Chapter 8 - Subprograms

Fundamental Characteristics of Subprograms
1. A subprogram has a single entry point.
2. A subprogram suspends execution of the calling program unit.
3. Control always returns to the caller when the called subprogram terminates.

Basic Definitions:
1. A subprogram definition is a description of the actions of the subprogram abstraction.
2. A subprogram call is an explicit request that the subprogram be executed.
3. A subprogram parameter profile consists of the names, kinds, and types of the parameters.
4. The protocol of a subprogram is its parameter profile plus, if it is a function, the return type.
5. A subprogram declaration provides the protocol, but not the body, of the subprogram. Importance of declarations?

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

Variable Number of Parameters
- Pascal, Modula-2, and Ada - dynamic only.
- If local variables are stack-dynamic:
- C - both (variables declared to be static are)
- FORTRAN 77 and 90 - most are static

Languages - Chapter 8

K.Dincer

Local Referencing Environments

Local variables: Variables that are declared inside subprograms.
- Local variables are stack-dynamic:
  - Support for recursion
  - Storage for locals is shared among subprograms

Disadvantages:
- Allocation/deallocation time
- Loss of information at destruction
- Subprograms cannot be history sensitive
- Static locals are the opposite
  - A pseudo-random number generator.

Language Examples:
1. Pascal, Modula-2, and Ada - dynamic only
2. C - both (variables declared to be static are)
3. Fortran 90 - most are static

Precedence and Functions

Procedures
- provide user-defined statements (e.g., sort)
- produce results in calling program unit by
  - changing local but non-visible variables of caller
  - changing parameters supplied by caller
Functions
- provide user-defined operators (e.g., concatenation operator: juxtapose)
- the value produced by a function's execution is returned to the calling code, effectively replacing the called value
- C and C++ have only functions. However, they can behave like procedures. Why?

Prototypes & Forward/External declarations?

Parameters
- A subprogram can gain access to the data it is to process:
  - through direct access to local variables
  - through parameter passing (parameterized procedure)
  - more flexible
  - transmits computations (functions) rather than data as parameters is possible.

A formal parameter is a dummy variable listed in the parameter profile of a subprogram and used in the subprogram header and/or call statement.

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

Actual/ Formal Parameter Correspondence:
1. Positional: C, C++, Java
   - e.g., SORT(LIST => A, LENGTH => N)
   - Advantage: order is irrelevant
   - Disadvantage: user must know the formal parameter's names

2. Named: Ada
   - e.g., NODummy(A => A, N => N);
   - Advantage: order is irrelevant
   - Disadvantage: user must know the formal parameter's names

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 at 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?

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 Model of Transfer:
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, too.
   - Accessors must ensure (indirect addressing)

2. Pass by address (out mode)

Implementation Models of Parameter Passing:
1. Pass by value (in mode)
   - Either by physical move or access path

   Disadvantages of access path:
   - Requires more storage
   - Cost of the moves, especially if parameter is in 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 sub(1: int, x: int);

3. Pass-by-value-result (inout mode)
   Physical move, both ways
   Also called pass-by-copy
   Disadvantages:
   a. Those of pass-by-result
   b. Those of pass-by-value

4. Pass-by-reference (inout mode)
   Pass an access path (an address)
   Also called pass-by-sharing
   Advantages:
   a. Faster access, i.e., indirect.
   a. Can allow sharing.
   Disadvantages:
   a. Slower access.
   b. Can allow sharing.
   i. Actual parameter collisions
      e.g. void fun(int *a, int *b);
   sub(1, x);
   ii. Array element collisions
      e.g. sub(a[i], a[j]); /* if i = j */
   Also, sub(a[i], b);
   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 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 difficult references (done)
   - Too tricky: hard to read and understand

Language Examples:
1. FORTRAN
   - Before F77, pass-by-reference
     – In F77, scalar variables are often
       passed by value-result

2. ALGOL 60
   - Pass-by-reference is default. pass-by-value
     result is optional

3. ALGOL W
   - Pass-by-value result

4. Pascal and Modula-2
   - Pass-by-reference is optional

5. C
   - Pass-by-reference is default

6. C++
   - Pass-by-reference is default
   - allows reference type
     actual parameters

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

8. Java
   - Like C, except references instead of pointers
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 manages the execution of programs.

- **Values**: copy it to the stack; references are indirect to the stack.
- **Result**: same
- **References**: regardless of form, put the address in the stack.
- **Names**: 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:
```c
void swap1(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
... swap(c, d);
```

Pascal and pass-by-value:
```pascal
procedure swap1(a, b: integer);
    var temp : integer;
    begin
        temp := a;
        a := b;
        b := temp;
    end;
... swap3(c, d);
```

C and simulated pass-by-reference:
```c
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:
```pascal
procedure swap3(a, b: integer);
    var temp : integer;
    begin
        temp := a;
        a := b;
        b := temp;
    end;
...swap3(c, d);
...swap3(i, list[i]);
```

C with aliasing:
```c
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.

- **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.
In Pascal:

```pascal
procedure integrate(function fun (x: real):real;
lowerbd, upperbd: real;
var result: real);

var funval : real;
begin
  funval := fun (lowerbd);
end;
```

In C:

```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.

```c
int square(int x)
{ return x * x; }

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

The following call will give a compilation error. Why?

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

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

Another Example:

```c++
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 generic_sort
}
```

Example use:

```c++
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, 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?
  2. What types of return values are allowed?

- Accessing Nonlocal Environments:
  - Includes parameter passing, a subprogram can access variables from external environments.

- Overloaded subprograms are those that are usable 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 non-local 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 BLOCK2/ C, D, E
   ```
2. **Static scoping** - discussed in Chapter 4
3. **External declarations**
   - Subprograms are not nested.
   - Globals are created by external declarations (they are only 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++)**

```c++
struct String {
    char *sPtr;
    int 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.