

Chapter 8

Fundamental Characteristics of Subprograms

1. A subprogram has a single entry point
2. The caller is suspended during execution called subprogram
3. Control always returns to the caller when called subprogram execution terminates

Basic definitions:

A subprogram definition is a description of the actions of the subprogram abstraction

A subprogram call is an explicit request that 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

Chapter 8

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:

1. Positional
2. Keyword
e.g. SORT(LIST => A, LENGTH => N);

Advantage: order is irrelevant
Disadvantage: user must know the formal parameter names

Default Values:
e.g. procedure SORT(LIST : LIST_TYPE;
 LENGTH : INTEGER := 100);
 ...
 SORT(LIST => A);

Chapter 8

Procedures provide user-defined statements

Functions provide user-defined operators

Design Issues for Subprograms

1. What parameter passing methods are provided?
2. Are parameter types checked?
3. Are local variables static or dynamic?
4. What is the referencing environment of a parameter in a subprogram?
5. Are parameter types in passed subprograms checked?
6. Can subprogram definitions be nested?
7. Can subprograms be overloaded?
8. Are subprograms allowed to be generic?
9. Is separate or independent compilation supported?

Chapter 8

Local referencing environments

If local variables are stack-dynamic:

- Advantages:
 - a. Support for recursion
 - b. Storage for locals is shared among subprograms
- Disadvantages:
 - a. Allocation time
 - b. Indirect addressing
 - c. Subprograms cannot be history sensitive

Static locals are the opposite

Language Examples:

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

Chapter 8

Parameters and Parameter Passing

Semantic Models in mode, out mode, inout mode

Conceptual Models of Transfer:

1. Physically move a value
2. Move an access path

Implementation Models:

1. 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

2. Pass-by-result (out mode)

- Local value is passed back to the caller
- Physical move is usually used
- Disadvantages:
 - a. If value is passed, time and space
 - b. In both cases, order dependence problem

```
procedure sub1(y: int, z: int);  
..  
sub1(x, x);
```

x Value of caller depends on order of assignments at the return

3. 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

4. Pass-by-reference (mode)

- Pass an access path
- Also called pass-by-sharing

- Advantage: passing process is efficient

- Disadvantages:

- a. Slower accesses
- b. Collisions

. Actual parameter collisions:

e.g. `procedure sub1(a: int);`

`...
sub1(x, x);`

ii. Array element collisions:

e.g.

```
sub1(a[i], a[j]); /* if i = j */  
sub2(a[i], a[i]);
```

iii. Collision from global variables

- Root cause of all of these is: subprogram is provided wider **visibility** than necessary

- Pass-by-value-result does not have aliases (but has other problems)

Chapter 8

5. Pass-by-name (multiple mode)

- By textual substitution

Formals are bound to an access method at the time of the call, but actual binding or address takes place at the time of 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

e.g.

```
procedure sub1(x: int; y: int);
```

```
begin
```

```
  x := 1;
```

```
  y := 2;
```

```
  x := 2;
```

```
  y := 3;
```

```
end;
```

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

Chapter 8

- If actual is an expression with a side effect, a variable that is also accessible in the program, it is also like nothing else

e.g. `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 pass by name:

- Very inefficient references

- Too tricky; hard to read and understand

Language Examples:

1. FORTRAN

- Before 77, pass-by-reference

- 77 - scalar variables are often pass-by-value

- result

2. ALGOL 60

- Pass-by-name is default; pass-by-value is optional

Chapter 8

3. ALGOL W
 - Pass-by-value-result
4. C
 - Pass-by-value
5. Pascal and Modula-2
 - Default is pass-by-value; pass-by-reference optional
6. C++
 - Like C, but also allows reference type parameters, which provide the efficient pass-by-reference with in-mode semantics
7. Ada
 - All three semantic modes are available; if cannot be referenced in, it cannot be assigned
8. Java
 - Like C++, except only references

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

Copyright © 1998 by Addison Wesley Longman, Inc. **10**

Chapter 8

Implementing Parameter Passing

ALGOL 60 and most of its descendants use the run-time stack

- Value - copy it to the stack; references 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 formal; these are unstacked
 - Very expensive, compared to reference or value-result

Ada

- Simple variables are passed by copy (value)
- Structured types can be either by copy or by reference
 - This can be a problem, because
 - a) Of aliases (reference allows aliasing; value-result does not)
 - b) Procedure termination by error can be different actual parameter result
- Programs with such errors are rare

Copyright © 1998 by Addison Wesley Longman, Inc. **11**

Chapter 8

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 stack mapping function
- C and C++
 - Programmer is required to include the declared sizes of all but the first subscript in the parameter list
 - This disallows writing flexible subprograms
 - Solution: pass a pointer to the array; the user must include the storage mapping function, which is in terms of the subscript parameters (See example, p. 351)
- Pascal
 - Not a problem (declared size is part of the type)
- Ada
 - Constrained arrays - like Pascal
 - Unconstrained arrays - declared size is part of the object declaration (See book example)

Copyright © 1998 by Addison Wesley Longman, Inc. **12**

Chapter 8

- Pre-90 FORTRAN
 - Formal parameter declarations for array include passed parameters
 - e.g.


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

Design Considerations for Parameter Passing

1. Efficiency
2. 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 pass structures of significant size

- Also, functions should not allow reference parameters

Copyright © 1998 by Addison Wesley Longman, Inc. **13**

Chapter 8

Parameters that are Subprogram Names

Issues:

1. Are parameter types checked?
 - Early Pascal and FORTRAN 77 do not
 - Later versions of Pascal, Modula-2 and FORTRAN 90 do
 - ~~Al~~ies not allow subprogram parameters
 - C and C++ - pass pointers to functions parameters can be type checked
2. What is the correct referencing environment subprogram that was sent as a parameter
 - Possibilities:
 - a. It is that of the subprogram that enabled shallow binding
 - b. It is that of the subprogram that declared Deep-binding
 - c. It is that of the subprogram that passed Ad hoc binding (Has never been used)

Copyright © 1998 by Addison Wesley Longman, Inc. **14**

Chapter 8

- For static-scoped languages, deep binding is most natural
- For dynamic-scoped languages, shallow binding is most natural

Example:

```

sub1
├── sub2
│   ├── sub3
│   │   └── call sub4(sub2)
│   └── sub4(subx)
│       └── call subx
└── call sub3
  
```

What is the referencing environment of sub2 when it is called?

Copyright © 1998 by Addison Wesley Longman, Inc. **15**

Chapter 8

C++ template functions are instantiated *implicitly* when the function is named in a call or when its address is taken with the `&` operator

Another example:

```
template <class Type>
void generic_sort(Type list[], int len) {
    int top, bottom;
    Type temp;
    for (top = 0; top < len - 2; top++)
        for (bottom = top + 1; bottom < len - 1;
            bottom++) {
            if (list[top] > list[bottom]) {
                temp = list [top];
                list[top] = list[bottom];
                list[bottom] = temp;
            } /** end of for (bottom = ...
        } /** end of generic_sort
}
```

Example use:

```
float flt_list[100];
...
generic_sort(flt_list, 100); // Implicit
                             // instantiation
```

Chapter 8

Def: Independent compilation: compilation of some of the units of a program separately from the rest of the program, without the interface information

Def: Separate compilation: 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.

Language Examples:

FORTRAN II to FORTRAN 77 - independent

FORTRAN 90, Ada Modula-2, C++ - separate

Pascal - allows neither

Functions

Design Issues:

1. Are side effects allowed?
 - a. Two-way parameter passing (not allow)
2. What types of return values are allowed?

Chapter 8

Functions (continued)

Language Examples for possible return types:

1. FORTRAN, Pascal, Modula-2 - only simple types
2. C - any type except functions and arrays
3. Ada - any type (but subprograms are not types)
4. C++ and Java - like C, but also allow complex types to be returned

Accessing Nonlocal Environments

Def: Thenonlocal variables of a subprogram are those that are visible but not declared in the subprogram

Def: Global variables are those that may be visible in all of the subprograms of a program

Chapter 8

Methods:

1. FORTRAN COMMON
 - The only way in FORTRAN to access nonvariables
 - Can be used to share data or share state
2. Static scoping discussed in Chapter 4
3. External declarations
 - Subprograms are not nested
 - Globals created by external declarations (they are simply defined outside any
 - Access is by either implicit or explicit declaration
 - Declarations (not definitions) give type externally defined variables (and say defined elsewhere)
4. External modules and Modules
 - More about these later (Chapter 10)
5. Dynamic Scope discussed in Chapter 4

Chapter 8

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)

Example (Ada) (assume VECTOR_TYPE has been defined to be an array type with elements):

```
function ***(A, B : in VECTOR_TYPE)
  return INTEGER is
  SUM : INTEGER := 0;
  begin
  for INDEX in A's range loop
    SUM := SUM + A(INDEX) * B(INDEX);
  end loop;
  return SUM;
end ***;
```

Are user-defined overloaded operators good or

Chapter 8

Coroutines

A coroutine is a subprogram that has multiple entries and controls them itself

- Also called symmetric control
- A coroutine call is named resume
- The first resume of a coroutine starts beginning but subsequent calls enter at the point just the last executed statement in the routine
- Typically, coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasiconcurrent execution of program units (the coroutines)
- Their execution is interleaved, but not overlapped
