Chapter 4 – Abstractions in programs and Information Exchange

Introduction to Programming Languages

First Edition, 2013

Author: Arvind Bansal

© Chapman Hall / CRC Press
ISBN: 978-146-6565142

Topics Covered

- Introduction
- Data abstractions
- Control abstractions
- Information exchange
- Parameter passing
- Side effects
- Exception handling
- Nondeterministic computations
- Program as data
- Software reuse
- Case study
- Summary

Introduction

- A program manipulates structured data in a well mannered way: sequentially or concurrently
    - Program execution maintains causality.
    - The final state holds the result.
- Program can be written at multiple levels
    - Machine level; Assembly level; Procedural; Declarative; Event based
    - High level programming supports more abstractions and maintainability
- Types of abstractions: data and control
    - Data abstractions are declared
    - Control abstractions are in program-body
    - In certain class of languages, the difference between program and data becomes fuzzy: a program can be created as data and then transformed
    - A meta program may treat other programs as data
    - In modular languages, program and data are grouped in package
- Encapsulation provides natural visibility boundary
    - Information can be hidden using encapsulation.
    - Import-export mechanism exchanges information between modules

---

Data Abstractions

- Structured information based upon common attributes
    - Can be *specific* or *generic.*
    - *Generic* data-entities can hold different types of information
- Classification of data entities
    - **Single entity** is an atom or literal that can not be split further.
    - **Composite data-entity** groups different types of elements
    - **Extensible data entity** can be extended dynamically
    - **Collection** is a bag of data-entities sharing some common attribute that is being accessed or processed
    - **Aggregation** allows the same repeated operation on different elements. Individual data elements could be single, composite, collection or extensible
- A data entity is visible and manipulated within its scope
- Ways to access data entities
    - Use a name. Multiple names referring to the same entity is called *aliasing*
    - The extent of distance from the point of declaration

Types of Data-abstractions - I

- Single data-entity
  - integer, real, atom, character, bit, byte, semaphore
  - Strings can be treated as both single-entity as well as aggregation
- Aggregation : composite, collection, extensible
- Composite data-entity is modeled as a tuple
  - Each element of the composite data-entity can be any data abstraction: single or aggregate
  - Tuples can be named or unnamed. Most languages use named tuple as a template for declaring different data-entities using the same template
  - Recursive definition of tuple such as linked-list is used to model an extensible data-abstraction as described later

List = (info, List) or nil

Tree = (tree, info, tree) or null tree

Types of Data Abstractions - II

- Collection: an indexible bag of data-abstractions
  - A bag can have more than one element having the same value
  - All elements in the collection have same data-abstraction
  - Collection can be modeled as: 1) an indexible sequence; or 2) a content addressable bag of (key, value) pairs
  - Indexible sequences are modeled as arrays or vectors
  - Arrays can be static or dynamic. Dynamic arrays and vectors are extensible
- Extensible: recursive and dynamic tuple
  - Extensible data structure have indefinite size, and need dynamic allocation due to indefinite size
  - Implemented using pointers as pointers can point to new memory locations at runtime
  - Vectors, trees, and vectors all use pointers for extension

---

Example of Data Abstraction - I

Data abstraction: class

Abstraction-type: *tuple*

Attribute-size: 6

Begin attribute-description

attribute1: *single-entity* course-number

attribute2: *single-entity* course-name

attribute3: *single-entity* instructor

attribute4: *bag of* student

attribute5: *tuple* location

attribute6: *tuple* time

End attribute-description

End data abstraction

Example of Data Abstraction - II

Data-abstraction: student

Abstraction-type: *tuple*

Attribute-size: 4

Begin attribute-description

attribute1: *single-entity* student-id

attribute2: *single-entity* student-name

attribute3: *single-entity* department-name

attribute4: *single-entity* years-in-college

End attribute-description

End data-abstraction

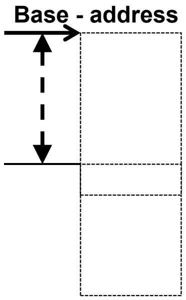
Example of Data Abstraction - III

```
Data-abstraction:  location
Abstraction-type: tuple
          Begin attribute-description
                    attribute1: single-entity  building
                    attribute2: single-entity   room
          End attribute-description
End data-abstraction

Data-abstraction:  time
Abstraction-type: tuple
          Begin attribute-description
                    attribute1: single-entity  start-time
                    attribute2: single-entity   duration
          End attribute-description
End data-abstraction
```

Accessing Data-entities

- In program individual fields are written as
  - *<data-entity>*.*<field-name>*
- Composite data-entities use offset method to retrieve fields in low level implementation
  - Code generator calculates the offset of each individual field, and adds the offset to the base address
- Collection of data-entities uses the index and individual size of data-entity to compute the address of the ith data-element
  - Address of the ith data-element =
    base-address + (I – 1) * size of(data-element)
- Tree based representation uses key-comparison
- In hash based implementation, hash function is used to find the index of the element



Interconnected Data-entities

- Interconnected data-entities relate multiple data-entities using some relationship
  - Relationship could be between entities or between attributes or between attribute of one entity and other entity that defines the attribute
- Interconnected data-entities can be modeled as a graph
  - Data-entities are modeled as nodes in the graph
  - Relationships are modeled as edges in the graph
  - Interconnected network is called semantic network in Artificial Intelligence
- In Lisp, each entity is modeled as a frame
  - Frame is a pair of the form (entity-name, property-list)
  - Property-list is {(property$_1$, value$_1$), **…,** (property$_N$, value$_N$)})
  - Where (property, value) could be (attribute, value) or (relationship, destination-entity) or (property, procedure to compute value)

Persistence

- Two major properties
  - Visibility across programs
  - Effect beyond life-time of a program / programming module
- Example of persistence
  - Writing in a file
  - Writing in a database
- Global Transient data-entities
  - Visbility across program modules within a program
  - Effect limited to the life time of a program
- Example of Global transient data entities
  - Blackboards: a tuple space of (key, value) pairs

Files and Streams

- File is a persistent sequence of data-entities that is kept on a secondary storage
  - Files can be very large, and may not fit in memory space of a computer
  - Only part of a file may be used in a memory space at a time
- Stream is an active subsequence of a file that is in memory and is being processed
  - Streams are necessary because files can be very large
  - Allows resource reuse avoiding wastage of memory space
  - Allows concurrent processing of subsequent data being brought in
- Stream is used in
  - Processing large files, multimedia processing, and Internet based processing of data

---

Declarations and Environment

- Environment is mapping of identifiers → l-value (in the case of variables) or identifiers → r-value (as in the case of constants)
- Declaration changes the computational state (σ^E, σ^S, σ^D) by changing the environment σ^E
- Declaration can be: sequential or parallel
  - Sequential declaration D_1**;** D_2 updates the environment one declaration at a time: D_1**;** D_2 $(\sigma^E) = D_2(D_1(\sigma^E)) = D_2(\sigma'^E)$ **where** $\sigma'^E = D_1(\sigma^E)$
  - Parallel declaration D1 || D2 updates the original environment simultaneously and independently for both the declarations.
- **Examples from functional language Scheme**
  - Sequential declaration: (let* ((X 4) (Y (+ X 4))) will make the environment σ^E as $\sigma^E \oplus \{X \mapsto 4\} \oplus \{Y \mapsto 8\}$.
  - Parallel declaration: $\sigma^E = \{x \mapsto 6\}$

    (let ((X 4) (Y (+ X 4))) will make new environment as

    $\{x \mapsto 4, Y \mapsto 10\}$

Control Abstractions

- Types of control abstractions
  - Constructors – create a new data-entity
  - Mutators – destructively update the value of a data-entity
  - Selectors – retrieve individual fields of an aggregate
  - Conditionals – picks up an alternative sequence of statement based upon the evaluation of a condition
  - Iterations – Repeated execute a set of statements until the final condition is satisfied
  - Iterators – operate repeatedly the same operation on a collection of data elements
  - Evaluators – instructions that evaluate an expression
  - Sequencers – jump statements that take control to some label
  - Invocations – instructions that call another function and procedures

---

Continuation

- Continuation is the actual sequence of statements that are executed at runtime in a program
  - Continuation are difficult to predict at compile-time in the presence of sequencers and conditional statements
  - Continuation of control abstractions involving condition evaluation and jump statements is decided at run time
  - Continuation of the iterative statements involves completely unfolding the statements in the loop
- Example

```
x = 4; z = 6;  goto L;
z = 8;
L:  y = 5;
while (z > 4)  { x = y + 5;  z = z – 1; }
```

- Continuation of x = 4:
  - {z = 6; y = 5; **if** z > 4 **then exit**; x = y + 5; z = z – 1; **if** z > 4 **then exit**; x = y + 5; z = z – 1; **if** z > 4 **then exit**}.

Assignment Statement/Mutator

- Action of an assignment statement
  - Retrieve the values of the identifiers on the RHS expression
  - Evaluate the RHS expression
  - Write the evaluated value into the memory location of the variable
- Assignment statement changes the computational state by changing the store
  - $(\sigma^{\mathbf{E}}, \sigma^{S}, \sigma^{D})$ + assignment-statement → $(\sigma^{\mathbf{E}}, \sigma^{S'}, \sigma^{D})$
- Assignment sequence: x = 4; y = x + 4
  - Assignment x = 4 changes the store to $\sigma^{\mathbf{S}} \oplus \{\text{l-value}(X) \mapsto 4\}$ then y = X + 4 derives $\sigma^{\mathbf{S}} \oplus \{\text{l-value}(X) \mapsto 4\} \oplus \{\text{l-value}(Y) \mapsto 8\}$.
- Multiple assignment: x = y = 4 (Ruby, C++, Python)
  - Applies both the assignments simultaneously on the original store
  - $\sigma^{\mathbf{S}}$ becomes $\sigma^{\mathbf{S}} \oplus \{\text{l-value}(X) \mapsto 4\} \oplus \{\text{l-value}(Y) \mapsto 4\}$
- Parallel assignment: $\text{var}_1, \ldots \text{var}_N = \text{exp}_1, \ldots, \text{exp}_N$
  - Assigns $\text{eval}(\text{exp}_I)$ to var_I

Variations of Assignment

- Major types of assignment statements
  - 1) Destructive update or mutation; 2) Assign-once; 3) Unification
- Destructive update
  - Destroys the old value; supports memory reuse; causes side-effect. One direction information flow. LHS is always a variable
- Assign-once
  - Used in declarative languages. The value of variable once assigned can not be altered; less side-effect; does not support memory reuse without smart program analysis.
- Unification
  - Does not evaluate any expression. Information flow is bidirectional, and variables bound in both LHS and RHS
  - Matches arguments position by position, and is assign-once
  - Used in logic programming

Conditional Statements

- *<if-then-else>* **::=**
  if '('*<cond>*')' **then** *<stat>* [
  else *<stat>*]'**;**'
- *<when-stat>* ::= **When** *<cond>*
  <expr>
- *<unless-stat>* **::=**
  unless *<cond>* *<stat>*
  else *<stat>*
- 'Unless' is equivalent to 'if not'.

General purpose conditional in Lisp

(**cond** *((<predicate$_1$><expr$_1$>)*
…
(<predicate$_N$><expr$_N$>)
(**t** *< catch-all-expr>))*
)

Case Statement

case (*<expr>*) **of** :
<value-set$_1$>: *<command-seq$_1$>*;
…
<value-set$_N$>: *<command-seq$_N$>*;
otherwise: *<command-seq$_{N+1}$>*
end case

Definite Iteration

- Definite iteration: for-loop
  - Four major components: index, lower bound, upper bound and step-size
  - Index and lower limit and upper limits **cannot** be modified by the programmer within iterative block of statements
  - Index is updated after the statement of blocks using the step size
  - Can be simulated using indefinite iteration or tail-recursive procedures
  - Number of iterations = $\lceil$(upper-bound - lower-bound) / step-size $\rceil$
- Multi-paradigm languages like Ruby treat for-loop as methods as every data-element in Ruby is an object
  - 10.**times** {|i| **puts** i} will write 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 in separate lines

Variations of Definite Iteration

- Definite iteration picks index-value from a list of expressions
  - for '(' *<index-variable>* in *<list-of-expressions>* ') ' *<block>*
- A definite iteration picks index-value from an ordered set
  - for '('*<index-variable>* in *<set>* ') ' *<block>*
- Definite iteration can be simulated using indefinite iteration

Definite Iteration	Indefinite Iteration Simulation

for (*<index-variable>* = *<initial-expr>*;
<final-expr>; *<step-expr>*)
<block>
	initial-value = **eval**(*<initial-expr>*);
	step-size = **eval**(*<step-expr>*;
	final-value = **eval**(*<final-expr>*);
	Index = initial-value
	while (index =< final-value) {
	<block>; index = index + step-size;}

Indefinite Iteration

- The embedded iterative block keeps getting executed until the condition is satisfied
  - The condition can be altered in the embedded iterative block
  - The loop can repeat indefinitely if the conditions are always true
- Example constructs of single-entry single exit loops
  - **while** *'('<condition>')'* **'{'** *<statement-block>* **'}'**
  - **repeat** *<statement-block>* **until** *<condition>*
  - **do** *<statement-block>* **while** *<condition>*
- Multiple exit iterative loops (supported in Ada)
  - A general case of single-exit loops
  - There are multiple embedded conditions. If any of the conditions are satisfied then control can get out of the loop

  <multiple-exit-iter> **::=** **loop** { *<cond-exit>* **;** *<command-seq>*}* **end-loop**

  <cond-exit> **::=** **if** *<condition>* **then exit;**

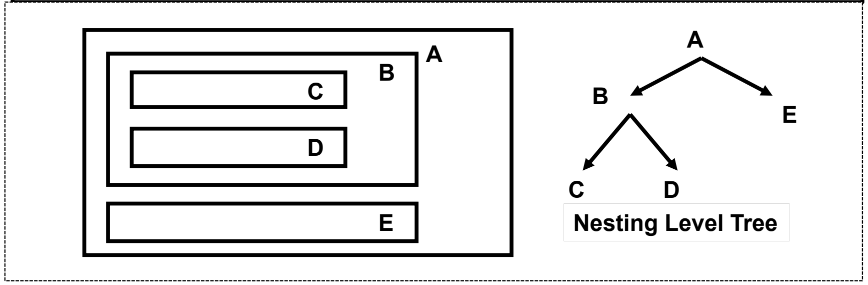
Iterators

- Iterators pick up the elements in a bag in a sequential manner. However, access is not index-based
- Iterators support iteration on list based and bag based collections in declarative and multiparadigm languages
- Example constructs
  - **foreach** *<variable>* **in** *<ordered-bag>* *<block>*
  - **for** (iterator i = data-object**.iterator**( ); i**.hasNext**( );) % in Java
    … visit i.**next**( ) **…**
  - [4, 5, 6]**.foreach** {|i| **puts** i} % will write 4, 5, and 6 in Ruby
- Advantages
  - Low level implementation detail of the data abstraction is not known to the programmer
- Disadvantages
  - Elements can not be skipped. They have to be processed in fixed order

---

Block Structure

- Blocks are a set of declarations followed by a set of commands
  - Blocks can be nested
  - Declarations in outer blocks are visible in inner nested blocks if there is no name conflict. In case of name conflict nearest declaration is visible
  - Sibling blocks do not share the environment
  - Memory used by sibling blocks can be reused by later siblings since memory locations of siblings are not shared



Program Units and Invocations

- A group of declarations and statements bound to an identifier, and can be invoked multiple times from different places
  - Some languages allow nested declarations / nested procedures
- EBNF for abstract syntax of program units

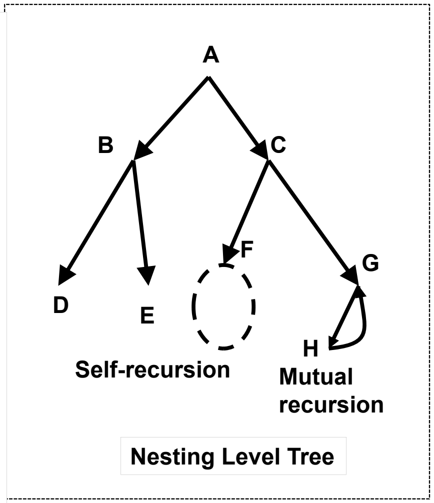
<program-unit> **::=** (**program | function | procedure**) *<identifier> <block>*
<block> **::=** '**{**' [{*<declaration>*}*] [{*<command>*}*] '**}**'
<declaration> **::=** *<sequencer-decl>*| *<type-decl>* | *<variable-decl>*
<command> **::=** null | *<assignment>*| *<conditional>* | *<definite-iteration>* |
<indefinite-iteration> | *<iterator>* | *<subprogram-call>*|
'**{**'*<block>*'**}**' | *<sequencers>*

- Compilation process is independent of number of invocations
  - Each program module is compiled exactly once
- **Invocation of a program is modeled as a DAG**
  - Direction from calling program unit to called unit
  - Each invocation of a program unit creates a distinct environment

---

Recursive Procedure

- Self-recursive procedure
  - A procedure that calls itself
- <u>Example</u>: factorial(n)
  - **function** fact(**integer** n)
    return (n * fact(n – 1))
  - **fact(3) → fact(2) → fact(1)**
- Mutually-recursive procedure
  - Starts a cycle of invocation such that last invocation invokes the first procedure
  - length(cycle) > 0
  - **P_0,** P_1, …, P_N, **P_0**



Nesting Level Tree

Modules

- Module is a logical encapsulation to control visibility of the embedded declarations and program units
  - Provides multiple units to use the same name
  - Allows reusable library by importing the module in a program
  - Regulates interaction with the program by using export and import.
- Declaration can be
  - Importing some program unit from other module
  - Exporting some program unit to make it visible outside
  - Declared variables or program unit or nested modules
- Abstract syntax

<module> **::= module** *<identifier>*
[**export** {*<program-unit>*}*]
[**import** {*<program-unit>*}*]
{ *<declaration>*}* {*<program-unit>*}* {*<module>*}*
end *<identifier>*

Export and Import

Export

- Makes an embedded unit visible to other program units
  - All the embedded units can be exported
  - A limited set of embedded units can be exported selectively

Import

- Allows an exported unit from other module to be used
  - All the exported units from a module can be imported
  - A limited set of exported units can be imported
  - Unless imported, exported units are not accessible

Objects and Classes

- An independent active unit of information
  - Encapsulates data entities and methods to work on data-entities
  - Encapsulation provides information hiding
  - Has a runtime computational state
  - Objects can be flat (single level) or nested depending upon language design philosophy
  - Methods / data-entities in an object are accessed by
    - *<object-name>. <method-name>* or *<object-name>.<entity-name>*
- Object-classes
  - Passive template generally with a hierarchical structure
  - Terminology used are base-class for the root node, subclasses for the descendant nodes
  - Objects are active instances of the object classes
  - Supports inheritance of methods between ancestor class and descendant classes

---

Class Hierarchy and Inheritance

Inheritance and Visibility

- Class structure is nested
  - Child subclass inherits from the parent class
  - Inheritance relationship is transitive and anti-symmetric
- Properties of inheritance
  - Inherited methods can be overridden
  - Methods can be sealed in parent class to avoid inheritance
- Visbility
  - Private methods/entities are invisible to objects from other classes
  - Public methods / entities are visible to objects from other subclasses / classes

Abstract Syntax and Example

<class> **::= class** *<identifier>*
[**subclass-of**] *<identifier>*
[**private** {*<data-decl>*}*]
[**protected** {*<data-decl>*}*]
[**public** { *<data-decl>* }*]
[**private** {*<method-decl>*}*]
[**protected** {*<method-decl>*}*]
public {*<method-decl>*}+

Examples

Hash-tables, Arrays, Matrices

Supporting Languages

CLOS, C++, Java, C#, Modula-3, Ruby, Scala

Information Exchange

- A means of sharing subset of environment and store between two program modules using attributes of information holders
  - Information holders are variables, identifiers
  - Attributes are name, address of the entity, and value of the entity
- Information is communicated using different mechanisms
  - **Global variables** – all programming modules can see
  - **Nonlocal variables** – outer modules → inner modules
  - Point of point visibility using parameter attributes of variables such as name, address, or value
- Examples
  - Block structured languages use global / nonlocal variables and various parameter passing mechanisms
  - Fortran also uses a common block defined as 'Common'. It is error prone due to alignment of two arrays

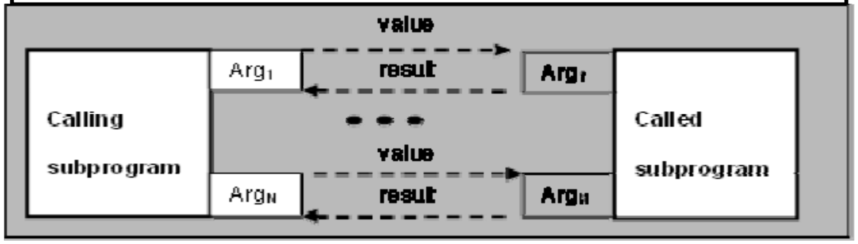
---

Parameter Passing I

- Point to point information exchange between calling and the called program
  - The point of information exchange in the calling program is called **actual parameter**
  - The corresponding point in the called program is called **formal parameter**
- Parameter attributes can be name, address or value
- Different Correspondence between actual and formal parameter
  - Aligned left to right – most popular
  - Matched using name association as in Ada
  - If numbers of actual and formal parameters do not match then formal parameter takes default values
  - 'Param' declaration – formal parameter can take indefinite size of actual parameters by treating as a list

Parameter Passing II

- Variable is **name** ↦ **location** ↦ **value**
- Call by name – textual substitution of formal parameter by actual parameter text
- Call by reference – formal parameter is a pointer to the location of the actual parameter
- Passing the value: three possibilities
  - Call by value; call by result; and call by value result



Variations in Other Paradigms

- Information exchange in object-oriented languages
  - *Class variables* seen in every objects of every subclass of a class
  - *Static global variable* seen across all the objects of every class
  - Parameter passing between methods of the same class
- Information exchange in functional programming languages
  - Can pass the whole function as a parameter
  - Dual nature of building function as data, and then transforming back to a function
  - Substitutes formal parameter by the text of the actual parameters
- Information exchange in logic programming languages
  - Uses unification of logical terms for parameter passing

Call by Value and Variations

- Also called call-by-copy or call-by in-mode
- Process of call-by-value
  - Creates memory locations fort each formal parameter
  - Evaluates the expression of the actual parameter
  - Evaluated value is copied to formal parameters' memory locations
  - After copying the evaluated value there is no more information sharing
  - Information sharing is one way actual parameter → formal parameter
- Advantages
  - Store of the calling procedure is not updated by the called procedure
  - Used when called procedure needs the value to perform computation
- Disadvantages
  - Result is not shared with the calling procedure
  - Excessive memory requirement and copying overhead in case of large data structures

---

Examples of Call-by-value

Example 1

```
program main
{   integer x, y, z;
   read(x, y);
   z = square_sum(x, y)
   print("square sum of the numbers:
~d and ~d is ~d", x, y, z);
}
function integer square_sum(a, b)
{   return(a*a + b*b); }
```

- **Formal parameters: a and b**
  - a ↔ x and b ↔ y
  - x and y are copied in a and b
  - x and y are not destructively updated by a and b

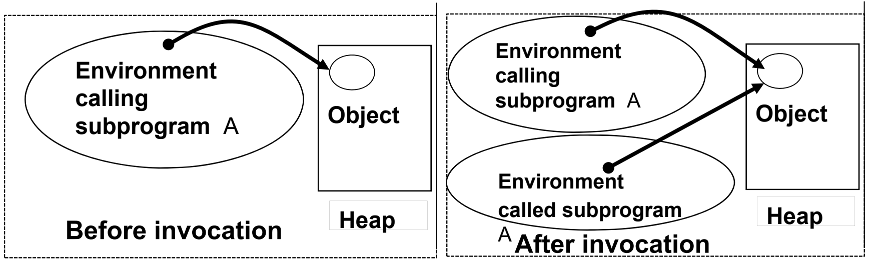
Example 2

```
program main
{integer x[100]; y[100];
  for (i = 0, i =< 99; i++) read(x[i], y[i]);
      call my_print(x, y);
}
subprogram  my_print(integer
a[100],  b[100])
{ integer c[100];
    for (i = 0; i =< 99; i++) {
           c[i] = a[i] + b[i];
    for (i = 0; i =< 99, i++)
        print("c[~d] =" ~d~n", i, c[i]);
    }
}
```

- **Excessive memory locations**

Sharing Complex Objects

- Complex objects: recursive data structures, extensible objects, dynamic objects are stored in heap
  - A pointer (or reference) from the environment to the object's first location in heap
  - Objects are shared by copying the pointer (or reference)
- Sharing is done by
  - call by copying – a variation of call by value


Environment calling subprogram A
Object
Heap
Before invocation
Environment calling subprogram A
Environment called subprogram A
Object
Heap
After invocation

Call by Reference (or Access)

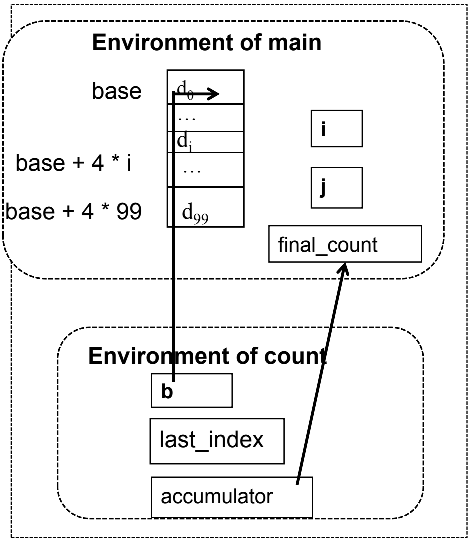
- Passes the l-value of the actual parameter to the formal parameter
  - Formal parameter is just a pointer to the first memory location of the actual parameter
  - A specific data element is accessed using a combination of formal parameter to access the base address and offset of the data-entity within the actual parameter
- **Advantages**
  - Minimal memory overhead for complex data structures
  - No need to explicitly pass the result back to actual parameter
  - Copying overhead during parameter passing is absent
- **Disadvantages**
  - Actual parameter location can be erroneously updated by called subprogram causing incorrect program behavior
  - Extra level of indirection causes memory access overhead in called subprogram
  - In distributed memory space call by reference is expensive due to data residing in different processor

---

Illustration – Call by Reference

Program

```
program main
{ integer d[100],  i, j, final_count;
  for (i = 0; i =< 99; i++)
     d[i] = random_number(1, 200);
      j = 50;
      call count (& d, j,  & final_count);
}
subprogram count (integer *b,
last_index,  *accum)
{ integer index;
   *accum = 0;
    while (index =< last_index)
    {if (*b[index] > 100)
         *accum = *accum + 1;  % end_if
          index = index + 1;
} % end_while
```



Variations of Call by Reference

- Read-only call by reference: in C++ and Modula3
  - Actual parameter can only be read but not updated
  - Advantage: no additional memory overhead or copying cost
  - Disadvantage: additional overhead of indirection in accessing
- Call by sharing: used in C#, Java, CLU, and C++
  - First parameter passing is call by reference; subsequent are call by value
  - All call subprograms point to the actual parameter memory location
  - **Advantage:** same as call by reference for the chain of called subprograms
  - **Disadvantage**: **same as call by reference**

Call by Value vs. Call by Reference

Call by Value	Call by Reference
Excessive copying cost	Minimal copying cost
Excessive memory overhead	Minimal memory overhead
No passing of the result	Continuous update of the actual parameter
Single memory access overhead	Additional memory access overhead due to indirection
No erroneous update of the actual parameter	May cause erroneous destructive update of actual parameter location
Useful when the local computation on passed parameter is heavy in the called procedure	Useful when the local computation is light but passed parameter is quite large in size

---

Call by Result

- Also called copying out mode
- Mechanism
  - The actual parameter does not effect initial value of the formal parameters
  - The formal parameter is initialized to some default value
  - During the execution of the called procedure, there is no communication between formal and actual parameters
  - After the called procedure is over, the result from formal parameters is passed back to the corresponding actual parameters
- Overheads: same as call-by-value
  - Excessive overhead of copying cost and memory allocation

Call by Value-Result

- Also called *parameter passing by in-out mode*
- Actual parameter value ←→ formal parameter value
  - Like call-by-value, formal parameter is treated like local variables
  - Evaluated value of the actual parameter expression is passed initially to the formal parameter
  - No communication during the execution of the called procedure
  - The result is passed back after the end of the called procedure
- Advantages
  - Like call by reference the result is passed back to the actual parameter
  - Formal parameters are accessed directly as local variables
- Disadvantages
  - Twice the copying overhead of call-by-value
  - Same memory overhead as call by value

Example – Call by Value-result

```
Program main
{ integer a[100], b[100], c[100, 100], i;
  for (i= 0;  i =< 99; i++) read(a[i]);
for (I = 0; i =< 99; i++) read(b[i]);
call multiply (value-result a[100],
b[100], c[100, 100]);
}
subprogram multiply(integer x[100],
y[100], z[100, 100])
{ integer i, j;
  for (i = 0; j =< 99; i++)
      for (j = 0; j=< 99; j++)
            z[i, j] = x[i] * y[j];
}
```

- Copies the values of
  - a[100] into x[100],
  - b[100] into y[100] and
  - c[100, 100] into z[100, 100]
- Passes the result back of
  - x[100] into a[100]
  - y[100] into b[100]
  - z[100, 100] into c[100, 100]
- Information is passed left to right
  - a[100] ←→ x[100] followed by b[100] ←→ y[100] followed by c[100, 100] ←→ z[100, 100]

Call by Reference vs. Value-result

Call by Reference	Call by Value-result
Continuously modifies the actual parameters with computation order	Modifies actual parameters only after the end of called procedure in correspondence order
Needs only one memory location to point to actual parameter	Needs equal number of memory locations as actual parameters
Additional overhead of accessing due to additional level of indirection	Minimal overhead of access as formal parameters are in local environment
No copying cost	Excessive copying cost
Good where actual parameter has large memory locations but computational requirement is less	Good when computation outweighs the copying overhead

---

Example of Difference

Call by Reference

```
main ( );
integer i;
{i = 1; sub(& i, & i);}
void sub(integer  *j,  *k);
{*k = 4; *j = 2}
```

- The final value is dependent upon the order of writing into the memory location.
  - The formal parameters j and k both point to memory locations of actual parameter I
  - The final value in call by reference is i = 2

Call by Value-result

```
main ( );
integer i;
{i = 1; sub(value-result i, i);}
void sub(integer  j,  k);
{k = 4; j = 2}
```

- The final value is dependent upon the order of correspondence between the formal and actual parameter
  - The last value passed is k
  - The final value is decided by the value of k and not the last computation
  - The final value of I is 4

Call by Name

- Formal parameter substituted by text of the actual parameter
- Substituted body is evaluated on demand using thunking
  - Thunk is a parameterless procedure that is evaluated every time on demand when actual parameter is accessed in calling procedure
  - Name conflict between actual parameter and local variable in the called procedure is resolved by renaming the local variable
  - Expression evaluation is delayed until actual parameter is accessed

Program

```
 program main
{    integer x, y, z;
     real w;
     x = 3; y = 4; z = 5;
    call sub(x + y,  x + z,  w); }
subprogram sub (name a,  b, w)
{   integer z;
    z = a * a + b * b; w = square_root(z); }
```

Runtime behavior after call to sub

```
 program main
{    integer x, y, z;
     real w;
     x = 3; y = 4; z = 5;
     {    integer z1; % rename the variable
          z1 = (x + y) * (x + y) + (x + z) * (x + z);
          w = square_root(z1); }
}
```

Issues in Call-by-Name

- Overhead of demand based evaluation every time actual parameter is accessed
- Overhead of runtime renaming of local variables
- Memory location mix-up with name due to on demand evaluation
- In the following example, instead of swapping the value of k =2 and a[2] in the second swap call, it copies the value of k = 2 in a[0], and a[2] remains untouched

```
program main
{   integer  i, j, k; integer a[5];
    k = 3; j = 2;
    for (I = 0; I =< 4; i++) a[i] =0;
    swap(name j, k);
    swap(name k, a[k];  }
subprogram swap(name m, n);
{   integer  temp;
    temp = m; m = n; n = temp; }
```

```
program main
{   integer  i, j, k; integer a[5];
    k = 3; j = 2;
    for (I = 0; I =< 4; i++) a[i] =0;
    {   integer temp;
        temp = j; j = k; k = temp; }
    {  integer temp;
        temp = k; k = a[k]; a[k] = temp;  }
}
```

Call by Need – A Variation

- Call by need caches the evaluated value after the first time evaluation of the expression.
  - Value is retrieved from the cache in future
  - Call by need is call-by-name in the first evaluation and call by value in subsequent evaluation
- <u>Example</u>
  - z1 = (x + y) * (x + y) + (x + z) * (x + z);
  - (x + z) and (x + y) are evaluated only once, and the value is put in cache. Second time it is looked in cache
- Advantages
  - More efficient than call by name
  - In most cases indexes of the subscripted variables is not reevaluated. Hence there no need of call by name
  - Used in functional language Haskell to improve the efficiency

Passing Subprograms as Parameters

- Pass the reference to the first instruction and the environment of the function to the called procedure
- Issues
  - Runtime checking of the arguments of the passed function such as type information and number of arguments
  - Statically typed languages have this problem of checking types of arguments
  - Dynamically typed languages do not have such problem
  - Dynamically typed languages can also check types using metalogical predicates
- **Not preferred in statically typed languages**

Parameter passing in Distributed Computing

- Distributed computing needs invocation of procedures on remote processors
  - Calling program and called subprogram may execute on different address spaces
  - Information transfer requires communication across processors
  - Involves evaluation of expression and copying the object to remote processors
- Types of parameter passing
  - **Call by moving also called call-by copy:** makes a copy of the object on the remote processor executing called procedure. However, object is not copied back. It is equivalent to call-by-value
  - **Call by reference:** The address of the object of the calling program is transferred to the remote processor. Object remains on the original processor.
  - **Call by visit or call by copy restore:** makes a copy of the object on the remote processor executing called procedure. Resulting object is copied back. It is equivalent to call by value-result

Side-effects

- **A permanent effect that outlives the called subprogram**
  - Updates the store of other programs
  - Writing in the persistent objects such as file or stream
  - Raising an exception
- **Store of other programs can be effected by the use of**
  - Global variables; reference; non-local variables; persistent data objects
- **Usage**
  - Send results to calling programs
  - Sharing results of partial computations with other programs
  - Memory reuse
- Problems
  - **Loss of commutativity**

```
program main
{ integer A, B, X, Y;
  X = 3; Y = 4;
  A = sq_sum(&X, &Y) + X; % A = 34
  B = X + Y;  % B = 25
  print(A, B, X, Y);
}
function integer sq_sum(integer  *X, *Y);
{ *X = *X *  *X; *Y = *Y  *  *Y;
   return (*X + *Y);
 }
```

Scratch pad computation on X and Y in function sq_sum effects the actual parameters incorrectly; X becomes 9; Y becomes 16 after call to sq_sum

sq_sum(&X, &Y) + X is not the same as X + sq_sum(&X, &Y)

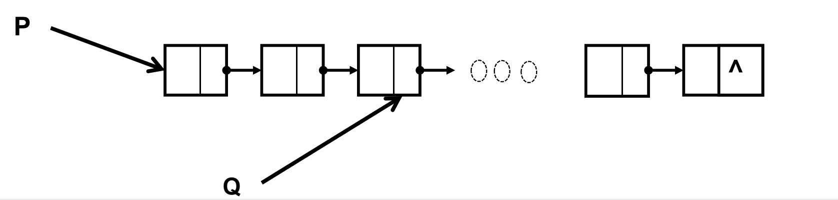
Aliasing and Side-effects

- Aliasing means two names or pointers map to same memory location
  - Binding one variable also binds the other variable
  - May have drastic effect if the shared location is written into in called procedure
- **Example**
  - First swap properly swaps the value of y and z
  - Second swap makes x = 0 instead of keeping the same value due to the use of formal parameters x and y pointing to same location

```
program main
{ integer  x, y, z;
  x = 3; y = 4; z = 5;
   swap(&y, &z);  % y ← → z
   swap(&x, &x);  % move(x, 0)
   print(x, y, z)
% x = 0; Y = 5 and z = 4
 }
subprogram swap(integer *x,   *y);
{           *x = *x + *y;
            *y = *x - *y;
            *x = *x - *y;
}
```

Shared Data-structures

- Multiple pointers pointing to a shared subset of memory locations
- Releasing one pointer may release and recycle shared memory locations
- This may cause allocation of recycled memory to other processes causing memory corruption
- In the figure below releasing P may also release data structure shared by Q



Regulating Side Effects

- Programmer's discipline
  - Use local variables for scratch-pad computations
  - Variables with scope outside the local environment should be modified only for known information exchange
- Disallow pointer arithmetic
  - No arithmetic on pointer types to disallow type violation across data-entities at run-time
- Disallow independent pointers
  - Pointers only associated with recursive data structures or dynamic data objects internally to avoid inconsistent operations across different data-entities of different types
- Disallow destructive updates
  - Variables are assign-once as in functional and logic programming
  - The variables in calling programs are not modified by the called subprograms

---

A Comprehensive Example

```
program main( )
{          integer i, j, k, a[6];
           i = 0; j = 0; k = 2;
           for (i = 1; i =< 5; i++) a[i] = 10;
           messy(a[1], &a[2], &j, &j, #a[3], $k);
}
subprogram   messy(integer  A, *B, *C, *D, E, F)
{   A = *B + *C;  % statement 1: A = value(a[2]) + value(j) = 10 + 0 = 10
    *B = *D + E + F;   % a[2] = value(j) + value(E) + value(F)  = 0 + 10 + 0 = 10
    *C = A + E; % j = value(A) + value(E) = 10 + 10 = 20
    *D = *C - *D; % j = value(j) – value(j) =  20 – 20 = 0
     E = *B + *C;  % E = value(a[2]) + value(j) = 10 + 0 = 10
     F = E + A; % statement 6: value(E) + value(A) = 10 + 10 = 20
}
```

Exception Handling

- A programming language construct for graceful termination in case of error conditions
  - Error can be data dependent and not logical
  - Operating system traps do not provide graceful error-handling
  - Handlers can be built-in or user defined
- Exception handlers can:
  - Pass control to other routine;
  - Release resources;
  - Correct error condition,
  - Return control to the next instruction after the handler
  - Return to the calling routine

<extend-stat> ::= **try** *<stat>*
if *<exp$_1$>* **raise** *<excep$_1$>*;
if *<exp$_2$>* **raise** *<excep$_2$>*;
…
if *<exp$_M$>* **raise** *<excep$_M$>*;
exception-handlers
{ **when** *<excep$_1$>*: *<block$_1$>*
when *<excep$_2$>*: *<block$_2$>*
…
when *<excep$_N$>*: *<block$_N$>*;
}
[**finally** *<block$_F$>*]

Example of Exception Handling

```
subprogram illustrate_exceptions
integer  i;
real deposit, account;
file myfile;
stream mystream;
exception  incorrect_debt;  % user defined exception
open_file(myfile, mystream, read);
exception-handler {
        when file-not-found: write('Account file missing'); return }
read(mystream, account);
if (account > 0)   raise incorrect_debt;   % raise the user defined exception
exception-handler {  % handle the user defined exception
        when incorrect_debt: write('Incorrect deposit '); close(mystream); return; }
close(mystream);
return
```

- **Two exceptions: file-not-found and incorrect_debt**
  - File-not-found is built-in, and incorrect_debt is user-defined

---

Nondeterministic Computation - I

- Allows alternate control flows provided final condition is met
- If the solution of the problem is modeled as directed graph:
  - Each computational state is modeled as node in the graph
  - Each edge shows the statement executed to go to next state
  - Nondeterministic computation allows multiple paths from the initial state to the final state
- Properties of programs supporting non-determinism
  - Commutativity of operators such as '+', '*', logical-OR, and logical-AND
  - Selection statements and constructs such as if-then-else statements, case statements
  - Two independent statements whose effect on store is disjoint
- <u>Example</u>

if (X >=Y) then smaller = Y |
if (Y >= X) then smaller = X ⇔
	if (Y >= X) then smaller = X |
	if (X >= Y) then smaller = Y

Nondeterministic Computation - II

- Two parts to a computation
  - Part that checks the computational state, and can only read the values
  - Part that modifies the store
  - Boolean expressions that only read the values can be executed in any order
  - All Boolean expressions are equally likely

If { *<Bool-Expr$_1$ >* → *<Com$_1$>* |
<Bool-Expr$_2$ > → *<Com$_2$>* |
… |
<Bool-Expr$_n$ > → *<Com$_n$>* **}**

Guarded Commands

- A high level construct for nondeterministic computation
  - A guarded command is a Guard followed by the corresponding command
  - If then-else and case statements can be transferred to a disjunction of guarded commands called guarded-commands.
- Features of guarded commands
  - Boolean expressions are weakest preconditions known as guards that are tested before executing the corresponding commands
  - Guards can only read from store; only commands can write
  - Guards are necessary conditions but not sufficient
  - Corresponding command is executed only after the guard succeeds
  - After a guard succeeds, other guards are not tried
- Limitations of guarded commands
  - Guarded commands are incomplete. A solution may exist in a different guarded command other than the guarded command where guard succeeds
- Advantages
  - Can be used to scientifically build program using stepwise construction

---

Guarded Commands Constructs

- Two major constructs
  - Selection construct
  - Loop construct
- Selection construct
  - Described earlier
- Loop construct
  - Guarded commands embedded inside a loop
  - Loop executes until there is no successful guard in the embedded guarded commands
  - Final condition includes

$\neg$ Guard$_1$ $\wedge \ldots \wedge$ $\neg$ Guard$_N$

- Selection construct

If { *<Guard$_1$ >* $\rightarrow$ *<Com$_1$>* |
…
<Guard$_n$ > $\rightarrow$ *<Com$_n$>***}**

- Loop construct

loop { *<Guard$_1$ >* $\rightarrow$ *<Com$_1$>* |
…
<Guard$_n$ > $\rightarrow$ *<Com$_n$>* **}**

- Example

```
loop {
          a0 > a1 → swap(a0, a1) |
          a1 > a2 → swap(a1, a2) |
          a2 > a3 → swap(a2,  a3)
     }
```

Final condition: a0 ≤ a1 ≤ a2 ≤ a3

Stepwise Program Construction

- Progressively move from final condition to initial condition
  - Use axiomatic semantics to find out the weakest preconditions using post conditions and statements
  - Use De Morgan's law to derive out individual guards from pre-conditions
- Example

Final condition: $a_0 \le a_1 \le a_2 \le a_3 \rightarrow a_0 \le a_1 \wedge a_1 \le a_2 \wedge a_2 \le a_3$

Statements: swap(a_i, a_{i+1}) **Pre-condition** $a_i > a_{i+1;}$ Post-condition $a_i \le a_{i+1}$

Given statement and post condition

$a_0 \le a_1$ **and** swap(a_0, a_1) **precondition:** $a_0 > a_1$
$a_1 \le a_2$ **and** swap(a_1, a_2) **precondition:** $a_1 > a_2$
$a_2 \le a_3$ **and** swap(a_2, a_3) **precondition:** $a_2 > a_3$

De Morgan's theorem connects logical-and to logical -r of conditions in guards

Guard is $a_i < a_{i+1}$ and command is swap(a_i, a_{i+1})

Program as Data

- Many languages support duality of program and data
  - Program are constructed as data, converted to program, and executed
  - Program constructed as data is called first class objects. The process of representing a program as data is called reification
- Application of this dual nature
  - Programs are written to analyze or reason about another programs for useful properties at runtime. Such programs are called meta-programs
  - An editor needs to modify a program as data
- Functions as first class objects
  - Declarative languages support metalogical predicates that convert data to function (or predicate) and vica versa
  - **Example:** (apply 'first '(Arvind Tom)) returns 'Arvind
- Meta Programming and Reflexivity
  - Manipulate or reason about program properties
  - Languages supporting metaprogramming are called **reflexive**
  - **Example:** Lisp, Prolog, Scala, Ruby etc.

Software Reuse and Interoperability

- Need for software reuse
  - To avoid duplication of the previously developed software
  - To reduce bugs during software development
- Need for Interoperability
  - To integrate with software developed using different languages in different programming paradigms
  - Paradigms often differ in supported control and data abstractions
- Interoperability revisited
  - Information transfer across subprograms written in different languages
  - Two approaches to solve the problem of interoperability: 1) development of a common middleware language, or 2) translation of data types from one language to another language
- Common Middleware Language
  - Uses common type and metadata to transfer information across different languages
  - Metadata contains abstract information in tables about data structures in compiled program
  - Common middleware approaches are .NET and Java Virtual Machine

---

Summary I

- Data abstractions can be single entity, composite entity, collection of data entities or extensible data entities
- Single data entity could be
  - Mathematical type such as integer, float, Boolean, character, string
  - Enumeration type
- Composite entity could be name tuple where each field is
  - A composite entity, single entity, collection
- A collection could be
  - A set, an ordered bag, and a bag of key value pairs
- Extensible entities can be implemented using
  - Linked-lists, vectors, trees, and hash tables
- Control abstractions can be
  - Assertion; expression evaluation; definite and indefinite iteration like for-loop, while-loop, do-while loop; selection like if-then-else and case statements; recursions; and procedure calls
  - Blocks provide natural visibility boundaries for data and code

---

Summary II

- Modules regulate visibility of embedded subprograms and entities
  - Blocks are contained within subprogram; module contain subprograms
  - Modules can be saved as library to be used by different programs
- Information exchange between subprograms
  - Variables are information holders needed to exchange information
  - Using global variables, non-local variables, and parameter passing
- Parameter passing
  - **Value exchange based**: call by value and call by sharing. Call by value copies the value, and call by sharing copies the memory location. Call by sharing is used in object based languages to access objects in heap
  - **Memory location exchange**: call by reference. Copies the address of the actual parameters' base address into formal parameter
  - **Name exchange**: the formal parameter is substituted by the text of actual parameter, name conflicts are removed by local variable renaming and substituted program is used instead of procedure call
  - **Call by need**: call by name during the first invocation followed by call by value in subsequent expression evaluation

---

Summary III

- Information exchange in distributed computing
  - **Call by copying** copies the object to remote processor
  - **Call by reference** copies the address of the object to remote processor
  - **Call by visit** copies the object to remote processor that is copied back after the termination of the called procedure
- Side-effects and aliasing
  - Updating a store that modifies the store beyond the lifetime of the called subprogram; used for information exchange; can have bad effect if used for purposes other than information exchange
  - Can break fundamental laws of computations such as commutativity
- Exception Handling
  - Exception handlers are programming language constructs to recover from error conditions gracefully without crashing the program
  - Exception handlers return the control to a block that handles the error, and then either exits or takes the control back for further execution

Summary IV

- Software Reuse and Interoperability
  - Needed for robust software development with reduced duplication, and for using programs written in different languages
  - Interoperability needs development of common interface data structure, and middleware languages to which all other languages are translated
- First class objects and meta-programming
  - First class objects are functions built as data at runtime
  - Metaprograms treat other programs as data, reason about programs, or transform other programs to new programs
- Nondeterministic computation and guarded commands
  - Allows alternate paths in the computational space provided final condition is met
  - Guarded commands is a high level programming constructs used for nondeterministic computation
  - Guarded commands has two parts: guard that tests the weakest precondition in the computational state, and command part that alters the computational state. Guards are necessary but not sufficient conditions