Chapter 8 - Subprograms

Fundamental Characteristics of Subprograms

1. A subprogram has a single entry point
2. The caller is suspended during execution of the called subprogram
3. Control always returns to the caller when the called subprogram's 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 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.
- A ***subprogram declaration*** provides the protocol, but not the body, of the subprogram. *Importance of declarations?*

Prototypes & Forward/External declarations?

Parameters

A subprogram can gain access to the data it is to process:

- through direct access to nonlocal variables
  - reduced reliability
- through parameter passing (*parameterized computation*)
  - more flexible
  - transmitting computations (functions) raher than data as parameters is possible.

A ***formal parameter*** is a dummy variable listed in the subprogram header and used in the subprogram. *Why dummy?*

An ***actual parameter*** represents a value or address used in the subprogram call statement.

Actual/Formal Parameter Correspondence:

1. ***Positional: C, C++, Java***
2. ***Keyword: ADA***
e.g. SORT(LIST => A, LENGTH => N);
Advantage: order is irrelevant
Disadvantage: user must know the formal parameter's names

Default Values: C++, F90, ADA
e.g., float exponent(float a, int exp=1)

Variable Number of Parameters
e.g., printf(...) in C.

Procedures and Functions

Procedures

- provide user-defined statements: e.g.*sort*.
- produce results in calling program unit by
  - by changing nonlocal but visible variables of caller
  - by changing parameters supplied by caller

Functions

- provide user-defined operators
  e.g., exponentiation operator: *power*(...)
- the value produced by a function's execution is returned to the calling code, effectively replacing the call itself.
- in C++: overloaded operators can be defined
- C and C++ have only functions. However, they can behave like procedures. *How?*

Design Issues for Subprograms

1. What parameter passing methods are provided?

2. Are parameter types checked?

3. Are parameter types in passed subprograms checked?

4. Are local variables static or dynamic?

5. Whether subprogram names be passed as parameters? If so what is the referencing environment of a passed subprogram?

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?

Local Referencing Environments

Local variables: variables that are declared inside subprograms.

- If local variables are stack-dynamic:
  Advantages:
  a. Support for recursion
  b. Storage for locals is shared among some subprograms
  Disadvantages:
  a. Allocation/deallocation time
  b. Indirect addressing
  c. Subprograms cannot be history sensitive
- Static locals are the opposite
  e.g., A pseudo-random number generator.

Language Examples:

1. FORTRAN 77 and 90 - most are static (therefore no recursion), but can have either (SAVE forces static.)

2. C - both (variables declared to be **static** are) (default is stack dynamic.)

3. Pascal, Modula-2, and Ada - dynamic only.

Parameters and Parameter Passing

Parameter passing methods are the ways in which parameters are transmitted to and/or from called subprograms.

Semantic Models of Parameter Passing:
in mode, out mode, inout mode

Conceptual Models of Transfer:
1. Physically move a value (callee ↔ caller)
2. Move an access path (an address)

Implementation Models of Parameter Passing:

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 C++ can do this. *How?*
- Accesses cost more (indirect addressing)

Disadvantages of physical move:
- Requires more storage
- Cost of the moves, especially if parameter is a long array.

2. Pass-by-result (*out* mode)
Local's 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 may be a *parameter collision* problem:
e.g.

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

Value of x in the caller depends on order of assignments at the return-not portable !
c. Time of evaluation of actual parameter addresses is implementation dependent:
- at the time of the call
- at the time of the return

e.g., list[index] where index changes within the subprogram.

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

4. Pass-by-reference (*inout* mode)
Pass an access path (an address)
Also called **pass-by-sharing**

Advantage: passing process is efficient, no extra space or time required.
Disadvantages:
a. Slower accesses, i.e., indirect.
b. Can allow aliasing:
i. *Actual parameter collisions:*
e.g.

```
void fun(int *a,  int *b);
  ...
  sub1(&x, &x);
```

ii. *Array element collisions:*
e.g. **sub1(a[i], a[j]); /* if i = j */**
Also, **sub2(a, &a[i]);**
iii. Collision between formals and globals

The root cause of the aliasing problem is: The called subprogram is provided wider access to nonlocals than is necessary, such as with static scoping.

Pass-by-value-result does not allow these aliases (but has other problems!)

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 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 contains a reference to a variable that is also accessible in the program: it is also like nothing else

Disadvantages of pass by name:
- Very inefficient references (slow)
- Too tricky; hard to read and understand

e.g.

```
procedure sub1(x: int; y: int);
begin
  x := 1;
  y := 2;
  x := 2;
  y := 3;
end;
. . .
sub1(i, a[i]);
```

e.g. (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);
```


Language Examples:

1. FORTRAN (always *inout* mode semantics)
- Before F77, pass-by-reference
- In F77 - scalar variables are often passed by value-result

2. ALGOL 60
- Pass-by-name is default; pass-by-value is optional

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 is optional

6. C++
- Like C, but also allows reference type actual parameters;
- the corresponding formal parameters can be pointers to constants, which provide the efficiency of pass-by-reference with in-mode semantics

7. Ada
- All three semantic modes are available
- If out, it cannot be referenced
- If in, it cannot be assigned

8. Java
- Like C, except references instead of pointers.

Type Checking Parameters
Now considered very important for reliability

e.g.
function sub1(float v) { . . . }
. . .
sub1(1) may provide unexpected results.

Language Examples:
- 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

e.g.
double sin (x)
double x;
{ . . . } **value=sin(count)** is legal!
avoids type checking!

double sin (double x)
{ . . . } **value=sin(count)** is legal!
is type-checked!

Implementing Parameter Passing Methods

ALGOL 60 and most of its descendants use the **run-time stack**
It is initialized and maintained by the **run-time system** which is a system program that manages the execution of programs.

- **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 called **thunks** evaluate the address of the parameter; called for each reference to the formal.
  - Very expensive, compared to reference or value-result

See Figure 8.2 in 3rd Ed.

Examples

C & pass-by-value:

```
void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
... swap(c, d);
```

Pascal and pass-by-value:

```
procedure swap1 (a, b: integer)
  temp : integer;
begin
  temp := a; a:=b; b:=temp
end;
```

C and *simulated* pass-by-reference:

```
void swap2(int* a, int* b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
...swap2(&c, &d)
...swap2(&i, &list[i]);
```


ADA and pass-by-value-result:

```
procedure swap3 (a : integer, b: integer)
  temp : integer;
begin
  temp := a;
  a:=b;
  b:=temp
end;
...swap3(c, d)
...swap3(i, list[i]);
```

C with aliasing: (i and a are aliases)

```
int i = 3;    /* global variable */
void fun (int a, int b) {
  i = b;
}
void main( ) {
  int list[10];
  list[i] = 5;
  fun(i, list[i]);
}
```

What happens if pass-by-value-result?
. . .pass-by-reference?

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.

- **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, 4th Ed., p.344, 3rd Ed.)
- **Pascal**
  - Not a problem (declared size is part of the array's type)
- **Pre-90 FORTRAN**
  - Formal parameter declarations for arrays must include passed parameters.

Design Considerations for Parameter Passing

1. Efficiency
2. One-way or two-way data transfer is desired.

These two are in conflict with one another!

Good programming => limited access to nonlocal 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.

See **Examples of Parameter Passing**.

Parameters that are Subprogram Names

Some situations can be conveniently handled if subprogram names can be sent as parameters to other subprograms.

e.g. A numerical integration subprogram that computes the area under the graph of a given function by sampling the function at a number of different points.

How it works?

- transmission of the subprogram code could be done by passing a single pointer.

Issues:

1. Are parameter types checked?
   If so, the description of the subprogram's parameters must be sent, along with the subprogram name.

- Early Pascal and FORTRAN 77 do not
- Later versions of Pascal, Modula-2, and FORTRAN 90 do
- C and C++ - pass pointers to functions; parameters can be type checked

2. We skip the other issues such as "correct referencing environment."
   In most statically scoped languages, it is that of the subprogram that declared it:s "Deep binding."

In Pascal:

```
procedure integrate(function fun (x: real):real;
                    lowerbd, upperbd: real;
                    var result: real);
. . .
var funval : real;
begin
. . .
funval := fun (lowerbd);
. . .
end;
```

In C:

```
void bubble(int *work,
        const int size,
        int (*compare) (int, int) )
{
   if ((*compare)(i, j))
      . . .
}
int ascending(const int, const int);
. . .
bubble(a, SIZE, ascending);
```


Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment.
- C++ has overloaded subprograms built-in, and users can write their own overloaded subprograms.
  - Every incarnation of an overloaded procedure must have a unique protocol.

e.g.

```
int square(int x) { return x * x; }
double square(double y) { return y * y; }
```

The following call will give a compilation error. Why?

```
void fun(float b = 0.0);
void fun( );
. . .
fun( );
```


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*.

Examples:

- C++ template functions

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

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
```


Separate and Independent Compilation

Essential in construction of large software systems:

- only the updates modules need to be recompiled during development or maintenance.
- *Linker* collects the newly compiled and previously compiled units.

Independent compilation is compilation of some of the units of a program separately from the rest of the program, <u>without the benefit of interface information.</u>

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.

Language Examples:

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

Functions

Design Issues:

1. Are side effects allowed?

Not desired, so parameters should always be *in* mode: Not possible in Pascal/C.

a. Two-way parameters (Ada does not allow)

b. Nonlocal reference (all allow)

2. What types of return values are allowed?

FORTRAN, Pascal - only simple types

C - any type except functions and arrays

C++ and **Java** - like C, but also allow classes to be returned.

Accessing Nonlocal Environments

Besides parameter passing, a subprogram can access variables from external 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.

Remember static and dynamic scoping!

Methods:

1. FORTRAN COMMON

- The only way in pre-90 FORTRANs to access nonlocal variables.
- Can be used to share data or share storage.

```
sub1:   REAL A(100)
        INTEGER B(250)
        COMMON /BLOCK1/ A,B

sub2:   REAL C(50), D(100)
        INTEGER E(200)
        COMMON /BLOCK1/ C, D, E
```

2. *Static scoping* - discussed in Chapter 4

3. *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)

4. *Dynamic Scope* - discussed in Chapter 4

User-Defined Overloaded Operators

Nearly all programming languages have *overloaded operators:* e.g., + in C.

Users can further overload operators in C++ (Not carried over into Java.)

Example (C++):

```
class String {
   private:
       char *sPtr;   // Pointer to start of string
       int length;   // string length
}
int String::operator==(const String &right)
{
   return strcmp(sPtr, right.sPtr)==0);
}
```

Are user-defined overloaded operators good or bad?

- too much overloading may hinder readability
- in a large project, different groups may overload the same operators differently.

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