Chapter 8
(8.1 – 8.4)

Control Abstraction:
Subroutines and parameters

Control Abstraction

Programmer defined control structures

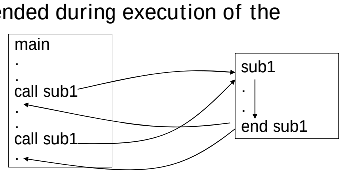
- Subprograms
  - Procedures
  - Functions
- Coroutines
- Exception handlers
- Processes

Subprograms

Issues related to subprograms

How is control transferred to & from the subprogram?

- Caller explicitly names the subprogram unit
- Subprogram implicitly transfers control back to caller
- Caller is suspended during execution of the subprogram



Subprograms

Issues related to subprograms

How is data transferred in & out of the subprogram?

- Parameters
- Global variables/shared data
- Function return values

Terminology

- *Formal parameter* – Used in subprogram definition
- *Actual parameter (argument)* – Used in subprogram call

Correspondence of: Formal ⇔ Actual

- Positional
- Keyword

Terminology

Subprogram definition describes

- the interface to the subprogram
- the actions of the subprogram

Subprogram header contains

- its name
- list of parameters
- return type

Subprogram call is an explicit request for the subprogram to be executed

Passing Parameters

Call By Value

- A copy of the actual parameter is assigned to the formal parameter at entry
- Formal parameter behaves as a local variable
  - Algol-68, Pascal, PL/1, Lisp, C++, Java (primitives)
  - Ada IN parameters are a variation which uses call by *constant value*
    - Statements that change formal parameter are not allowed

Passing Parameters

Call By Result

- Formal parameter behaves like a local variable during subprogram execution
- Upon return, the value of the formal parameter is copied into the actual parameter
  - Ada OUT parameters

Passing Parameters

Call By Value-Result

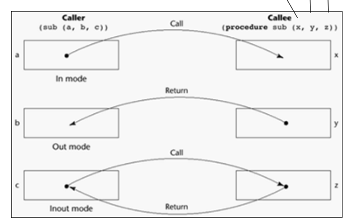
- Combines the semantics of Value and Result parameter passage
  - Ada IN OUT parameters

IN-OUT

OUT

IN

Comparison of in, out, in-out



Passing Parameters

Call By Reference

- Formal parameter is an *alias* for the actual parameter (stores the address of the argument)
- Access to the formal parameter involves dereferencing
  - FORTRAN, Pascal, PL/1, Algol-68, C (arrays), C++
  - Ada compilers *may* use pass by reference for OUT parameters that are structures (arrays, etc.)
- Java object parameters are references. The reference cannot be changed, but the contents of the object being referenced can. (**Call by Sharing**)

Passing Parameters

Call By Name

- Behaves as if the name of the actual parameter is substituted in place of the formal parameter
- This is an example of very late binding between formal and actual parameters
- Very flexible, but not efficient or readable
  - Algol-68

Passing Parameters

Notes

- All arguments, except those passed by value, must be bound to memory cells
- Generally arguments are bound to formal parameters at subprogram entry
- The amount of type checking, matching of parameters, etc. varies with different languages
- Call by reference is not necessarily more efficient than call by value-result

Example

Subprogram

```
void sub (int x, int y)   {
  x++; y++;
  print(i, x, y);
}
```

Main program

```
int main ( )   {
      int i = 0; a[ ] = {5, 10};
      sub (i, a[i]);
      print(i, a[0], a[1]);
      i = 1;
      sub(i, i);
      print(i);
}
```

Trace using call by value, value result, reference, and name

Parameter Passing

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*

Figure 8.3 Parameter passing modes.

C/C++ Parametes

Parameters passed by value in C

Call by reference can be simulated with pointers

```
void proc(int* x, int y){*x = *x+y } …
proc(&a,b);
```

Directly passed by reference in C++

```
void proc(int& x, int y) {x = x + y }
proc(a,b);
```

Default Parameters

Ada

```
procedure Proc(x : integer; y := 0 : integer; z := 5 : integer )

Proc(10, 20, 30);
Proc(10, 20);   -- z gets 5 by default
Proc(10);   -- y gets 0 and z gets 5
```

C++ uses a similar approach

So does Ruby, but without types declared

Java Varargs

Java 1.5+ supports variable length parameter lists:

```
class VarGreeter {
    public static void printGreeting(String... names) {
       for (String n : names) {
          System.out.println("Hello " + n + ". ");
       }
    }

    public static void main(String[] args) {
      printGreeting("Phil", "Brian", "Ashley", "Mark");
    }
  }
```

Ruby also supports this: `def proc(*arg)`

Perl Parameters

Passed implicitly as a list (array)

```
sub p {
  $x = shift;
  $y = shift;
  print "$x  $y\n";
}

p(99, 88);
```

Subprograms as Parameters

- Powerful feature supported by some languages
- Can be as simple as passing a pointer to the beginning of the subprogram
- But this will not allow checking parameters of the passed subprogram
- Ruby closures and blocks are examples of passing subprograms

Subprograms as Parameters

The MODULA-2 solution

```
(* Declare a type for procedures with one integer
parameter passed by reference *)

   TYPE ProcType = PROCEDURE( VAR INTEGER);

   PROCEDURE P( X : INTEGER; P1 : ProcType);

   P(10, ReadInt);
   P(5, INC);
```

ReadInt and INC are both procedures with one INTEGER parameter passed by reference

Simple Subprogram Implemention

Call Semantics (prologue):

1. Save the execution status of the caller
2. Carry out the parameter-passing process
3. Pass the return address to the callee
4. Transfer control to the callee

Simple Subprogram Implemention

Return Semantics (epilogue):

1. If call-by-value-result parameters are used, move the current values of those parameters to their corresponding actual parameters
2. If it is a function, move the functional value to a place the caller can get it
3. Restore the execution status of the caller
4. Transfer control back to the caller

Simple Subprogram Implemention

Required Storage

Status information of the caller, parameters, return address, and functional value (if it is a function)

Implementing Subprograms with Stack-Dynamic Local Variables

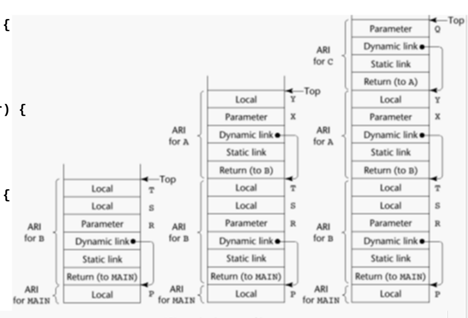
The activation record format is static, but its size may be dynamic

The **dynamic link** points to the top of an instance of the activation record of the caller

An activation record instance is dynamically created when a subprogram is called

Stack For a C Program

```
void A(int x) {
   int y;
   ...
   C(y);
   ...
}
void B(float r) {
   int s, t;
   ...
   A(s);
   ...
}
void C(int q) {
   ...
}
void main() {
   float p;
   ...
   B(p);
   ...
}
```



Generics

- Provide parameterized types
- Source code of subprogram is reused
- Compiler creates copies of subprograms for each type used (static binding)
  - Ada GENERIC, C++ template
- Java guarantees that all instances share same run time code
  - Everything is Object, compiler inserts type casts

Ada Generics

```
generic
  type T is private;
  type VECTOR is array (integer range <>) of T;
  procedure GENERIC_SORT(A: in out VECTOR);
  procedure GENERIC_SORT(A: in out VECTOR) is
    TEMP : T;
    begin
    for INDEX_1 in A'FIRST..PRED(A'LAST) loop
      for INDEX_2 in SUCC(INDEX_1)..A'LAST loop
        if A(INDEX_1) > A(INDEX_2) then
          TEMP := A(INDEX_1);
          A(INDEX_1) := A(INDEX_2);
          A(INDEX_2) := TEMP;
        end if;
      end loop;
    end loop
  end GENERIC_SORT;
```

```
type INT_ARRAY is array (1..50) of INTEGER;
procedure INTEGER_SORT is
            new GENERIC_SORT(T => INTEGER, VECTOR => INT_ARRAY);
```

Parameters are types

C++ Generics (Templates)

```
template <class T>
void sort(T list[], int len) {
  T temp;
  for (int i = 0; i < len - 1; i++)
    for (int j = i+1; j < len; j++) {
      if (list[i] > list[j]) {
        temp = list [i];
        list[i] = list[j];
        list[j] = temp;
    }
}
```

```
float flt_list[100];
...
sort(flt_list, 100);
```

Java Generics

```
public static <T extends Comparable<T>>
                    void sort(T[] list) {
  T temp;
  for (int i= 0; i < list.length - 1; i++)
    for (int j = i+1; j < list.length; j++) {
      if (list[i].compareTo(list[j]) > 0) {
        temp = list[i];
        list[i] = list[j];
        list[j] = temp;
      }
    }
}
```

```
  Integer [] a = {3, 6, 8, 2, 5};
  sort(a);
```

Java Generics

A common use of generics in Java (and other languages) is with Collections:

```
ArrayList<Integer> list =
                    new ArrayList<Integer>();
list.add(0, 42);
int val = list.get(0);
…
for (Integer i : list) { ... }
```

Autoboxing/ unboxing of primitive types