

Subprograms

In Text: Chapter 8

1

Outline

- Definitions
- Referencing environments
- Parameter passing modes and mechanisms
- Independent & separate compilation
- Advanced subprogram issues

■ Chapter 8: Subprograms ■

2

Characteristics of Subprograms

- Fundamental characteristics:
 - A subprogram has a single entry point
 - The caller is suspended during execution of the called subprogram
 - Control always returns to the caller when the called subprogram's execution terminates

■ Chapter 8: Subprograms ■

3

Basic Definitions

- A **subprogram definition** is a description of the actions of the subprogram abstraction
- A **subprogram call** is an explicit request that the subprogram be executed
- A **subprogram header** is the first line of the definition, including the name, the kind of subprogram, and the formal parameters
- The **parameter profile** of a subprogram is the number, order, and types of its parameters
- The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return type

■ Chapter 8: Subprograms ■

4

More Definitions

- 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

■ Chapter 8: Subprograms ■

5

Actual/Formal Param Correspondence

- Two basic choices:
 - Positional
 - Keyword
- Sort (List => A, Length => N);
- For named association:
 - Advantage: order is irrelevant
 - Disadvantage: user must know the formal parameter's names

■ Chapter 8: Subprograms ■

6

Default Parameter Values

- Example, in Ada:

```
procedure sort (list : List_Type;  
               length : Integer := 100);  
    ...  
sort (list => A);
```

Two Types of Subprograms

- **Procedures** provide user-defined statements
- **Functions** provide user-defined operators

Design Issues for Subprograms

- What parameter passing methods are provided?
- Are parameter types checked?
- Are local variables static or dynamic?
- What is the referencing environment of a passed subprogram?
- Are parameter types in passed subprograms checked?
- Can subprogram definitions be nested?
- Can subprograms be overloaded?
- Are subprograms allowed to be generic?
- Is separate/independent compilation supported?

Local Referencing Environments

- If local variables are stack-dynamic:
 - Advantages:
 - Support for recursion
 - Storage for locals is shared among some subprograms
 - Disadvantages:
 - Allocation/deallocation time
 - Indirect addressing
 - Subprograms cannot be history sensitive
- Static locals are the opposite

Local Variables in Example PLs

- FORTRAN 77 and 90—most are static, but can have either (SAVE forces static)
- C—both; default is stack dynamic, but variables declared to be static are
- Pascal, Modula-2, and Ada—dynamic only

Parameters and Parameter Passing

- Semantic Models: in mode, out mode, inout mode
- Conceptual Models of Transfer:
 - Physically move a value
 - Move an access path
- Implementation Models:
 - Pass-by-value
 - Pass-by-result
 - Pass-by-value-result
 - Pass-by-reference
 - Pass-by-name

Pass-By-Value

- in mode
- Either by physical move or access path
- Disadvantages of access path method:
 - Must write-protect in the called subprogram
 - Accesses cost more (indirect addressing)
- Disadvantages of physical move:
 - Requires more storage
 - Cost of the moves

■ Chapter 8: Subprograms ■

13

Pass-By-Result

- out mode
 - Local's value is passed back to the caller
 - Physical move is usually used
 - Disadvantages:
 - If value is moved, time and space
 - In both cases, order dependence may be a problem
- ```
procedure sub1(y: int, z: int);
 ...
 sub1(x, x);
```
- Value of x in the caller depends on order of assignments at the return

■ Chapter 8: Subprograms ■

14

---

---

---

---

---

---

---

---

## Pass-By-Value-Result

- inout mode
- Physical move, both ways
- Also called pass-by-copy
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

■ Chapter 8: Subprograms ■

15

---

---

---

---

---

---

---

---

## Pass-By-Reference

- inout mode
- Pass an access path
- Also called pass-by-sharing
- Advantage: passing process is efficient
- Disadvantages:
  - Slower accesses
  - Can allow aliasing:
    - Actual parameter collisions: `sub1(x, x);`
    - Array element collisions: `sub1(a[i], a[j]); /* if i = j */`
    - Collision between formals and globals
- Root cause of all of these is: The called subprogram is provided wider access to nonlocals than is necessary
- Pass-by-value-result does not allow these aliases (but has other problems!)

■ Chapter 8: Subprograms ■

16

---

---

---

---

---

---

---

---

## Pass-By-Name

- multiple modes
- 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
- Purpose: flexibility of late binding
- Resulting semantics:
  - If actual is a scalar variable, it is pass-by-reference
  - If actual is a constant expression, it is pass-by-value
  - If actual is an array element, it is like nothing else
  - If actual is an expression with a reference to a variable that is also accessible in the program, it is also like nothing else

■ Chapter 8: Subprograms ■

17

---

---

---

---

---

---

---

---

## Pass-By-Name Example 1

```
procedure sub1(x: int; y: int);
begin
 x := 1;
 y := 2;
 x := 2;
 y := 3;
end;

sub1(i, a[i]);
```

■ Chapter 8: Subprograms ■

18

---

---

---

---

---

---

---

---

## Pass-By-Name Example 2

■ Assume k is a global variable

```
procedure sub1(x: int; y: int; z: int);
begin
 k := 1;
 y := x;
 k := 5;
 z := x;
end;

sub1(k+1, j, i);
```

---

---

---

---

---

---

---

---

## Disadvantages of Pas-By-Name

- Very inefficient references
- Too tricky; hard to read and understand

---

---

---

---

---

---

---

---

## Param Passing: Language Examples

- FORTRAN
  - Before 77, pass-by-reference
  - 77—scalar variables are often passed by value-result
- ALGOL 60
  - Pass-by-name is default; pass-by-value is optional
- ALGOL W: Pass-by-value-result
- C: Pass-by-value
- Pascal and Modula-2: Default is pass-by-value; pass-by-reference is optional

---

---

---

---

---

---

---

---

## Param Passing: PL Examples (cont.)

- C++: Like C, but also allows reference type parameters, which provide the efficiency of pass-by-reference with in-mode semantics
- Ada
  - All three semantic modes are available
  - If out, it cannot be referenced
  - If in, it cannot be assigned
- Java
  - Like C++, except only references

■ Chapter 8: Subprograms ■

22

---

---

---

---

---

---

---

---

## Type Checking Parameters

- Now considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal, Modula-2, FORTRAN 90, Java, and Ada: it is always required
- ANSI C and C++: choice is made by the user

■ Chapter 8: Subprograms ■

23

---

---

---

---

---

---

---

---

## Implementing Parameter Passing

- ALGOL 60 and most of its descendants use the run-time stack
- Value—copy it to the stack; references are indirect to the stack
- Result—same
- Reference—regardless of form, put the address in the stack
- Name:
  - Run-time resident code segments or subprograms evaluate the address of the parameter
  - Called for each reference to the formal
  - These are called thunks
  - Very expensive, compared to reference or value-result

■ Chapter 8: Subprograms ■

24

---

---

---

---

---

---

---

---



## Ada Param Passing Implementations

- Simple variables are passed by copy (value-result)
- Structured types can be either by copy or reference
- This can be a problem, because
  - Aliasing differences (reference allows aliases, but value-result does not)
  - Procedure termination by error can produce different actual parameter results
- Programs with such errors are "erroneous"

■ Chapter 8: Subprograms ■

25

---

---

---

---

---

---

---

---

## Multidimensional Arrays as Params

- 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
- C and C++
  - Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
  - This 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, which is in terms of the size parameters (See example, p. 351)

■ Chapter 8: Subprograms ■

26

---

---

---

---

---

---

---

---

## More Array Passing Designs

- Pascal
  - Not a problem (declared size is part of the array's type)
- Ada
  - Constrained arrays—like Pascal
  - Unconstrained arrays—declared size is part of the object declaration (See book example p. 351)
- Pre-90 FORTRAN
  - Formal parameter declarations for arrays can include passed parameters

```
SUBPROGRAM SUB(MATRIX, ROWS, COLS, RESULT)
 INTEGER ROWS, COLS
 REAL MATRIX (ROWS, COLS), RESULT
 ...
END
```

■ Chapter 8: Subprograms ■

27

---

---

---

---

---

---

---

---

## Design Considerations for Parameter Passing

- Efficiency
- One-way or two-way
- These two are in conflict with one another!
- Good programming => limited access to variables, which means one-way whenever possible
- Efficiency => pass by reference is fastest way to pass structures of significant size
- Also, functions should not allow reference parameters

■ Chapter 8: Subprograms ■

28

---

---

---

---

---

---

---

---

## Subprograms As Parameters: Issues

- Are parameter types checked?
  - Early Pascal and FORTRAN 77 do not
  - Later versions of Pascal, Modula-2, and FORTRAN 90 do
  - Ada does not allow subprogram parameters
  - C and C++ - pass pointers to functions; parameters can be type checked

■ Chapter 8: Subprograms ■

29

---

---

---

---

---

---

---

---

## Subprogs as Params: Issues (cont.)

- What is the correct referencing environment for a subprogram that was sent as a parameter?
  - Possibilities:
    - It is that of the subprogram that called it (shallow binding)
    - It is that of the subprogram that declared it (deep binding)
    - It is that of the subprogram that passed it (ad hoc binding, never been used)
  - For static-scoped languages, deep binding is most natural
  - For dynamic-scoped languages, shallow binding is most natural

■ Chapter 8: Subprograms ■

30

---

---

---

---

---

---

---

---

## Overloading

- An **overloaded subprogram** is one that has the same name as another subprogram in the same referencing environment
- C++ and Ada have overloaded subprograms built-in, and users can write their own overloaded subprograms

■ Chapter 8: Subprograms ■

31

---

---

---

---

---

---

---

---

## Generic Subprograms

- A **generic** or polymorphic subprogram is one that 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**
- See Ada generic and C++ template examples in text

■ Chapter 8: Subprograms ■

32

---

---

---

---

---

---

---

---

## Independent and Separate Comp.

- **Independent compilation** is compilation of some of the units of a program separately from the rest of the program, without the benefit of interface information
- **Separate compilation** is compilation of some of the units of a program separately from the rest of the program, using interface information to check the correctness of the interface between the two parts

■ Chapter 8: Subprograms ■

33

---

---

---

---

---

---

---

---

## Language Examples:

- FORTRAN II to FORTRAN 77: independent
- FORTRAN 90, Ada, Modula-2, C++: separate
- Pascal: allows neither

---

---

---

---

---

---

---

---

## Functions

### Design Issues:

- Are side effects allowed?
  - Two-way parameters (Ada does not allow)
  - Nonlocal reference (all allow)
- What types of return values are allowed?
  - FORTRAN, Pascal, Modula-2: only simple types
  - C: any type except functions and arrays
  - Ada: any type (but subprograms are not types)
  - C++ and Java: like C, but also allow classes to be returned

---

---

---

---

---

---

---

---

## Accessing Nonlocal Environments

- The **nonlocal variables** of a subprogram are those that are visible but not declared in the subprogram
- **Global variables** are those that may be visible in all of the subprograms of a program

---

---

---

---

---

---

---

---

## Methods for Accessing Nonlocals

- FORTRAN COMMON
  - The only way in pre-90 FORTRANs to access nonlocal variables
  - Can be used to share data or share storage
- Static scoping—discussed in Chapter 4
- External declarations: C
  - Subprograms are not nested
  - Globals are created by external declarations (they are simply defined outside any function)
  - Access is by either implicit or explicit declaration
  - Declarations (not definitions) give types to externally defined variables (and say they are defined elsewhere)
- External modules: Ada and Modula-2: Sec Ch. 10
- Dynamic Scope: discussed in Ch. 4

■ Chapter 8: Subprograms ■

37

---

---

---

---

---

---

---

---

## User-Defined Overloaded Operators

- Nearly all programming languages have overloaded operators
- Users can further overload operators in C++ and Ada (not carried over into Java)
- Ada Example (where Vector\_Type is an array of Integers):  
function "\*" (a, b : in Vector\_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 "\*";
- Are user-defined overloaded operators good or bad?

■ Chapter 8: Subprograms ■

38

---

---

---

---

---

---

---

---

## Coroutines

- A **coroutine** is a subprogram that has multiple entries and controls them itself
- Also called **symmetric control**
- 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
- Typically, coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasiconcurrent execution of program units (the coroutines)
- Their execution is interleaved, but not overlapped

■ Chapter 8: Subprograms ■

39

---

---

---

---

---

---

---

---