An Example: pass-by-value-result vs. pass-by-reference

<table>
<thead>
<tr>
<th></th>
<th>pass-by-value-result</th>
<th>pass-by-reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>(entry to p)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(after y := y + 1)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(at p's return)</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

Aliases can be created due to pass-by-reference

- Given void fun(int &first, int &second),
  - Actual parameter collisions
    - E.g., fun(total, total) makes first and second to be aliases
  - Array element collisions
    - E.g., fun(list[i], list[j]) can cause first and second to be aliases if i == j
  - Collisions between formals and globals
    - E.g., int* global;
      void main() { ... sub(global); ... }
      void sub(int* param) { ... }
    - Inside sub, param and global are aliases
Pass-by-Name

• Implement an inout-mode parameter transition method
• The body of a function is interpreted at call time after textually substituting the actual parameters into the function body
• The evaluation method is similar to C preprocessor macros

An Example in Algol

procedure double(x);
  real x;
begin
  x := x * 2;
end;
Therefore, double(C[j]) is interpreted as C[j] = C[j] * 2
Another Example

- Assume k is a global variable,
  procedure sub2(x: int; y: int; z: int);
  begin
    k := 1;
    y := x;
    k := 5;
    z := x;
  end;
- How is the function call sub2(k+1, j, i) interpreted?

Disadvantages of Pass-by-Name

- Very inefficient references
- Too tricky; hard to read and understand
Implementing Parameter-Passing Methods

• Most languages use the runtime stack to pass parameters
  – Pass-by-value
    • Values are copied into stack locations
  – Pass-by-result
    • Values assigned to the actual parameters are placed in the stack
  – Pass-by-value-result
    • A combination of pass-by-value and pass-by-result
  – Pass-by-reference
    • Parameter addresses are put in the stack

An Example

• Function header: void sub (int a, int b, int c, int d)
  – a: pass by value
  – b: pass by result
  – c: pass by value-result
  – d: pass by reference
• Function call: main() calls sub(w, x, y, z)
Design Considerations for Parameter Passing

- Efficiency
- Whether one-way or two-way data transfer is needed
One Software Engineering Principle

- Access by subroutine code to data outside the subroutine should be minimized
  - In-mode parameters are used whenever no data is returned to the caller
  - Out-mode parameters are used when no data is transferred to the callee but the subroutine must transmit data back to the caller
  - Inout-mode parameters are used only when data must move in both directions between the caller and callee

A practical consideration in conflict with the principle

- Pass-by-reference is the fastest way to pass structures of significant size
Parameters that are subroutines

- In some situations, subroutine names can be sent as parameters to other subroutines
- Only the transmission of computation is necessary, which could be done by passing a functional pointer

Two complications with subroutine parameters

- Are parameters type checked?
  - Early Pascal and FORTRAN 77 do not type check
  - Later versions of Pascal, Modula-2, and FORTRAN 90 do
  - C and C++ do
Two complications with subroutine parameters

• What referencing environment should be used for executing the passed subroutine?
  – The environment of the call statement that enacts the passed subroutine (shallow binding)
  – The environment of the definition of the subroutine (deep binding)
  – The environment of the call statement that passed it as an actual parameter (ad hoc binding)

An Example

function sub1() {
  var x;
  function sub2() {
    alert (x);
  };
  function sub3() {
    var x;
    x = 3;
    sub4(sub2);
  };
  function sub4(subx) {
    var x;
    x = 4;
    subx();
  };
  x = 1;
  sub3();
}

• For shallow binding, the referencing environment of sub2 is sub4
• For deep binding, the referencing environment of sub2 is sub1
• For ad hoc binding, the referencing environment of sub2 is sub3
What is the output of alert(x)?

- Shallow binding?
- Deep binding?
- Ad hoc binding?

Referencing Environment for Subroutine Parameters

- Deep binding and ad hoc binding can be the same when a subroutine is declared and passed by the same subroutine
- In reality, ad hoc binding has never been used
- Static-scoped languages usually use deep binding
- Dynamic-scoped languages usually use shallow binding
An Example

function Sent() {
    print(x);
};
function Receiver(func) {
    var x;
    x = 2;
};
function Sender() {
    var x;
    x = 1;
    Receiver(Sent)
};

• In static-scoped languages, Receiver is not always visible to Sent, so deep binding is natural

• In dynamic-scoped languages, it is natural for Sent to have access to variables in Receiver, so shallow binding is appropriate

Design Issues for Functions

• Are side effects allowed?
  – Ada requires in-mode parameters, and does not allow any side effect
  – Most languages support two-way parameters, and thus allow functions to cause side effects
Design Issues for Functions

• What types of values can be returned?
  – FORTRAN, Pascal, and Modula-2: only simple types
  – C: any type except functions and arrays
  – Ada: any type (but subroutines are not types)
  – JavaScript: functions can be returned
  – Python, Ruby and functional languages: methods are objects that can be treated as any other object

Overloaded Subroutine

• A subroutine that has the same name as another subroutine in the same referencing environment, but its number, order, or types of parameters must be different
  – E.g., void fun(float);
    void fun();
• C++ and Ada have overloaded subroutines built-in, and users can write their own overloaded subroutines
Generic Subroutine

- A generic or polymorphic subroutine takes parameters of different types on different activations

- An example in C++
  ```cpp
  template<class Type>
  Type max(Type first, Type second) {
      return first > second ? first : second;
  }
  int a, b, c;
  char d, e, f;
  ...
  c = max(a, b);
  f = max(d, e);
  ```

Generic Subroutine

- Overloaded subroutines provide a particular kind of polymorphism called ad hoc polymorphism
  - Overloaded subroutines need not behave similarly

- Parametric polymorphism is provided by a subroutine that takes generic parameters to describe the types of parameters

- Parametric polymorphic subroutines are often called generic subroutines
Coroutine

• A special kind of subroutine. Rather than the master-slave relationship, the caller and callee coroutines are on a more equal basis

• A coroutine is a subroutine that has multiple entry points, which are controlled by coroutines themselves

Coroutine

• The first execution of a coroutine begins at its beginning, but all subsequent executions often begin at points other than the beginning

• Therefore, the invocation of a coroutine is named a resume

• Typically, coroutines repeatedly resume each other, possibly forever

• Their executions interleave, but do not overlap
**An Example**

- The first time `co1` is resumed, its execution begins at the first statement, and executes down to `resume(co2)` (with the statement included).
- The next time `co1` is resumed, its execution begins at the first statement after `resume(co2)`.
- The third time `co1` is resumed, its execution begins at the first statement after `resume(co3)`.

```plaintext
sub co1() {
  ...
  resume(co2);
  ...
  resume(co3);
}
```

**Coroutine**

- The interleaved execution sequence is related to the way multiprogramming operating systems work:
  - Although there may be one processor, all of the executing programs in such a system appear to run concurrently while sharing the processor.
  - This is called **quasi-concurrency**.
- Coroutines provide quasi-concurrent execution of program units.
Reference