameter Type Checking

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed and their types include the types of the parameters, so parameters can be type checked
- FORTRAN 95 type checks
- Ada does not allow subprogram parameters; an alternative is provided via Ada’s generic facility
- Java does not allow method names to be passed as parameters
9.6.2. Parameters that are Subprograms: Referencing Environment

- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- *Deep binding*: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

```javascript
function sub1()
{
    var x;
    function sub2()
    {
        alert(x); //Creates a dialog box for x
    };
    function sub3()
    {
        var x;
        x=3;
        sub4(sub2);
    };
    function sub4(subx)
    {
        var x;
        x=4;
        subx();
    };
    x=1;
    sub3();
}
```

9.7. 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
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name
Generic Subprograms

- A generic or polymorphic subprogram takes parameters of different types on different activations
- Overloaded subprograms provide *ad hoc polymorphism*
- 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*
  - A cheap compile-time substitute for dynamic binding
- Ada
  - Versions of a generic subprogram are created by the compiler when explicitly instantiated by a declaration statement
  - Generic subprograms are preceded by a generic clause that lists the generic variables, which can be types or other subprograms
- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - Generic subprograms are preceded by a template clause that lists the generic variables, which can be type names or class names
- Java
  - Differences between generics in Java 5.0 and those of C++ and Ada:
    1. Generic parameters in Java 5.0 must be classes
    2. Java 5.0 generic methods are instantiated just once as truly generic methods
    3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
    4. Wildcard types of generic parameters
- C# 2005
  - Supports generic methods that are similar to those of Java 5.0
  - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
  - Examples of parametric polymorphism: C++

```c
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
```c
int max (int first, int second) {
    return first > second ? first : second;
}
```
9.9. Design Issues for Functions

• 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

9.10. User-Defined Overloaded Operators

• Operators can be overloaded in Ada, C++, Python, and Ruby
• An Ada example

\[
\text{function } "*" (A, B: in Vec_Type): return Integer is
  Sum: Integer := 0;
  begin
  for Index in A'range loop
    Sum := Sum + A(Index) * B(Index)
  end loop;
  return sum;
end "*";
\]

\[c := a * b; \quad \text{-- } a, b, \text{ and } c \text{ are of type Vec_Type}\]
9.11. Coroutines

- A coroutine is a subprogram that has multiple entries and controls them itself—supported directly in Lua.
- Also called symmetric control: caller and called coroutines are on a more equal basis.
- A coroutine call is named a resume.
- The first resume of a coroutine is to its beginning, but subsequent calls 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.

```lua
sub co1()
{ ...
    resume co2();
    ...
    resume co3();
    ...
}
```

Coroutines Illustrated: Possible Execution Controls

(a)
Coroutines Illustrated: Possible Execution Controls

(b)

Coroutines Illustrated: Possible Execution Controls with Loops

First resume

Subsequent resume
9.12. 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