Subprograms

CS 315 – Programming Languages
Pinar Duygulu
Bilkent University
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
  - Therefore, only one subprogram is in execution at a given time.
- Control always returns to the caller when the called subprogram's execution terminates
Basic definitions

**Subprogram definition**: heading and the actions of the subprogram.

**Subprogram header**: First line of definition
   Name, formal parameters, their types.

FORTRAN example:
   `SUBROUTINE name (parameters)`

**Subprogram call**: explicit request for execution.

FORTRAN example:
   `CALL name (parameters)`
Parameters

• Parameters in the header are called **formal parameters**.
• Parameters in the call are called **actual parameters**.

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
Parameters can appear in any order
Parameters

Example in Ada,
SUMER (LENGTH => 10,
  LIST => ARR,
  SUM => ARR_SUM);

Formal parameters: LENGTH, LIST, SUM.
Actual parameters: 10, ARR, ARR_SUM.

The programmer doesn’t have to know the order of the formal parameters.
But, must know the names of the formal parameters.
Parameters

• Ada, and FORTRAN 90 allow positional parameters and keyword parameters to be used together.

  SUMER (10, SUM => ARR_SUM, LIST => ARR);

• Once a keyword appears in the call, all remaining parameters must be keyword parameters.
Parameters

- In C++, FORTRAN 90 and 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

Ada example:

```ada
function Comp_Pay (Income: Float;
    Exampptions: Integer := 1;
    Tax_Rate: Float) return Float;
```

Therefore, the call doesn’t have to provide values for all parameters.

A sample call may be

```
Pay := Comp_Pay (2000.0, Tax_Rate => 0.23);
```
Procedures and functions

• There are two categories of subprograms

• Procedures
  – 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 have side effects
Design issues for subprograms

- local variables statically / dynamically allocated
- parameter-passing methods
- type checking between actual / formal parameters
- can pass subprograms as parameters (what is the referencing environment?)
- overloaded subprograms
- generic subprograms (can run on different datatypes)
Local referencing environments

- Local variables of a subprogram can be statically or dynamically bound to storage

- **Statistical binding** \([\text{static local variables}]\)
  
  **Advantages:**
  - efficient (no indirection, allocation / deallocation)
  - no run time effort
  - history sensitive
  
  **Disadvantages:**
  - inability for recursion

- **Dynamic Binding** \([\text{Stack-dynamic local variables}]\)
  
  **Advantages:**
  - allow recursion
  - storage can be shared by several subprograms
  
  **Disadvantages:**
  - Allocation/de-allocation, initialization time
  - indirect addressing
  - no history sensitivity
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
Parameter passing methods

- **Semantic Models of Parameter Passing**

- **in mode**: formal parameters can receive data from corresponding actual parameter

- **out mode**: formal parameters can transmit data to corresponding actual parameter

- **inout mode**: can do both.
Pass by value

- in mode
- value of the actual parameter is used to initialize the value of the formal parameter.
- then the formal parameter behaves as a local variable.
- **Advantage**: Actual variable is protected.
- **Disadvantage**: Costly. A new copy has to be made.
Pass-by-result

- out mode
- no value is transmitted to the subprogram
- corresponding formal parameter acts as a local variable.
- actual parameter must be a variable
- just before termination, the value is passed back to the actual parameter.
- initial value of the actual parameter must not be used by the subprogram.
  - Require extra storage location and copy operation
Pass-by-result

• **Problem: Actual parameter collision**

definition:
subprogram sub(x, y) { x <- 3 ; y <-5;}
call:
sub(p, p)
*what is the value of p here? (3 or 5?)*

• The values of x and y will be copied back to p. Which ever is assigned last will determine the value of p.
• The order is important
• The order is implementation dependent ⇒ Portability problems.
Pass-by-result

- **Problem:** Time to evaluate the address of the actual parameter
  - at the time of the call
  - at the time of the return
- The decision is up to the implementor.

**definition:**
subprogram sub(x)
i <- 5 is changed as a global variable here
x <- ..
call:
i <- 3
sub(A[i])
The decision is up to the implementor.
Pass-by-value-result

- inout mode
- actual values are moved
- also called as pass-by-copy
- combination of pass-by-value and pass-by-result
- the value of the actual parameter is used to initialize the corresponding formal parameter
- the formal parameter acts as a local parameter
- At termination, the value of the formal parameter is copied back.
Pass-by-reference

- inout mode
- access path is transferred
- actual parameter is shared with the called subprogram
- Also called as pass-by-sharing

**Advantage:**
- **Efficient:**
  - No need to duplicate the space
  - No need to copy values
Pass-by-reference

- **Disadvantage:** *Slow access:* Extra level of indirection
- **Dangerous:** Actual parameter may be modified unintentionally.
- **Aliasing:**
  - definition:
    subprogram sub(x, y)
    call:
    sub(p, p)
  Here, $x$ and $y$ in the subprogram are aliases.
- **Another way of aliasing:**
  - definition:
    var x *global variable*
    subprogram sub(y)
    call:
    sub(x)
  Here, $x$ and $y$ in the subprogram are aliases.
Pass-by-name

• **inout mode**
• **Actual parameter** is textually substituted for the corresponding formal parameter in all occurrences in the subprogram.

• **Late binding**: actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.

• If the actual parameter is a **scalar variable**, then it is equivalent to **pass-by-reference**.
• If the actual parameter is a **constant expression**, then it is equivalent to **pass-by-value**.

• **Advantage**: flexibility
• **Disadvantage**: slow execution, difficult to implement, confusing.
Pass-by-name

Example

procedure BIG;
integer GLOBAL;
integer array LIST[1:2];
procedure SUB (P: integer);
begin
P := 3;
GLOBAL := GLOBAL + 1;
P := 5;
end;
begin
LIST[1] := 3
GLOBAL := GLOBAL + 1
LIST[GLOBAL] :=5
end.

Execution:

LIST[1] := 3
GLOBAL := 1 + 1
Parameter passing methods of the major languages

- **FORTRAN 77** and later implementations use **pass-by-value-result** (Earlier implementations used call-by-reference)
- **C** uses **pass-by-value, pass-by-reference** is achieved by pointers.
- **Java**: all parameters are passed by value, object 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`
- **PHP**: very similar to C#
- **Perl**: all actual parameters are implicitly placed in a predefined array named `@_`
- **Ada** uses **in** (default), **out**, and **inout** parameters.
  ```
  procedure Adder (A: in out Integer;
                  B: in Integer;
                  C: out Float);
  ```
- **Fortran 95** is similar to Ada:
  ```
  Subroutine Adder (A, B, C)
  Integer, Intent(Inout) :: A
  Integer, Intent(In) :: B
  Integer, Intent(Out) :: C
  ```
Type-checking parameters

- Types of actual and formal parameters should be checked.
- Otherwise, small typographical errors can lead to errors that are difficult to detect.
- 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
Multidimensional arrays as parameters

- In languages C and C++, subprograms can be compiled separately from the calling programs.
- The compiler must be able to build the map function for a multidimensional array, when passed as a parameter.
- These languages store arrays in row major order.
- Since the mapping function for row major order needs the number of columns, it must be provided. For example,

```c
void
assign (float matrix[][5], float val, int r, int c)
{
    matrix[r][c] = val;
}

void
assign (float matrix[[r][c]], float val, int r, int c)
{
    address(matrix[[r][c]]) =
    address(matrix[0][0])+(r*n+c)*float_size
}
```

```c
main ()
{
    float m1[10][5], m2[15][5];
    assign (m1, 1.2, 3, 4);
    assign (m2, 2.3, 15, 2);
}
```
Multidimensional arrays as parameters

- Problem: It is not possible to write functions that accept any size of multidimensional arrays. So, it is not flexible.
- Solution is to pass the array as a pointer, and pass the sizes as parameters.

```c
void assign (float *matrix, int num_cols, float val, int r, int c) {
    *(matrix + (r * num_cols) + c) = value
}
```

- This is a solution, but the resulting code is not readable. Further, it is not writable, the programmer must write the map function himself.
Parameters that are subprograms

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  - The address of the code segment of the called subprogram must be passed.
  - Description of the subprogram’s parameters must be sent.
  - In C and C++, functions cannot be passed as parameters, but pointers to functions can be passed.
  - In languages that allow nested subprograms, such as JavaScript, there is a problem of referencing environment.
Parameters that are subprograms

- **Important question** is:
- What is the **referencing environment** for executing the passed subprogram? (For non local variables)
- There are 3 choice:
  - **Shallow binding**: the environment of the calling program.
  - **Deep binding**: the environment of the subprogram in which the called subprogram is declared.
  - **Never been used**: the environment of the subprogram that includes the call statement that passed the subprogram as an actual parameter.

- **Shallow binding**: appropriate for dynamic scope binding.
- **Deep binding**: appropriate for static scope binding.
Parameters that are subprograms

Example:

```javascript
function sub1() {
    var x;  // declared in
    function sub2() {
        window.status = x;
    } // sub2
    function sub3() {
        var x;
        x = 3;
        sub4(sub2);    // passed in
    } // sub3
    function sub4(subx) {
        var x;
        x = 1;
        subx();  // called by
    } // sub4
    x = 2;
    sub3();
} // sub1
```

Passed subprogram S2 is called by S4
is declared in S1
is passed in S3

Output:

is called by S4
is declared in S1
is passed in S3
Overloaded subprograms

- An overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment.
- Each one must have a different number of arguments, or order of arguments or types of arguments to distinguish the one meant.
- The correct meaning (the correct code) to be invoked is determined by the actual parameter list. The code whose formal parameter types match the types of the actual parameters is used.
- In case of functions, the return type may be used to distinguish.
- Ada and C++ allow subprograms to be overloaded.
Generic subprograms

- A subprogram whose parameters can be of different types.
- The same formal parameter can get values of different types.
- Ada and C++ provide Generic (Polymorphic) Subprograms
Generic subprograms

Generic Subprograms in Ada

Example:

```
generic

type ELEMENT is private;
type VECTOR is array (INTEGER range <>) of ELEMENT;
procedure GENERIC_SORT(LIST: in out VECTOR);

procedure GENERIC_SORT(LIST: in out VECTOR) is
  TEMP: ELEMENT;
  begin
  . . .
  end GENERIC_SORT;
```

This is a template for a procedure. No code is generated here by the compiler. It can be instantiated as

```
procedure INT_SORT is new GENERIC_SORT (ELEMENT => INTEGER;
  VECTOR => INT_ARRAY);
```

At this point, the compiler builds a version of GENERIC_SORT, and names it INTEGER_SORT.
Generic subprograms

Generic Functions in C++

They are called **template functions**.

Example:

```cpp
template <class Type>
Type max (Type first, Type second) {
    return first > second ? first : second;
}
```

This function can be instantiated for any type for which `>` is defined.

It can be instantiated implicitly when function is named in a call as:

```cpp
int a, b, c;
char d, e, f;
...
c = max (a,b);
f = max (d,e);
```
Design issues for functions

• Functional side effects;
• Return types.
User defined overloaded operators

- Allowed in Ada and C++
- Example in Ada
- function “*” (A, B: in MATRIX) return MATRIX is
  . . .
Coroutines

- A coroutine is a special kind of subprogram.
- Rather than master-slave relationship, as in subprograms; a caller and called coroutines are on a more equal basis.
- This is called symmetric unit control model.
- SIMULA67 is the first high-level PL that included coroutines.
- SIMULA67 was a PL suitable for Simulation.
- Modula-2 also support coroutines.
- Coroutines have multiple entry points. Entry points are controlled by the coroutine itself.
- Coroutines are history sensitive, thus they have static variables.
- Coroutines are not called, they are resumed. Secondary execution does not necessarily start from the beginning.
- Coroutines are created in an application by a special unit called master unit, which is not a coroutine.
- A coroutine may have an initialization code that is executed only when it is created.
- Only one coroutine executes at a given time.
- There is a single thread of control in a program with coroutines.
- The master unit resumes one of the created coroutines, then coroutines resume each other. Finally one coroutine reaches the end of execution, transfers the control to the master unit.
Coroutines