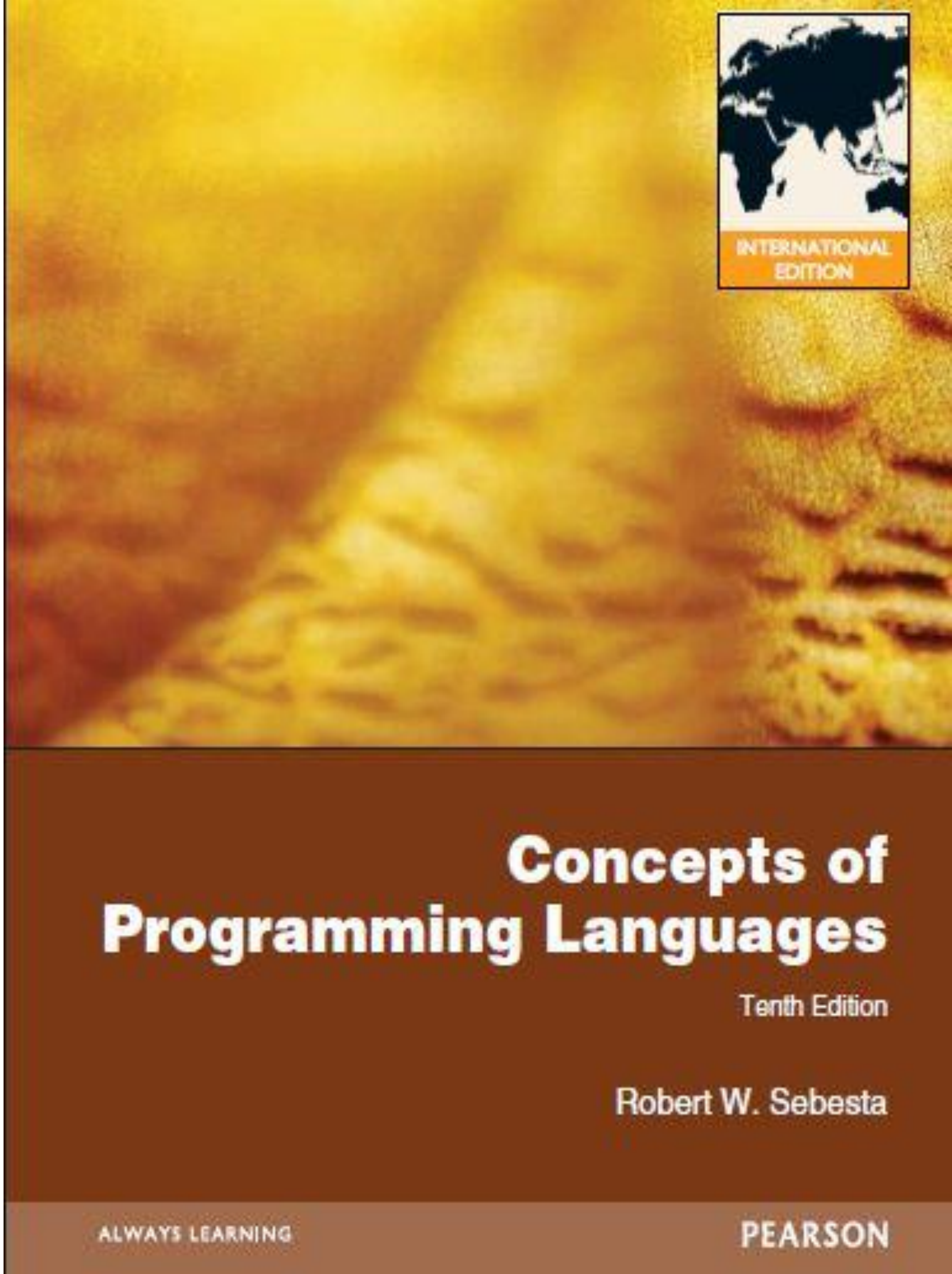
Programming Language



Instructor:
Min-Chun Hu
anita_hu@mail.ncku.edu.tw

Lecture 9 Subprograms

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded Operators
- Closures
- Coroutines

Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
  - Data abstraction
    - Emphasized in the1980s

Fundamentals of Subprograms

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates

Basic Definitions

- A *subprogram definition* describes the interface and the actions of the subprogram abstraction
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
  - Python, Ruby:
    - **`def`** `functionname (parameters) :`
  - JavaScript:
    - **`function`** `functionname(parameters)`
  - C-based language:
    - **`void`** `functionname(parameters)`

Basic Definitions (Cont.)

- In Python, function definitions are executable. The function cannot be called until its **def** has been executed
  - ```
    if …
      def fun(…):
        …
    else
      def fun(…):
        …
    ```
    ```

Basic Definitions (Cont.)

- In Ruby, function definitions can appear either in or outside of class definitions.

```
ruby> foo = "abc"
      "abc"
ruby> foo.length
      3
ruby> foo = ["abcde", "fghij"]
      ["abcde", "fghij"]
ruby> foo.length
      2
ruby> foo = 5
      5
ruby> foo.length
     ERR: (eval):1: undefined method `length' for 5(Fixnum)
```

Basic Definitions (Cont.)

- In Lua, all functions are anonymous
  - `cube=function(x) return x*x*x end`

Basic Definitions (Cont.)

- The *parameter profile* of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type
- Function declarations in C and C++ are often called *prototypes* (often placed in header files)
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

- Positional parameters
  - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
  - Safe and effective
- Keyword parameters
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
    - `sumer(length=my_length,list=my_array,sum=my_sum)`
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*: User must know the formal parameter's names

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
  - **Python example:**

```
def compute_pay(income, exemption=1, tax_rate)
pay = compute_pay(20000.0, tax_rate=0.15)
```

  - In C++ (no keyword parameters), default parameters must appear last because parameters are positionally associated:

```
float compute_pay(float income, float tax_rate, int
exemption=1)
pay = compute_pay(20000.0, 0.15);
```

Variable numbers of parameters

- C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **`params`**
  - `public void DisplayList (params int[ ] list)`

Variable numbers of parameters

- Ruby supports a complicated but highly flexible actual parameter configuration. The actual parameters can be sent as elements of a hash literal and the hash item can be followed by a single parameter preceded by an asterisk.

```
List = [2, 4, 6, 8]
def= tester(p1, p2, p3, *p4)
 …
end
…
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

p1 is 'first'

p2 is {mon => 72, tue => 68, wed => 59}

p3 is 2

p4 is [4, 6, 8]

Variable numbers of parameters

- Python supports parameter that are similar to Ruby
  - the actual is a list of values and the corresponding formal parameter is a name with an asterisk

```
def manyArgs(*arg):
  print "I was called with", len(arg), "arguments:", arg

>>> manyArgs(1)
I was called with 1 arguments: (1,)
>>> manyArgs(1, 2,3)
I was called with 3 arguments: (1, 2, 3)
```

- Lua use a simple mechanism
  - a variable number of parameters is represented as a formal parameter with three periods; they are accessed with a `for` statement or with a multiple assignment from the three periods

```
function DoIt (…)
        local a, b, c = …
        d=a*b*c
        return d
end

DoIt(4, 7, 3)
```

Procedures and Functions

- All subprograms are collections of statements that define parameterized computations
- There are two categories of subprograms
  - *Procedures/Subroutines*
    - do not return values
  - *Functions*
    - return values or not
    - structurally resemble procedures but are semantically modeled on mathematical functions
    - They are expected to produce no side effects. (It modifies neither its parameters nor any variable defined outside the function.)
    - In practice, program functions have side effects

Design Issues for Subprograms

- Are local variables statically or dynamically allocated?
- What parameter passing methods are provided?
- Is parameter type checking needed?
- Can subprograms be overloaded or generic?
- Can subprogram definitions appear in other subprogram definitions (nested)?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- If the language allows nested subprograms, are closures supported?

Local Referencing Environments

- Local variables can be stack-dynamic
  - Advantages
    - Support for recursion
    - Storage for locals variables in an active subprogram can be shared with the local variables in all inactive subprograms
  - Disadvantages
    - Allocation/de-allocation, initialization time
    - Indirect addressing
    - Subprograms cannot be history sensitive
- Local variables can be static
  - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

Local Referencing Environments: Examples

- In most contemporary languages, local variables in a subprogram are stack dynamic
- In C-based languages, locals are by default stack dynamic, but can be declared **`static`**
- The methods of C++, Java, Python, and C# only have stack dynamic locals
- In Lua, all implicitly declared variables are global; local variables are declared with **`local`** and are stack dynamic

Parameter Passing

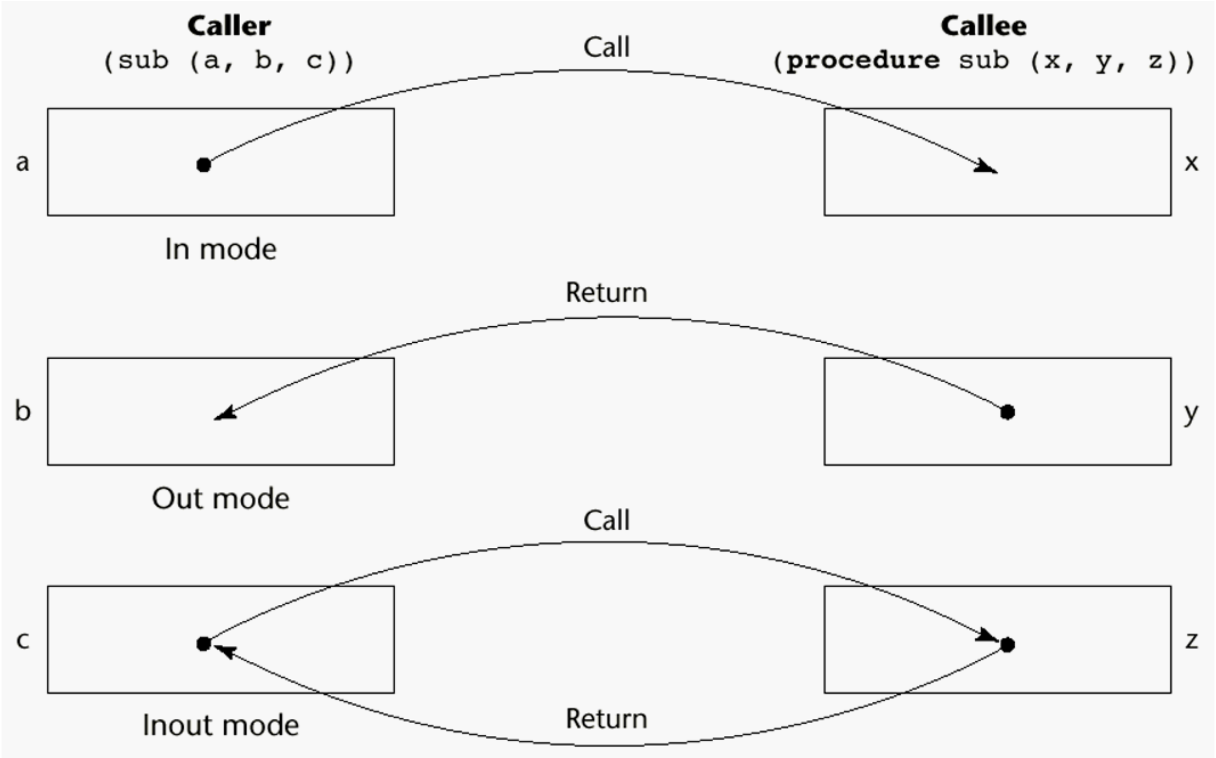
- Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms.
- Three semantics models of parameter-passing:
  - In mode
  - Out mode
  - Inout mode

Nested Subprograms

- If a subprogram is needed only within another program, why not place it there and hide it from the rest of the program?
- Algol 60, Algol 68, Pascal, Ada, JavaScript, Python, Ruby, Lua, and most functional programming languages allow subprograms to be nested

Parameter-Passing Methods

- Parameter-passing methods:
  - the ways in which parameters are transmitted to and/or from called subprograms.
  - Semantics models of parameter passing:



Pass-by-Reference (C example)

Pass the address of variables to simulate inout mode

```
caller() {
   int a =3;
   int b=1;
   int k[10];
   k[3] =7;
   swap(&a, &b);
   swap(&b, &k[b]);
}
swap(int *c, int*d) {
  temp =*c;
  *c =*d;
   *d =temp;
}
```

Pass-by-Name (Inout Mode)

```
integer a=3;
integer b=1;
integer k[10];
k[3]=7;
swap(a, b);
swap(b, k[b]);
swap(int x, int y) {
  temp =x;
  x =y;
  y =temp;
}
```

Conceptual Models of Transfer

- Two conceptual models of how data transfers take place in parameter transmission:
  - Physically move (copy) a value
  - Move an access path (pointer/reference) to a value

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying because accesses are more efficient
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move):
    - additional storage is required (stored twice) and the actual move can be costly for large number of parameters
  - *Disadvantages* (if by access path method):
    - write-protection is required and accesses cost more (indirect addressing)

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram
- The corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
- Require extra storage location and copy operation

Pass-by-Result (Out Mode)

- Potential problems:
  - Actual parameter collision:
    - `sub(p1, p1);` whichever formal parameter is copied last will represent the current value of `p1`
    - **C#:**

```
void Fixer(out int x, out int y){
    x=17;
    y=35;
}
…
f.Fixer(out a, out a)
```

  - Compute address of list[sub] at the beginning of the subprogram or end?

```
void DoIt(out int x, int index){
x=17;
index=42;
}
…
sub=21;
f.DoIt(list[sub], sub);
```

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
  - The actual parameter is copied to the formal parameter at the subprogram entry and then copied back at subprogram termination
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

Pass-by-Reference (Inout Mode)

```
void fun(int &first, int &second)
```

- Pass an access path (address)
- Also called pass-by-sharing
- Advantage:
  - Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters because of indirect addressing
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases harmful to readability and reliability (access broadened)

Pass-by-Reference (Inout Mode)

- Aliases are created when:
  - Pass the same variable twice:

    ```
    fun(total, total)
    ```
  - Collisions between array elements:

    ```
    fun(list[i], list[j]) when i==j
    ```
  - The formal parameters include an element of an array and the whole array

    ```
    fun(list[i], list);
    ```
  - Collision between formal parameters and nonlocal variables that are visible

    ```
    int * global;
    void main(){…
    sub(global);…}
    void sub(int * param){
    …}
    ```

Comparisons

	Advantage	Disadv.
Pass-by value	Fast linkage cost and access time for scalar variables	•Multiple storage for parameters •Time for copying values
Pass-by-result	Fast linkage cost and access time for scalar variables	Multiple storage for parameters Time for copying values Acutal param. collision
Pass-by-value-result	Fast linkage cost and access time for scalar variables	Multiple storage for parameters Time for copying values Acutal param. collision
Pass-by-Reference	Passing process is efficient (no copying and no duplicated storage)	Slower accesses (compared to pass-by-value) to formal parameters Un-wanted aliases (access broadened) Potentials for un-wanted side effects

Pass-by-Name (Inout Mode)

- Pass by textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Not part of any widely used language, however, it is used at compile time by the macros in assembly language
  - Can be thought of as inline expansion
- Allows flexibility in late binding

```
inline int max(int a,int b)
{
return (a>b)?a:b;
}
```

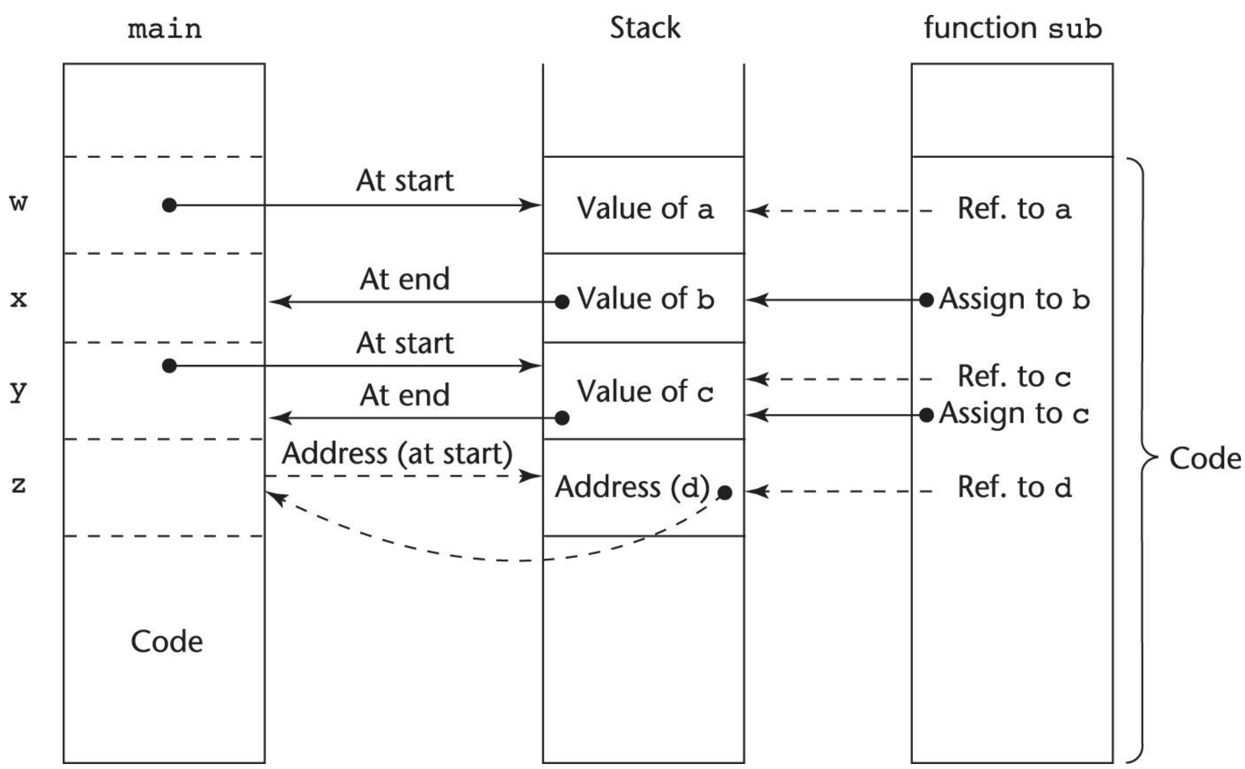
```
Int maximum=max(val1,val2);

Int maximum=((val1>val2)?val1:val2);
```

Implementing Parameter-Passing Methods

- In most languages parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack (regardless of the type of the actual parameter)

Implementing Parameter-Passing Methods



Function header: `void sub(int a, int b, int c, int d)`
Function call in main: `sub(w, x, y, z)`
(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)

Parameter Passing Methods of Major Languages

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All parameters are passed by value
  - Object parameters are passed by reference
- Ada
  - Three semantics modes of parameter transmission: `in`, `out`, `in out`; `in` is the default mode
  - Formal parameters declared `out` can be assigned but not referenced; those declared `in` can be referenced but not assigned; `in out` parameters can be referenced and assigned

Parameter Passing Methods of Major Languages (Cont.)

- Fortran 95+
  - Parameters can be declared to be in, out, or inout mode
- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with `ref`
- PHP: very similar to C#, except that either the actual or the formal parameter can specify ref
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal, FORTRAN 90+, Java, and Ada: it is always required
- ANSI C and C++: choice is made by the user
    - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

Multidimensional Arrays as Parameters

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function
  - In C:
    address(mat[i,j])=address(mat[0,0])+i*number_of_columns+j

```
void fun(int matrix [] [10]){
…}
void main( ){
   int mat[5][10];
   fun(mat);
}
```

Multidimensional Arrays as Parameters: C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

```
void fun(int *mat_ptr, int num_rows, int num_cols);
*(mat_ptr + (row*num_cols) + col) =x ;

#define mat_ptr(r,c)  (*mat_ptr+( (r) * (num_cols) + (c)));
mat_ptr(row,col)=x ;
```

Multidimensional Arrays as Parameters: Ada

- Ada – not a problem
  - Constrained arrays – size is part of the array's type
  - Unconstrained arrays – declared size is part of the object declaration

Multidimensional Arrays as Parameters: Fortran

- Formal parameters that are arrays have a declaration after the header
  - For single-dimension arrays, the subscript is irrelevant
  - For multidimensional arrays, the sizes are sent as parameters and used in the declaration of the formal parameter, so those variables are used in the storage mapping function

Multidimensional Arrays as Parameters: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (`length` in Java, `Length` in C#) that is set to the length of the array when the array object is created

Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  1. Are parameter types checked?
  2. For nested subprograms, what is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Referencing Environment

- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
  – Most natural for dynamic-scoped languages
- *Deep binding*: The environment of the definition of the passed subprogram
  – Most natural for static-scoped languages
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

Parameters that are Subprogram Names: Referencing Environment (Cont.)

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

Shallow binding:
  output : 4

Deep binding:
  output : 1

ad hoc binding:
  output : 3

Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

Generic Subprograms

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- Overloaded subprograms provide *ad hoc polymorphism*
- *Subtype polymorphism* means that a variable of type T can access any object of type T or any type derived from T (OOP languages)
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*
  - A cheap compile-time substitute for dynamic binding

Generic Subprograms (Cont.)

- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - Generic subprograms are preceded by a `template` clause that lists the generic variables, which can be type names or class names

```
template <class Type>
  Type max(Type first, Type second) {
  return first > second ? first : second;
  }
```

Generic Subprograms (Cont.)

- Java 5.0
  - Differences between generics in Java 5.0 and those of C++:

    1. Generic parameters in Java 5.0 must be classes
    2. Java 5.0 generic methods are instantiated just once as truly generic methods
    3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
    4. Wildcard types of generic parameters

Generic Subprograms (Cont.)

- Java 5.0 (Cont.)

```
public static <T> T doIt(T[] list) { … }
```

  - The parameter is an array of generic elements (`T` is the name of the type)
  - A call:

```
doIt<String>(myList);
```

Generic parameters can have bounds:

```
public static <T extends Comparable> T
  doIt(T[] list) { … }
```

The generic type must be of a class that implements the `Comparable` interface

Generic Subprograms (Cont.)

- Java 5.0 (Cont.)
  - Wildcard types

    `Collection<?>` is a wildcard type for collection classes

```
void printCollection(Collection<?> c) {
   for (Object e: c) {
     System.out.println(e);
   }
 }
```

  – Works for any collection class

Generic Subprograms (Cont.)

- C# 2005
  - Supports generic methods that are similar to those of Java 5.0
  - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
  - Another – C# 2005 does not support wildcards

Generic Subprograms (Cont.)

- F#
  - Infers a generic type if it cannot determine the type of a parameter or the return type of a function – *automatic generalization*
  - Such types are denoted with an apostrophe and a single letter, e.g., `'a`
  - Functions can be defined to have generic parameters

```
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

  – `%A` is a format code for any type

  – These parameters are not type constrained

Generic Subprograms (Cont.)

- F# (continued)
  - If the parameters of a function are used with arithmetic operators, they are type constrained, even if the parameters are specified to be generic
  - Because of type inferencing and the lack of type coercions, F# generic functions are far less useful than those of C++, Java 5.0+, and C# 2005+

User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- A Python example

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

Use: To compute `x + y`, `x.__add__(y)`

Closures

- A *closure* is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - A static-scoped language that does not permit nested subprograms doesn't need closures
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
  - To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

Closures (Cont.)

- A JavaScript closure:

```
function makeAdder(x) {
  return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) +
               "<br />");
document.write("add 5 to 20: " + add5(20) +
               "<br />");
```

  – The closure is the anonymous function returned by `makeAdder`

Closures (Cont.)

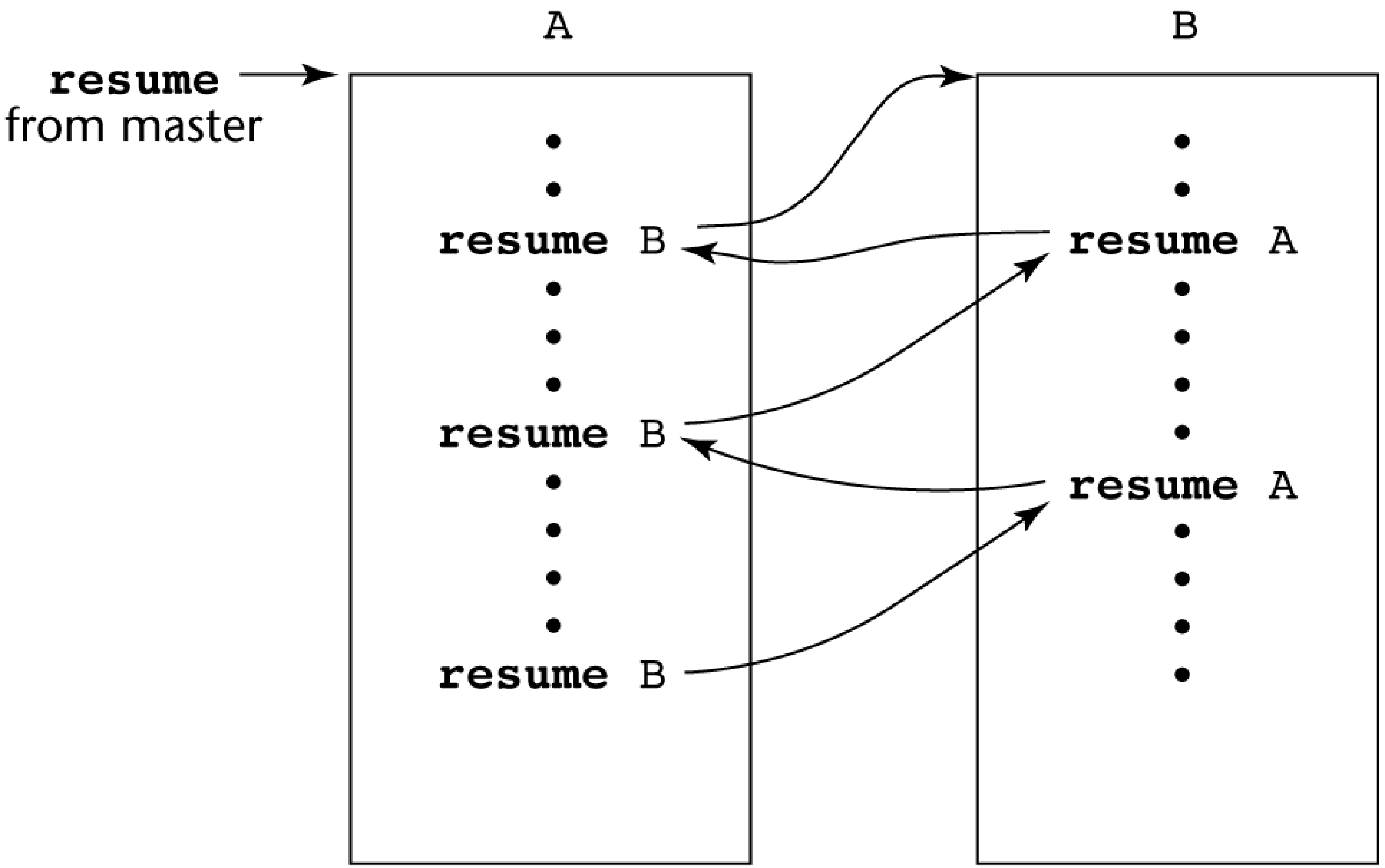
- C#
  - We can write the same closure in C# using a nested anonymous delegate
  - `Func<int, int>` (the return type) specifies a delegate that takes an **`int`** as a parameter and returns and **`int`**

```
static Func<int, int> makeAdder(int x) {
   return delegate(int y) {return x + y;};
}
...
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

Coroutines

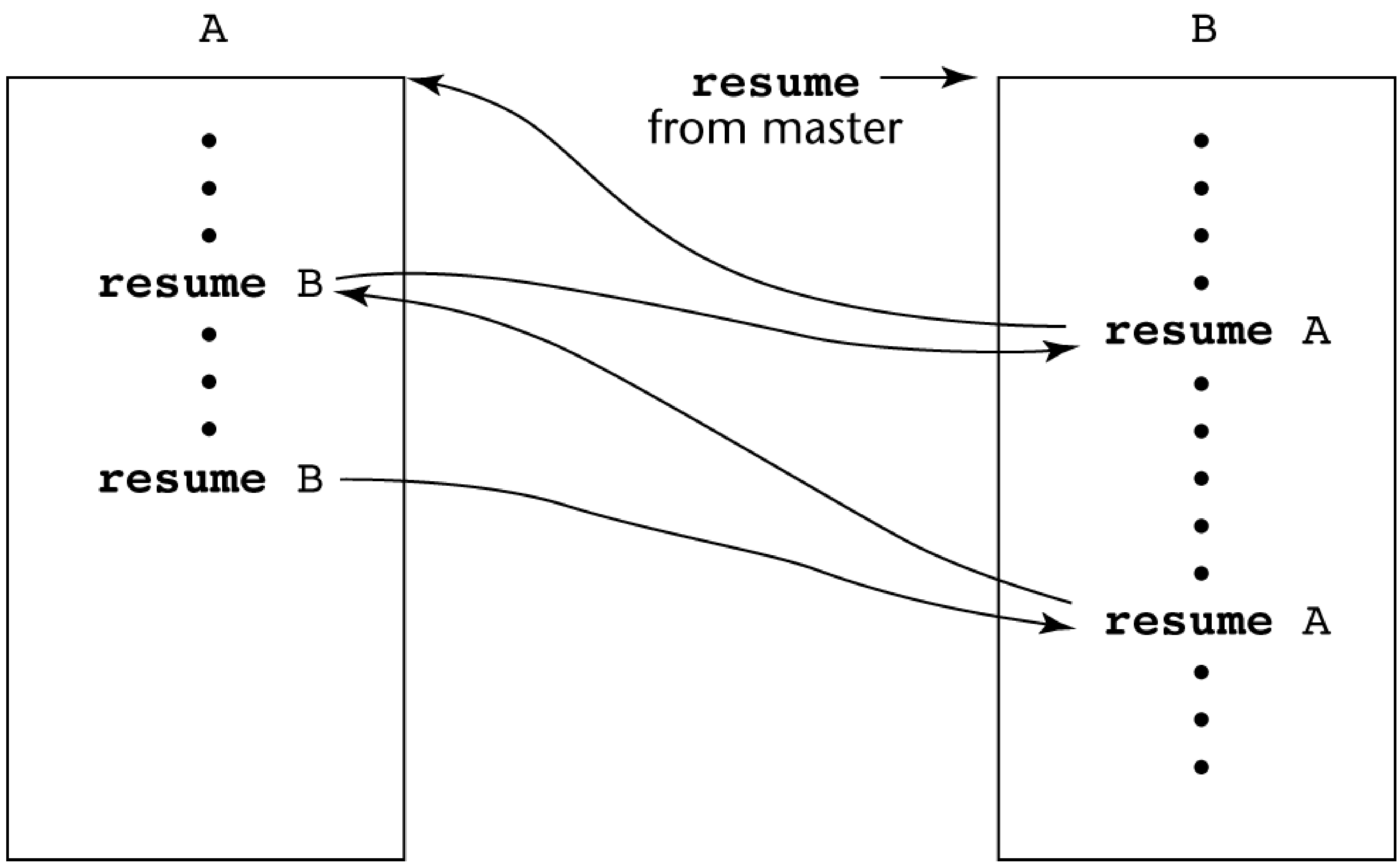
- A *coroutine* is a subprogram that has multiple entries controlled by the coroutine themselves
  - supported directly in Lua
- Also called *symmetric control since* caller and called coroutines are on a more equal basis
- The invocation of a coroutine is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

Coroutines Illustrated: Possible Execution Controls



(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops