

Parameter Passing

In this set of notes you will learn about:

- Parameter passing modes
 - Call by value
 - Call by reference
 - Call by sharing
 - Call by result
 - Call by value/result
 - Call by name
- Subroutine closures as parameters
- Special-purpose parameters
 - Conformance arrays
 - Default and optional parameters
 - Positional versus named parameters
 - Variable number of parameters
- Function returns

Note: These notes cover Section 8.3 of the textbook

Parameter Passing

- *Formal parameters*: the parameters that appear in the declaration of a subroutine
 - For example, `int n` in `fac`'s declaration written in C:


```
int fac(int n)
{ return n ? n*fac(n-1) : 1; }
```
- *Actual parameters*: the variables and expressions that are passed to a subroutine in a subroutine call
 - For example, `n-1` in the recursive call to `fac` in the example above
- *Arguments* can also be passed to built-in operations, but the way arguments are handled and evaluated often depends on the built-in operator
 - For example, `n ? n*fac(n-1) : 1` is a conditional expression that evaluates and returns `n*fac(n-1)` if `n` is nonzero (in C/C++), otherwise `1` is returned
 - Syntax of built-in operations often suggests that they are special, but not always, e.g. Lisp and Smalltalk have a uniform syntax for built-in operations and user-defined subroutines

Parameter Passing Modes in C

- *Call by value* parameter passing only: actual parameter is evaluated and its *value* is assigned to the formal parameter of the function being called
- A formal parameter behaves like a local variable and can even be modified in the function without affecting the actual parameter
 - For example


```
int fac(int n)
{ if (n < 0) n = 1;
  return n ? n*fac(n-1) : 1;
}
```
- Objects can be modified in a function by passing pointers to the object to the function
 - For example


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

 where `a` and `b` are integer pointer formal parameters, and `*a` and `*b` in the function body *dereference* the pointers to access the integers
 - A function call should explicitly pass pointers to integers, e.g. `swap(&x, &y)`, where `x` and `y` are integer variables and `&x` and `&y` are the addresses of their values
- Arrays and pointers are interchangeable in C: an array is automatically passed as a pointer to the array

Parameter Passing Modes 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. an expression, it is evaluated and assigned to an invisible temporary variable whose reference is passed
- For example


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

 - When called with `SHIFT(X, Y, 0)` this results in `Y` being assigned to `X`, and `Y` is set to `0`
 - When called with `SHIFT(X, 2, 3)` this results in `2` being assigned to `X`

Parameter Passing Modes in Pascal

- Call by value and call by reference parameter passing
- Call by value is similar to C
- Call by reference is indicated by **var** parameters
 - For example

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

where the **var** formal parameters **a** and **b** are passed by reference (**var t** declares a local variable)
- Programs can suffer from unnecessary data duplication overhead
 - When a big array is passed by value it means that the entire array is copied
 - Therefore, passing arrays by reference instead of by value is not a bad idea to enhance efficiency
 - Do this *unless* the array is modified in the subroutine, because call by value will not affect actual parameter but call by reference does, which can lead to buggy code

Parameter Passing Modes in C++

- Call by value and call by reference parameter passing
- Call by value is similar to C
- Call by reference is indicated by using **&** for formal parameters
 - For example

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

where the formal parameters **a** and **b** are passed by reference, e.g. `swap(x, y)` exchanges integer variables **x** and **y**
- Large objects should be passed by reference instead of by value to increase efficiency
- Arrays are automatically passed by reference (like in C)
- To avoid objects to be inadvertently modified when passed by reference, **const** parameters can be used
 - For example

```
store_record_in_file(const huge_record &r)
{ ... }
```
 - Compiler will prohibit modifications of object in function
- **const** parameters are also supported in ANSI C

Parameter Passing Modes in Languages With Reference Model of Variables

- Smalltalk, Lisp, ML, Clu, and Java (partly) adopt *reference model of variables*
- Every variable contains the *memory address* of the variable's value
- Parameter passing of variables is *call by sharing* in which the address of the value is passed to a subroutine
 - This is implemented similar to call by value: the content of the variable that is passed is an address to the variable's value
- For expressions, *call by sharing* passes the value of the expression

Parameter Passing Modes in Java

- Call by value and call by reference/sharing parameter passing
- Java adopts both *value* and *reference models of variables*
 - Variables of built-in types are passed by value
 - Class instances are passed by sharing
 - To pass a variable of built-in type by reference, the **&** can be used for formal parameters (like in C++)

Parameter Passing Modes in Ada

- *Call by value, call by result, and call by value/result* parameter passing
- Indicated by Ada's **in** (by value), **out** (by result), and **in out** (by value/result) modes for formal parameters
 - For example

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

where **a** is passed out, **b** is passed in and out, and **c** is passed in
- **in** mode parameters can be read but not written in the subroutine
 - *Call by value*, but 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* uses a local variable to which the writes are made and the resulting value is copied to the actual parameter 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 and when the subroutine returns the variable's value is copied back to the actual parameter

Parameter Passing Modes in Ada (cont'd)

- The Ada compiler generates code for the example Ada `shift` procedure to implement call by value, call by result, and call by value/result with a structure that is somewhat similar to the following ANSI C function

```
void shift(int *a, int *b, const int c)
{ int tmpa, tmpb = *b, tmpc = c; // copy input values
  before start
  tmpa = tmpb; tmpb = tmpc;
  *a = tmpa; *b = tmpb; // copy result values
  before return
}
```
- That is, local variables are initialized with **in** mode parameters, operated on, and copied to **out** mode parameters
- This is more efficient for simple types, because it avoids repeated pointer indirection necessary to access variables in memory that are passed by reference
- The Ada compiler may decide to use *call by reference* for passing non-scalars (e.g. records and arrays) to optimize the program
 - This works for **in** mode, because the parameter cannot be written (like using `const` in C++ with reference parameters)
 - This works for **out** and **in out** modes, because the parameter is written anyway

Parameter Aliasing Problems

- An *alias* is a variable or formal parameter that refers to the same value location 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, 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 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
- Java adopts reference model of variables (call by sharing)
 - Watch out for aliases as problems are hard to correct
- Ada forbids parameter aliases
 - Allows compiler to use call by reference with the same effect as call by result
 - But not checked by compiler and resulting program behavior is undefined

Call by Name Parameter Passing

- C/C++ macros (also called **defines**) adopt *call by name*
 - For example

```
#define max(a,b) ( (a)>(b) ? (a) : (b) )
```
 - A "call" to the macro replaces the macro by the body of the macro (called *macro expansion*), for example `max(n+1, m)` is replaced by `((n+1)>(m)?(n+1):(m))` in the program text
 - Macro expansion is applied to the program source text and amounts to the substitution of the formal parameters with the actual parameter expressions
 - Formal parameters are often parenthesized to avoid syntax problems after expansion, for example `max(c?0:1, b)` gives `((c?0:1)>(b)?(c?0:1):(b))` in which `(c?0:1)>(b)` would have had a different meaning without parenthesis
- Call by name parameter passing reevaluates actual parameter expression each time the formal parameter is read
 - Watch out for reevaluation of function calls in actual parameters, for example

```
max(somefunc(), 0)
```

results in the evaluation of `somefunc()` twice if it returns a value `>0`

Call by Name Parameter Passing in Algol 60

- Algol 60 adopts call by name parameter passing by default (and also supports call by value)
- A powerful use of call by name is *Jensen's device* in which an expression containing a variable is passed to a subroutine with the variable

```
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
```

- To calculate

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

we can simply write `y := sum(3*x*x-5*x+2, x, 1, 10);` in which the `sum` function sums $3*x*x-5*x+2$ for variable `x` ranging from 1 to 10

Parameter Passing Problems

- Call by name problem
 - Cannot write a swap routine that always 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]` *this changes i*
`a[i] := t` *assigns t to wrong array element*
- Call by value/result problem
 - Behaves differently compared to call by reference in the presence of aliases

```
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 undefined

Conformant Array Parameters

- Some languages support *conformant array* (or *open array*) parameters, e.g. Ada, Standard Pascal, Modula-2, and C
- Pascal arrays have compile-time shape and size
 - Problem e.g. when required to sort 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 integer) : real
  ...
```

Function `sum` accepts real typed arrays and `low` and `high` act like formal parameters that get the lower and upper bound index of the actual array parameter at run time
- C passes only pointers to arrays to functions and array size has to be determined using some other means (e.g. as a function 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++ support *pointers* to subroutines
 - No need for reference environment, because nested subroutines are not allowed
 - 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]);
}
```

The `int (*f)(int)` is a formal parameter that is a pointer to a function `f` from `int` to `int`

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

- Parameters are *positional* when the first actual parameter corresponds to the first formal parameter, the second actual to the second formal, etc.
 - All programming languages adopt this natural convention
- Ada, Modula-3, Common Lisp, and Fortran 90 also support *named parameters*
- A named parameter (also called *keyword parameter*) names the formal parameter in a subroutine call
 - 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);
```
- Advantages:
 - Documentation of parameter purpose
 - Allows default parameters anywhere in formal parameter list: with positional parameters, the use of default parameters is often restricted to the last parameters only, because the compiler cannot always tell which parameter is optional in a subroutine call

Variable Number of Arguments

- C, C++, and Common Lisp are unusual in that they allow defining subroutines that take a variable number of arguments
 - Example in C/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
  // (must be int)
  va_end(args);         // clean up list of arguments
  return sum;
}
```

Function `plus` adds a bunch of integers together, where the number of arguments is the first parameter to the function, e.g. `plus(4,3,2,1,4)` returns 10
 - Used in the `printf` and `scanf` text formatting functions in C
- Using 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

Function Returns

- Most languages allow a function to return any type of data structure, except a subroutine
- Modula-3 and Ada allow a function to return a subroutine implemented as a closure
- C and C++ allow functions to return pointers to functions (no closures)
- Earlier 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 `end`
- 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; }
```
 - May require a temporary variable for incremental operations:

```
int fac(int n)
{ int rtn=1;
  for (i=2; i<=n; i++) rtn*=i;
  return rtn;
}
```
 - `return` statement is more flexible, but wastes time copying the temporary variable value into return slot of subroutine frame on the stack