

Ch.8: Subprograms (incomplete)

I. Introduction

- A. Programming language can have abstraction of *process* and abstraction of *data*. Process abstraction was the first to be explored.
- B. Babbage's Analytical Engine: could reuse collections of instruction cards. These could be viewed as a subprogram, which is an abstraction of process

II. Fundamentals of Subprograms

A. General Subprogram Characteristics

- 1. Each subprogram has a single entry point
- 2. The calling program unit is suspended during execution of the called subprogram
- 3. Control always returns to the caller when the subprogram execution terminates

B. Basic Definitions

- 1. A subprogram *definition* describes the actions of the subprogram
- 2. A subprogram *call* is the explicit request that the subprogram be executed.
- 3. A subprogram is *active* if it has been called and is still executing.
- 4. A subprogram *header* is the first line of the definition serves to:
 - a. specify that the following code is a syntactic unit of some particular kind (e.g. Procedure or Function)
 - b. provide a name for the subprogram
 - c. optionally specify a parameter list

5. Examples:

- a. In FORTRAN: SUBROUTINE ADDER (parameters)
- b. In Ada: procedure ADDER(parameters)
- c. In C: adder(parameters)

- 6. An *argument signature* (parameter profile) is the number, order, and types of its formal parameters
- 7. The *protocol* of a subprogram is its argument signature plus its return type (if a function)
- 8. Subprogram declarations are also known as *prototypes*, *forward declarations*, or *external declarations*

C. Parameters

- 1. Subprograms can access the data it is to process through
 - a. Access to nonlocal variables
 - b. Parameter Passing
- 2. Computations, rather than data may also be transmitted by using the name of a subprogram as a parameter
- 3. Parameters in the subprogram header are called *formal parameters*, sometimes called dummy variables
- 4. Subprogram call statements' list of parameters are called *actual parameters*, which are different from formal parameters
- 5. Correspondance between actual and formal parameters is usually done by position (*positional parameters*), though it can also be done by keyword (*keyword parameters*)
 - a. Eg. keyword parameters in Ada

```
sumer ( length => my_length,  
        list => my_array,  
        sum => my_sum);
```

b. E.g. mixing these two kinds of parameters in Ada

```
sumer ( length,  
        list => my_array,  
        sum => my_sum);
```

Note that after a keyword parameter appears in the list, all remaining parameters must be keyword parameters, as the keyword parameters need not be in any order.

6. Default Values for formal parameters (C++, FORTRAN 90, Ada)

a. E.g. in Ada

```
function compute_pay ( income: float;  
                      exemptions: integer := 1;  
                      tax_rate: float) return float:
```

No comma is needed for an absent actual parameter in Ada, since all subsequent parameters must be keyword parameters. E.g.

```
answer:= compute_pay ( 2000.0, tax_rate => 0.15);
```

b. In C++ there are no keyword parameters (only positional), so default parameters must appear last. E.g.

```
float compute_pay ( float income,  
                   float tax_rate,  
                   int exemptions = 1)
```

where a call using the default parameter could look like:

```
answer:= compute_pay (2000.0, 0.15); // default used for exemptions
```

c. In C there may be fewer actual parameters than formal parameters (such as for printf). It is the programmer's responsibility to ensure meaningful correspondance.

D. Procedures & Functions

1. Procedures define new statements

- a. They are collections of statements defining parameterized computations enacted by a single calling statement
- b. Results are produced by changing either non-local variables or parameters
- c. E.g. PROCEDURE sort (number_array: integer_array_type; limit: INTEGER)

2. Functions define new user-defined operators

- a. Modelled after mathematical functions, returning a single value
- b. E.g. FUNCTION average (number_array: integer_array_type): REAL

III. Design Issues for Subprograms

- A. What parameter-passing method or methods are used?
- B. Are the types of the actual parameters checked against the types of the formal parameters?
- C. Are local variables statically or dynamically allocated?
- D. If subprograms can be passed as parameters, what is the referencing environment of such a subprogram?
- E. If subprograms can be passed as parameters, are the types of parameters checked in calls to the passed subprograms?
- F. Can subprograms be overloaded?
- G. Can subprograms be generic?
- H. Is either separate or independent compilation possible?

IV. Local Referencing Environments

A. Subprograms own internally-defined variables are called *local variables*

B. Local variables can be bound either statically or dynamically

1. Static binding: Faster, more efficient, allows history-sensitive variables

e.g. C's *static*:

```
// Add list elements to on-going sum
int adder (int list[], int listLength) {
    static int sum = 0;    {Allocated at compile time}
    for( int count = 0; count < listLength; count++)
        sum += list[ count];
    return sum;
}
```

2. Dynamic binding: needed for recursion, requires more overhead to dereference since references must be indirect (offset in the stack)

V. Parameter-Passing Methods

A. Semantics models of Parameter Passing

1. Modes:

a. In mode: formal parameter used to receive data from the actual parameter

b. Out mode: formal parameter ignores corresponding actual parameter value, used to return data into the actual parameter

c. Inout mode: both of the above

2. Data transfer involves either

a. copying a value (data transfer)

b. transmitting an access path (address, or pointer)

B. Implementation Models of Parameter Passing

1. Pass-by-Value

a. The value of the actual parameter is used to initialize the formal parameter, which then acts as a local variable (in mode).

b. This is normally implemented by data transfer

c. Main disadvantage: additional storage is required, which can be expensive for large objects (may want to pass it as a variable parameter even if it won't be changed...)

2. Pass-by-Result

a. Used for out-mode parameters.

b. One problem: possible parameter collision:

```
sub( a: OUT INTEGER; b: OUT INTEGER) {
    ...
    b:= 5;
    a:= 7;
}
...
sub ( p1, p1);    { The subprogram call }
```

Depending on whether out-parameters are stored to actual parameters left-to-right or right-to-left will set p1 to either 5 or 7. This is compiler-dependent

c. Another problem: when is the formal parameter evaluated? If subprogram has parameter `list[index]`, then is the value for `index` set on the way in or once the value is being returned? In fact, `index` could be changed inside the subroutine.

3. Pass-by-Value-Result (aka pass-by-copy)

- a. Combines pass-by-value and pass-by-result, so is inout-mode. The actual parameter value is used to initialize the formal parameter, which is used as a local variable.
- b. At termination of the subprogram, the formal parameter is transmitted back to the actual parameter.

4. Pass-by-Reference

- a. Transmits an access path, is inout-mode.
- b. Is efficient in terms of space: no local copy need be made, though indirect addressing must be used.
- c. Problem: aliasing may occur, which hinders program readability and makes verification difficult. Examples:

```
void fun (int *first, int *second)
```

Used with the call:

```
fun (&list[ i], &list[ j])
```

or with a collision between parameters and nonlocal variables:

```
PROCEDURE bigsub;
VAR global: INTEGER;

    PROCEDURE smallsub (VAR local: INTEGER);
    BEGIN
        local = global * 5; { same as global = global * 5}
    END; {of smallsub}

BEGIN
    ...
    smallsub( global);
    ...
END; {of bigsub}
```

5. Pass-by-Name

- a. Does textual substitution for the formal parameter in all its occurrences in the subprogram. It is like a run-time macro substitution
- b. It is flexible, though difficult to understand
- c. Example in pseudo-Algol

```
PROCEDURE bigsub;
INTEGER global;
INTEGER ARRAY list [1:2];
    PROCEDURE sub(parameter);
    INTEGER parameter;
    BEGIN
        parameter := 3;
        global := global + 1;
        parameter := 5;
    END;
BEGIN
    list[1]:= 2; list[2]:= 2; global:= 1;
    sub( list[ global]);
end;
```

After execution, list[1] and list[2] have the values 3 and 5, respectively. This is an example of *late binding* which is used in OOP

C. Parameter-Passing Methods of the Major Languages

1. FORTRAN: inout-mode semantics; doesn't specify pass-by-reference or pass-by-value-result
2. ALGOL 60: pass-by-name, with pass-by-value as an option
3. C: pass-by-value. Pass-by-reference done by using pointers as parms.
 - a. Formal parameters can be typed as pointers to constants, where the actual parms. need not be constants. The actual parms. are then coerced to constants, allowing pass-by-reference efficiency with the semantics of pass-by-value
 - b. This is different from pointer constants, such as an array name
4. C++: includes the special pointer type *reference type*. These are implicitly dereferenced. For example:

```
void fcn (const int &a, int b, int &c) {... }
```
5. Constant parms. not exactly same as in-mode parms., since constant parms may not be locally redefined, but in-mode parms may (except in Ada).
6. Pascal & Modula-2: default is pass-by-value, with the keyword **VAR** used for pass-by-reference
7. Ada: has all three modes: in, out, and in out. E.g.

```
procedure Adder (  A : in out INTEGER;  
                  B : in  INTEGER;  
                  C : out  FLOAT; )
```

out-mode: can be assigned but not referenced.

in-mode: can be referenced, but not assigned

in out mode: can both reference and assign

D. Type-Checking Parameters

1. Used to catch typographical errors
2. Parameter type checking not required: FORTRAN 77
3. Parameter type checking is required: Pascal, Modula-2, FORTRAN 90, Ada
4. Original C: neither number or type of parameters were checked
5. ANSI C: parms may or may not be checked depending on formal parm. declaration
 - a. Not checked:

```
double sin (x)  
    double x;  
{ ... }
```

thus allowing

```
double value;  
int count;  
...  
value = sin(count);    //Runs, but shouldn't
```

- b. Are checked:

```
double sin (double x)    //The "prototype" method  
{ ... }
```

Here we can still have the call

```
value = sin(count)
```

since count is coerced into a double

6. C++: all parms. in prototype form, however type checking may be avoided by using ellipses:

```
double sin (...)  
{ ... }
```

which is then valid with any parameter list. In fact, the two modes may be mixed, as in:

```
printf (const char* ...)  
{ ... }
```

where printf must have at least a pointer to a constant character string, but everything else is optional. The number of additional parms is derived from the special symbols in the character string:

```
printf ("The sum is %d\n", sum);
```

E. Implementing Parameter-Passing Methods

1. Run-time stack is used for parameter communication [See overhead w/ descriptions below]
2. Pass-by-value: values copied onto stack, where those values become the formal parms
3. Pass-by-result: stack space allocated by caller, but values are stored there by the called subprogram and retrieved by the calling subprogram
4. Pass-by-value-result: combination of both of the above
5. Pass-by-reference: only address is passed on the stack. Formal parms use indirect addressing to retrieve value
6. Pass-by-name: a special parameterless subprogram is called each time one of these parameters is referenced. This subprogram (called a *thunk*) computes the address of the actual parameter and returns it. Compare to pass-by-reference:
 - a. pass-by-reference requires the overhead of indirect addressing
 - b. pass-by-name requires the overhead of a subprogram call
7. Problem in Ada: formal parms that are arrays or records can be implemented using *either* pass-by-value-result or pass-by-reference
 - a. This can create aliasing (global var. passed as a parameter, so subprog. can access global var. in two ways)
 - b. If subprog. terminates abnormally (via an exception), then the actual parm in pass-by-value-result will be unchanged, but it *will* be changed if pass-by-reference were used.

F. Multidimensional Arrays as Parameters

1. In C and C++, multidimensioned arrays are really "arrays of arrays," stored in row-major order. Thus the mapping function needs the number of columns, but not the number of rows. E.g.

```
void fcn (int matrix [][] [10])  
{ ... }  
void main()  
{  
    int table[5][10];  
    fcn( table);  
}
```

The problem with this is that a new function must be written for arrays of different sizes. To overcome this, a pointer to the array may be passed instead, along with the dimensions, such as:

```
void fcn (float *matrixPtr, int numRows, int numCols)  
{...}
```

where we can copy an element *x* into the [row][col] element by using:

```
*(matrixPtr + (row * numCols) + col) = x;
```

where we can increase the readability using a macro, such as:

```
#define matrixPtr(r,c) (*(matrixPtr + ((r) * numCols) + (c)))
```

and call it as:

```
matrixPtr(row,col) = x;
```

2. Pascal & Ada include size information with the type
3. Ada also allows an unconstrained array type as a formal parm [see overhead]

G. Design Considerations

1. When passing large data structures which are semantically pass-by-value, pass-by-reference may be more efficient (could use C++ constant reference parms)
2. This is why Ada allows formal parms that are arrays or records to be implemented using *either* pass-by-value-result or pass-by-reference

H. Examples of Parameter Passing

VI. Parameters that are Subprogram Names

- A. Original Pascal did not allow this. Consider example [overhead] in C to do bubble sort in ascending or descending order.
- B. Consider example in later versions of Pascal, where we do numerical integration to find the area under a curve. The function defining the curve may be different things.

```

PROCEDURE integrate (FUNCTION fun (x: REAL) : REAL;
                    lowerbd, upperbd: REAL;
                    VAR result: REAL);
...
VAR funval: REAL;
BEGIN
...
funval:= fun(lowerbd);
...
END;

```

Here we can see that we can do static type-checking of the formal parameter in integrate with the actual parameter

- C. What is the correct referencing environment for the passed subprogram? Three choices are:
 1. The environment inside the subprogram where the passed subprogram is used. (shallow binding)
 2. The environment where the passed subprogram is originally declared (deep binding)
 3. The environment of the caller where the subprogram is passed as the actual parm.
- D. Assuming the subroutine calls are legal, consider which referencing environment (and subsequent value of x) will be printed in the following program [overhead] when we call SUB2 from within SUB4.
 1. For choice 1 referencing environment is SUB4 and x would be 4. (shallow binding)
 2. For choice 2 referencing environment is SUB1 and x would be 1. (deep binding)
 3. For choice 3 referencing environment is SUB3 and x would be 3. (not used in practice)
- E. Deep binding is used for block-structured languages because of static binding of variables.

VII. Overloaded Subprograms: Programs with the same name in the same referencing environment

- A. Each overloaded subprogram must have a unique protocol
- B. E.g. in Ada an overloaded subprogram called SORT [overhead]
- C. The use of default parameters can lead to ambiguity, as in C++:

```

void fun (float b = 0.0);
void fun();
...
fun(); // matches with both declarations!

```

VIII. Generic Subprograms: also called polymorphic subprograms

- A. Parametric polymorphism: a subprogram that takes a generic parameter that is used to describe the types of the parameters of the subprogram
- B. In Ada: generic units. Example to sort elements of an array of any type [overhead] where at compile time an instantiation of the generic sort is made [overhead]
- C. In C++ generic functions are called template functions.

1. These are instantiated implicitly during a call or when its address is taken (with the & operator).

E.g. a template to find the maximum of two objects:

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

where the following code would do two instantiations:

```
int a, b, c;
char d, e, f;
...
c = max (a, b);
f = max (d, e);
```

2. The C++ version of the generic sort subprogram [see overhead]

- D. Both the examples in Ada and C++ above are examples of compile-time polymorphism. Smalltalk and C++ also support run-time instantiation according to the types of the actual parameters, which is known as dynamic polymorphism.

IX. Separate and Independent Compilation

- A. Recall that separate compilation became an issue in early FORTRAN since the computer would crash before long (> 500 lines) programs could be compiled.
- B. Separate compilation: units can be compiled at different times, though they are not independent of each other. E.g. the interfaces of Ada packages are maintained in a library that is accessible to the compiler
- C. Independent Compilation: program units can be compiled without information about any other program units. This has a detrimental effect on consistency and type checking between units.

X. Design Issues for Functions

A. Side Effects

- 1. Side effects are avoided if function parameters are in-mode
- 2. This is not the case in Pascal or C, as parameters may be pass-by-reference

B. Types of Return Values

- 1. Return types may not be structured types in Pascal. C return types may not be arrays or functions, though they can be handled by returning pointers to them.
- 2. C++ allows user defined types to be returned. Ada functions can return values of any type (first class)

XI. Accessing Nonlocal Environments

- A. nonlocal variables are visible within the subprogram, but not locally declared. Global variables are visible in all of the program units. (Note some nonlocal variables may not be global, as they are available only in some of the program units)
- B. Accessing nonlocal variables may be done through static or dynamic scoping (See Ch. 4). Both have their problems:
 - 1. Static scoping: program structure may be dictated by accessibility of procedures and variables, rather than well-engineered problem solutions
 - 2. Dynamic scoping:
 - a. all local variables from ancestors in the calling hierarchy are visible.
 - b. It is not possible to statically type check references to nonlocals, as their referencing environment changes depending on the call order.
- C. FORTRAN COMMON Blocks: shared storage may be interpreted two different ways. Problems analogous to that of unions [see overhead]
- D. External Declarations and Modules
 - 1. In both Ada & Modula-2, units can specify external modules to which access is required. In Modula-2 access may be further restricted to specific parts of the external modules
 - 2. In C external modules can have declarations which are included only where they are used

XII. User-Defined Overloaded Operators

- A. In Ada and C++ operators already in use may be overloaded to have new meanings
- B. Example overloading "*" to be the dot product of two INTEGER arrays, where a vector is defined to be an array of INTEGERS: (A C++ prototype)
int operator * (const vector &a, const vector &b, int len);
This can cause problems with readability, such as in the statement
C:= A * B;
Where understanding the statement requires keeping track of the operand types.