Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 23

- Control abstraction
- Parameter passing

Control Abstraction

- Subroutines (functions, procedures, …) support control abstraction
- Syntax issues…
- Implementation: calling sequences…
- In-line expansion…
- Parameters and parameter passing…

Parameters

- *Formal parameters*
  - Lisp: `(lambda (a b) (/ (+ a b)))`
  - C function:
    `float ave(float a, float b) { return (a+b)/2.0; }`
- *Actual parameters*
  - Lisp function arguments: `(ave x 10.0)`
  - C function arguments: `ave(x, 10.0)`
- *Operands* (of special forms or predefined operators)
  - Lisp special forms: `(if flag "yes" "no")`
  - C operators: `x > 0 && flag`
  - Operand handling often depends on the type of built-in operator, e.g. special forms and operators with short-circuit evaluation

What can be passed as parameter?

- first-class values (in all PLs)
  - Both simple and composite values of data types
- either variables or pointers to variables (in many PLs)
- either procedures or pointers to procedures (in some PLs).

- Depends also on the parameter passing *mode* and *mechanism*

Parameter Passing Modes and Mechanisms

- Parameter passing modes
  - In
  - In/out
  - Out
- Parameter passing mechanisms
  - Call by value (in)
  - Call by reference (in+out)
  - Call by result (out)
  - Call by value/result (in+out)
  - Call by name (in+out)
- Different mechanisms used by C, Fortran, Pascal, C++, Java, Ada (and Algol 60)

Parameter Passing in C

- *Call by value* parameter passing only
- Actual parameter is evaluated and its value assigned to the formal parameter
- A formal parameter in the function body behaves as a local variable
  - For example:

```
int fac(int n)
{ if (n < 0) n = 0;
  return n ? n*fac(n-1) : 1;
}
```

- Passing pointers allows the values of actuals to be modified
  - For example:

```
swap(int *a, int *b)
{ int t = *a; *a = *b; *b = t; }
```

  - A function call should explicitly pass pointers, e.g. `swap(&x, &y);`
- Arrays and pointers are exchangeable in C
  - An array is automatically passed as a pointer to the array
  - An array can be passed by value as field in a struct

Parameter Passing in Fortran

- *Call by reference* parameter passing only
- If the actual parameter is an l-value (e.g. a variable) its reference is passed to the subroutine
- If the actual parameter is an r-value (e.g. the value of an expression) it is assigned to a *hidden temporary variable* whose reference is passed to the subroutine
  - For example

```
SUBROUTINE SHIFT(A, B, C)
INTEGER A, B, C
A = B
B = C
END
```

  - For example, **`SHIFT(X, Y, 0)`** assigns **`Y`** to **`X`**, and **`Y`** is set to 0
  - For example, **`SHIFT(X, 2, 3)`** assigns 2 to **`X`** and the assignment to **`B`** in the subroutine has no effect, but in Fortran IV this was not handled correctly!

Parameter Passing in Pascal

- *Call by value* and *call by reference* parameter passing
- Call by value is similar to C
- Call by reference: indicated with **`var`** qualifier in argument list
- For example

```
procedure swap(var a:integer, var b:integer)
var t;
begin
   t := a;
   a := b;
   b := t
end
```

  where the **`var`** parameters **`a`** and **`b`** are passed by reference

- Programs can suffer from unnecessary data duplication overhead
  - When a big array is passed by value the entire array is copied!
  - Passing large arrays by reference avoids copy overhead
  - But no clear distinction between in-mode or in/out-mode any longer

Parameter Passing in C++

- *Call by value* and *call by reference* parameter passing
- Call by value is similar to C
- Call by reference: use reference (&) operator
- For example:
  ```
  swap(int &a, int &b)
  { int t = a; a = b; b = t; }
  ```
  where the reference parameters **a** and **b** are passed by reference
- Arrays are automatically passed by reference (like in C)
- Big objects should be passed by reference instead of by value
  - To protect data from modification when passed by reference, use **const**, i.e. make it an in-mode parameter
  - For example:
    ```
    store_record_in_file(const huge_record &r) { ... }
    ```
  - Compiler will prohibit modifications of object
- **const** parameters (e.g. pointers) are also supported in ANSI C

Parameter Passing by Sharing

- *Call by sharing*: parameter passing of variables in the *reference model*
  - Reference model: variables are references to (shared) values
  - Smalltalk, Lisp, ML, Clu, and Java (partly) adopt the reference model of variables
- The value of the variable is passed as actual argument, which in fact is a pointer to the (shared) value
  - Essentially this is pass by value of the variable!
- Java uses both pass by value and pass by sharing
  - Variables of primitive built-in types are passed by value
  - Class instances are passed by sharing

Parameter Passing in Ada

- *In-mode* parameters can be read but not written in the subroutine
  - Call by value and writes to the parameter are prohibited in the subroutine
- *Out-mode* parameters can be written but not read in the subroutine (Ada 95 allows read)
  - Call by result, which uses a local variable to which the writes are made
  - The resulting value is copied to the actual parameter to pass the value out when the subroutine returns
- *In-out-mode* parameters can be read and written in the subroutine
  - Call by value/result uses a local variable that is initialized by assigning the actual parameter's value to it
  - The resulting value is copied to the actual parameter to pass the value out when the subroutine returns
- *Call by value*, *call by result*, and *call by value/result* parameter passing implements in, out, and in/out parameters, respectively

Parameter Passing in Ada: Example

- For example

```
procedure shift(a:out integer,
                b:in out integer,
                c:in integer) is
begin
   a := b;
   b := c;
end shift;
```

- Here, **`a`** is passed **out**, **`b`** is passed **in** and **out**, and **`c`** is passed **in**
- The Ada compiler generates specific code for the example **`shift`** procedure for in, out, and in/out parameters to implement call by value (in), call by result (out), and call by value/result (in/out), which is similar to the following C function:

Parameter Passing in Ada (cont'd)

```
void shift(int *a, int *b, int c)
{ int tmpa, tmpb = *b, tmpc = c; // copy input values at begin
  tmpa = tmpb;
  tmpb = tmpc;                   // perform operations on temps
  *a = tmpa;
  *b = tmpb;                     // copy result values out before return
}
```

- Temps are initialized, operated on, and copied to out mode parameters
- This is more efficient than pass by reference, because it avoids repeated pointer indirection to access parameter values
- The Ada compiler may decide to use call by reference for passing non-scalars (e.g. records and arrays) for memory access optimization
- Okay for in-mode, because the parameter may not be written
- Okay for out and in-out modes, since the parameter is written anyway

Parameter Passing and Aliasing

- An *alias* is a variable or formal parameter that refers to the same value as another variable or formal parameter
  - Example variable aliases in C++:

```
int i, &j = i;  // j refers to i (is an alias for i)
i = 2;
j = 3;
cout << i;   // prints 3
```

  - Example parameter aliases in C++:

```
shift(int &a, int &b, const int &c)
{ a = b;
  b = c;
}
```

    The result of **`shift(x, y, x)`** is that **`x`** is set to **`y`** but **`y`** is unchanged
  - Example mixing of variable and parameter aliases in C++:

```
int sum = 0;
score(int &total, int val)
{ sum += val;
  total += val;
}
```

    The result of **`score(sum, 7)`** is that sum is incremented by 14

Parameter Passing and Aliasing (cont'd)

- Java adopts reference model of variables and call by sharing
  - in the reference model all assignments create aliases
- Ada forbids parameter aliases
  - Allows compiler to choose call by reference with the same effect as call by result
  - But most compilers don't check and the resulting program behavior is undefined

Parameter Passing in Algol 60

- ***Call by name*** parameter passing by default, also call by value
- Passes actual arguments such as expressions into the subroutine body for (re)evaluation (reevaluation done via code "thunks")
- Jensen's device utilizes call by name:

```
real procedure sum(expr, i, low, high);
  value low, high;          low and high are passed by value
  real expr;                expr and i are passed by name
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do
    rtn := rtn + expr;      the value of expr depends on the value of i
  sum := rtn                return value by assigning to function name
end sum
```

- **`y := sum(3*x*x-5*x+2, x, 1, 10)`** calculates

$$y = \sum_{x=1}^{10} 3x^2\text{-}5x\text{+}2$$

Macro Expansion

- C/C++ *macros* (also called *defines*) adopt a form of call by name
  - For example
    `#define max(a,b) ( (a)>(b) ? (a) : (b) )`
- *Macro expansion* is applied to the program source text and amounts to the substitution of the formal parameters with the actual parameters in the macro
  - For example **`max(n+1, m)`** is replaced by **`((n+1)>(m)?(n+1):(m))`**
    Note: formal parameters are often parenthesized to avoid syntax problems when expanding
- Similar to call by name, actual parameters are re-evaluated each time the formal parameter is used
  - Watch out for re-evaluation of function calls in actual parameters, for example **`max(somefunc(),0)`** results in the evaluation of **`somefunc()`** twice if it returns a value >0

Lazy evaluation / Call by need

- Some functional languages (Miranda, Haskell) realize ***lazy evaluation*** by using ***call by need*** parameter passing*:* an expression passed as argument is evaluated only if its value is needed.
- Unlike ***call by name,*** the argument is evaluated *only the first time*, using ***memoization***: the result is saved and further uses of the argument do not need to re-evaluate it
- Combined with *lazy data constructors*, this allows to construct potentially infinite data structures and call infinitely recursive functions without necessarily causing non-termination
- This also allows the programmer to define *control flow structures* as abstractions instead of primitives
- Lazy evaluation works fine with **purely functional** languages
- Side effects require that the programmer reason about the order that things happen, not predictable in lazy languages.

Parameter Passing Issues

- ***Call by name problem***: hard to write a "swap" routine that works:

```
procedure swap(a, b)
integer a, b, t;
begin
  t := a;
  a := b;
  b := t
end swap
```

- Consider swap(i, a[i]), which executes:

```
t := i
i := a[i]
a[i] := t
```

this changes i

assigns t to wrong array element

Parameter Passing Issue (cont'd)

- ***Call by value/result problem***: behaves differently compared to call by reference in the presence of aliases (that's why Ada forbids it)
- For example:

```
procedure shift(a:out integer,
                b:in out integer,
                c:in integer) is
begin
  a := b;
  b := c;
end shift;
```

- When **`shift(x,x,0)`** is called by reference the resulting value of **`x`** is 0
- When **`shift(x,x,0)`** is called by value/result the resulting value of **`x`** is either unchanged or 0 (because the order of copying out mode parameters is unspecified)

Conformant Arrays

- Some languages support *conformant arrays* (or *open arrays*)
  - Examples: Ada, Standard Pascal, Modula-2
- Pascal arrays are types with embedded constant array bounds
  - Arrays have fixed shape and size
  - Problem when for example sorting arrays of different sizes because sort procedure accepts one type of array with one size only
- Array parameters in Standard Pascal are conformant and array size is not fixed at compile-time
  - For example:
    ```
    function sum(A : array[low..high : integer] of real) : real
      ...
    ```
  - Function **`sum`** accepts real typed arrays and **`low`** and **`high`** act like formal parameters that are set to the lower and upper bound index of the actual array parameter
- C passes only pointers to arrays to functions and array size has to be determined using some other means (e.g. as another parameter)

Closures as Parameters

- Recall that a subroutine closure is a reference to a subroutine together with its referencing environment
- Standard Pascal, Ada 95, Modula-2+3 fully support passing of subroutines as closures
  - Standard Pascal example:

```
procedure apply_to_A(function f(n:integer) : integer;
                     var A : array [low..high : integer] of integer);
  var i : integer;
begin
  for i := low to high do A[i] := f(A[i])
end
```

- C/C++ supports function pointers
  - No need for reference environment, because no nested subroutines
  - Example:

```
void apply_to_A(int (*f)(int), int A[], int A_size)
{ int i;
  for (i = 0; i < A_size; i++)
    A[i] = f(A[i]);
}
```

Summary

parameter mode	representative languages	implementation mechanism	permissible operations	change to actual?	alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
`in, const`	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
`out`	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
`var, ref`	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
`in out`	Ada	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write*	yes*	yes*

Default Parameters

- Ada, C++, Common Lisp, and Fortran 90 support *default parameters*
- A default parameter is a formal parameter with a default value
- When the actual parameter value is omitted in a subroutine call, the user-specified default value is used
  - Example in C++:

```
void print_num(int n, int base = 10)
  ...
```

  - A call to `print_num(35)` uses default value 10 for base as if `print_num(35,10)` was called
  - Example in Ada:

```
procedure put(item  : in integer;
              width : in field := default_width;
              base  : in number_base := 10) is
  ...
```

  - A call to `put(35)` uses default values for the `width` and `base` parameters

Positional Versus Named Parameters

- *Positional parameters*: the order of formal and actual arguments is fixed
  - All programming languages adopt this natural convention
- *Named parameter*: (also called *keyword parameter*) explicitly binds the actual parameter to the formal parameter
  - Ada, Modula-3, Common Lisp, and Fortran 90
  - For example in Ada:
    ```
    put(item => 35, base => 8);
    ```
    this "assigns" 35 to item and 8 to base, which is the same as:
    ```
    put(base => 8, item => 35);
    ```
    and we can mix positional and name parameters as well:
    ```
    put(35, base => 8);
    ```
  - Pro: documentation of parameter purpose
  - Pro: allows default parameters anywhere in formal parameter list, whereas with positional parameters the use of default parameters is restricted to the last parameter(s) only, because the compiler cannot tell which parameter is optional in a subroutine invocation

Variable Argument Lists (*varargs*)

- C,C++, Common Lisp, Java allow defining subroutines that take a variable number of arguments
  - Example in C:

```
#include <stdarg.h>
int plus(int num, ...)
{ int sum;
  va_list args;                    // declare list of arguments
  va_start(args, num);             // initialize list of arguments
  for (int i=0; i<num; i++)
    sum += va_arg(args, int);      // get next argument (assumed to be int)
  va_end(args);                    // clean up list of arguments
  return sum;
}
```

- Function **`plus`** adds a set of integers, where the number of integers is the first parameter to the function: **`plus(4,3,2,1,4)`** is 10
  - Used in the **`printf`** and **`scanf`** text formatting functions in C
- Variable number of arguments in C and C++ is not type safe as parameter types are not checked
- In Common Lisp, one can write (+ 3 2 1 4) to add the integers

Java *varargs* (since 1.5)

- A vararg (last formal parameter, with ellipsis) is handled as an array of the specified base type
- The caller can provide a list of arguments or an array
- Iteration on the vararg with *enhanced for* makes the mechanism transparent to the user

```
public static int max(int... array) {
        int max = Integer.MIN_VALUE;
        for (int i = 0; i < array.length; i++)
            if (array[i] > max) max = array[i];
        return max;
}}
```

```
max();      // returns -2147483648
max(5);                 // returns 5
max(5,3);               // returns 5
max(5,3,8);             // returns 8
int [] arr = {1, 2, 3, 6, 7, 5};
max(arr);               // returns 7
```

```
public static int max(int... arg){
        int max = Integer.MIN_VALUE;
        for (int next : arg)
         if (next > max) max = next;
        return max;
    }
```

Function Returns

- Some programming languages allow a function to return any type of data structure, except maybe a subroutine (requires first-class subroutines)
  - Modula-3 and Ada allow a function to return a subroutine as a closure
  - C and C++ allow functions to return pointers to functions (no closures)
- Some languages use special variable to hold function return value
  - Example in Pascal:

```
function max(a : integer; b : integer) : integer;
begin
  if a>b then max := a else max := b
end
```

  - There is no return statement, instead the function returns with the value of **`max`** when it reaches the end
  - The subroutine frame reserves a slot for the return value anyway, so these languages make the slot explicitly available

Function Returns (cont'd)

- Ada, C, C++, Java, and other more modern languages typically use an explicit return statement to return a value from a function
  - Example in C:

```
int max(int a, int b)
{ if (a>b) return a;
  else return b;
}
```

- Programmers may need a temporary variable for incremental operations on the return value
  - For example:

```
int fac(int n)
{ int ret = 1;
  for (i = 2; i <= n; i++)
    ret *= i;
  return ret;
}
```