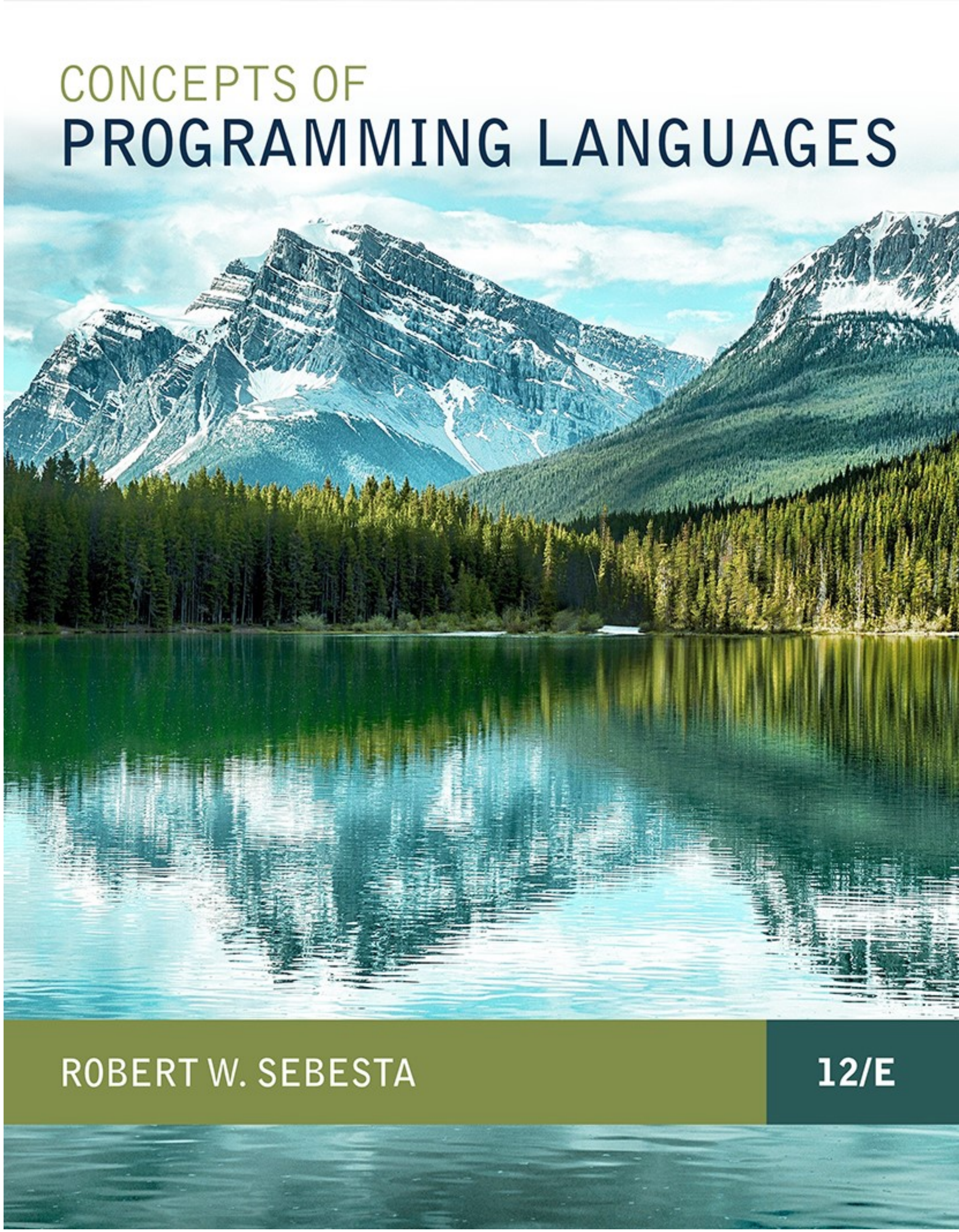
Chapter 9

Subprograms



ISBN 0-321-49362-1

Chapter 9 Topics

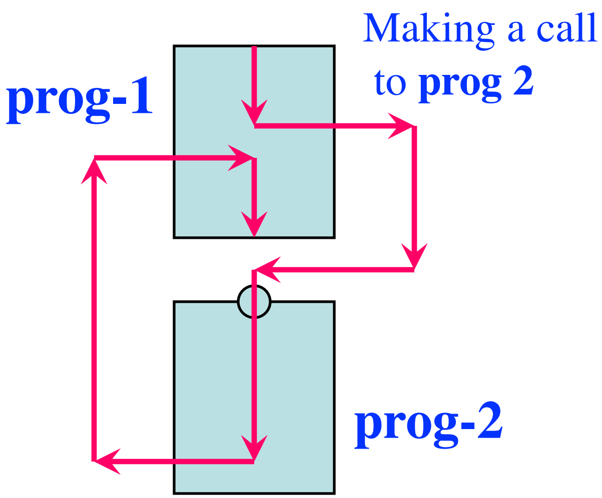
- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Design Issues for Functions
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded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

Introduction

- Two fundamental abstraction
  - Process abstraction
    - Emphasized from early days
    - E.g., sub-program, hiding details of execution
  - Data abstraction
    - Emphasized in the1980s
    - E.g., object, Instance of Abstract Data Type (**ADT**)

Fundamentals of Subprograms

- Each subprogram
  - has a single entry point
- The calling program
  - caller, prog-1, is suspended during execution of the called subprogram (callee, prog-2)
- Control
  - returns to the caller when the callee's execution terminates



Procedures and Functions

- Two categories of subprograms:
  - *Procedures:*
    - collection of statements that define parameterized computations
  - *Functions*:
    - structurally resemble procedures but are semantically modeled on mathematical functions
    - expected to produce no side effects
    - In practice, functions have side effects

Basic Definitions

- *Subprogram header*:
  - include the name (e.g., setDate), the kind of subprogram, and the formal parameters

- E.g., C++:

```
class Date {
    private:
        int year;    int month;    int date:
    public:
        Date(int year, int month, int date);
        void setDate(int year, int month, int date);
        int getYear() { return year;}
        int getMonth() {return month;}
        int getDate() { return date;}
};
```

Basic Definitions

- *Parameter profile* (aka *signature*) of a subprogram:
  - the number, order, and types of its parameters
    - *Formal parameter:* a dummy variable listed in the subprogram header and used in the subprogram
    - *Actual parameter:* represents a value or address used in the subprogram call statement
- *Protocol*:
  - a subprogram's parameter profile and return type (if any)

Actual/Formal Parameter Mapping

- 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
  - E.g., Java: int result = sum(6, 9);
- Keyword parameters
  - name of the formal parameter to which an actual parameter to be bound is specified with actual parameter
  - *Advantage*:
    - Parameters can appear in any order, thereby avoiding parameter correspondence errors
  - *Disadvantage*:
    - User must know the formal parameter's names
  - E.g., Python: sumsub(length=my_length, sum =my_sum)

Formal Parameter Default Values

- In certain languages (C++, Python, Ruby, PHP)
  - formal parameters can have default values, if **no** actual parameter is passed
  - In C++, **default** parameters must appear last because parameters are positionally associated (no keyword parameters)

  - E.g., C++, int sum(int acc1, int acc2, int acc3 = 10);
    int result = sum(10, 20);
    // no actual parameter for acc3

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**

E.g., C#: public void useParams(params int[] list) {…}

- In Lua
  - a variable number of parameters is represented as a formal parameter with **three periods**;

```
E.g., Lua:  function f (a, b, ...) end
            f(3, 4, 5, 8)
            --  a=3, b=4, arg={5, 8; n=2}
            --  arg: hidden parameter
```

Chapter 9 Topics

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

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprogram be generic?
- If the language allows nested subprograms, are closures supported?

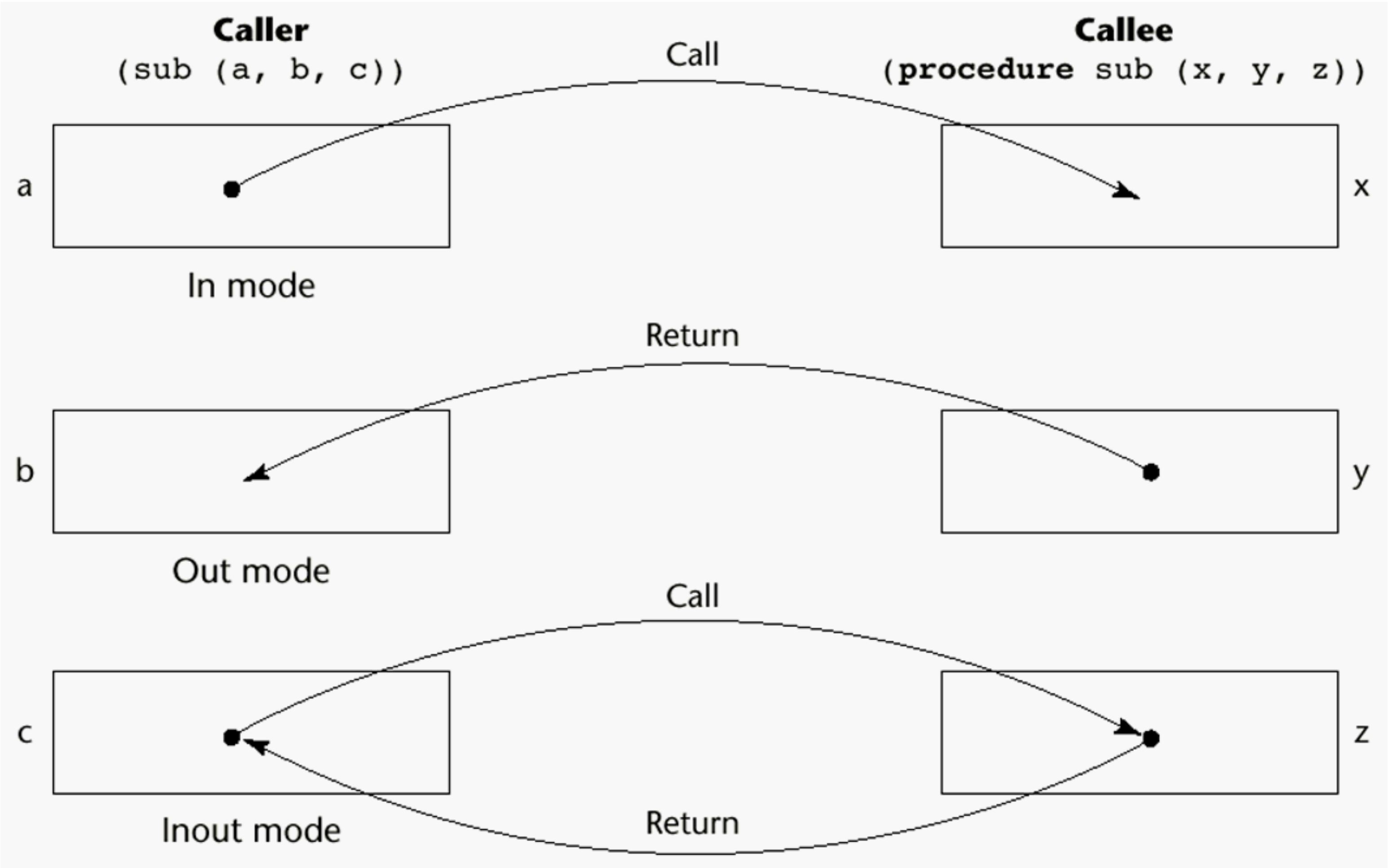
Chapter 9 Topics

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

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Conceptual Models of Transfer

- Physically move a value
  - Actual value
- Move an access path to a value
  - Pointer or reference

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path
  - *Disadvantages*
    - if by physical move: additional storage is required (stored twice) and the actual move can be costly (for large parameters)
    - if by access path method: must write-protect in the called subprogram 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 in callee acts as a *local variable*, its value is transmitted to caller's *actual* parameter when control is returned to the caller
  - Require extra storage location and copy operation

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage

Pass-by-Reference (Inout Mode)

- Pass an access path (i.e., 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
  - Potentials for unwanted side effects (collisions)

Pass-by-Name (Inout Mode)

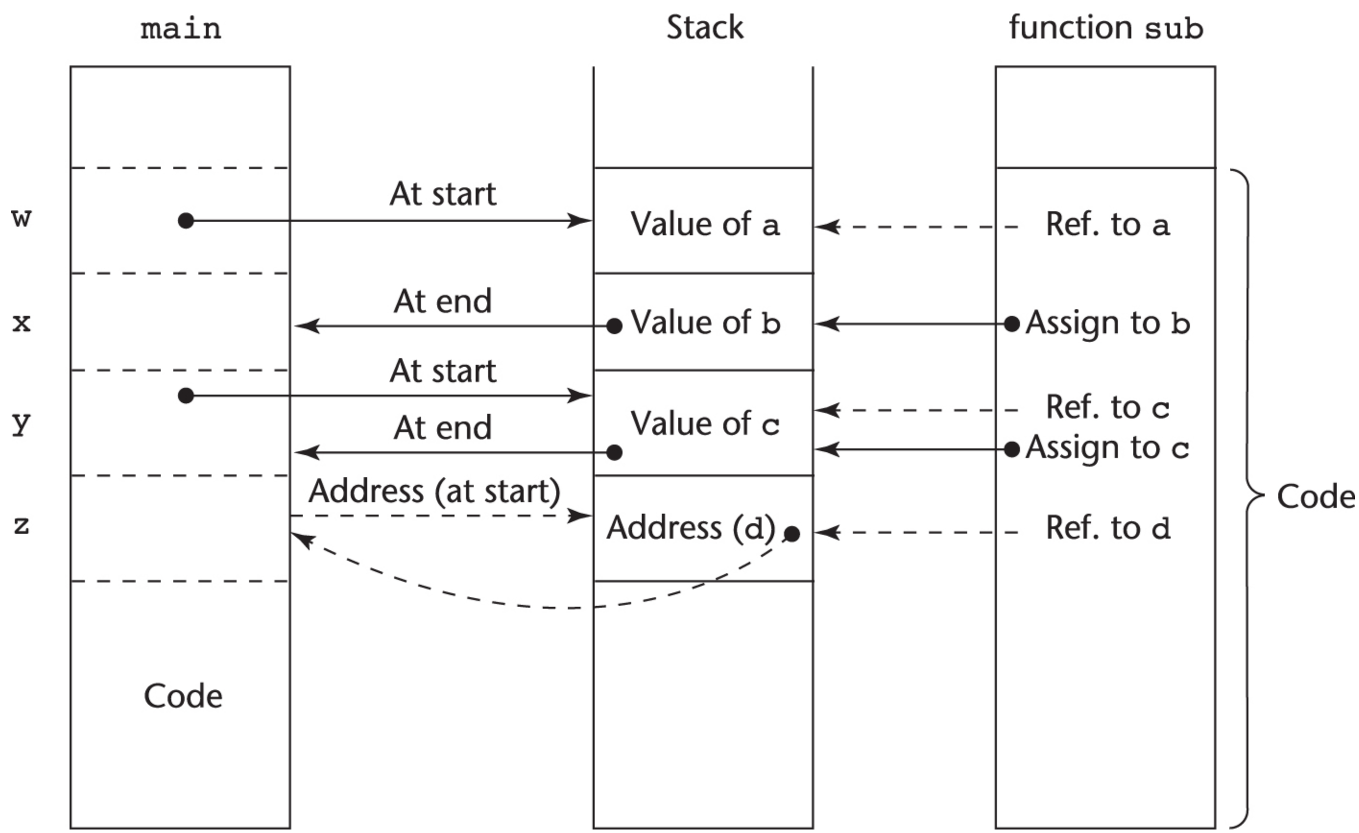
- By textual substitution
- Formal parameters are bound to an access **method** at the time of the call, but actual binding to a value or address takes place at the **time** of a reference or assignment
- Allows flexibility in late binding
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated
- E.g., JavaScript

```
function apply(a, b, f) { return f(a, b); }
function add(a, b) { return a + b; }
var result = apply(2, 4, add); // result is now 6
```

Implementing Parameter-Passing Methods

- In most languages parameter communication takes place thru the run-time memory stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack

Implementing Parameter-Passing Methods



In Ada: `procedure sub(w: in integer, x: out integer, y: in out integer)`
`sub(a, b, c); -- in procedure main`

In C: `void sub(int* z)...`
`sub(&d); // in main()`

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 non-object parameters are passed by value
  - An object parameter is passed by a reference value (reference)—pointer to the object
  - ref: https://docs.oracle.com/javase/specs/ @ 4.3.1: Objects

Parameter Passing Methods of Major Languages

- 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`**
  - **E.g.,** static void method1(ref int i) { … }
    main() { int val=1;… method1(ref val); … }
- PHP:
  - very similar to C#, except that either the actual or the formal parameter can specify **ref**

Parameter Passing Methods of Major Languages

- Perl:
  - all **actual parameters** are implicitly placed in a predefined/default array named `@_`
  - E.g., **Perl**;

```
sub printAry {  my @ary = @{$_[0] };
                print "@ary\n";        }
my @array = ('csci', '6221');
printAry(\@array);
```

- Python and Ruby
  - use pass-by-assignment (all data values are objects);
  - the actual is assigned to the formal.

E.g., **Python**:

```
def testFunc(deptName, classNum = "9999"):
        print "Dept Name: ", deptName
        print "Class Num: ", classNum
        return;
testFunc(deptName="CSCI", classNum="6221");
```

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: it is always required
- ANSI C and C++: choice is made by the user
  - e.g., double sin(double x) { … } ;
    double result; int count; result = sin(count);
- Other 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

Multidimensional Arrays as Parameters: C and C++

- Programmer is required to include the declared sizes of all, except for the **first** subscript in the formal parameter
- E.g., in C: void func(int matrix [][10],....) { … };
  void print3d(int DD[][3][3]) { ...};

Multidimensional Arrays as Parameters: Java and C#

- 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
- E.g., Java: int[][] multi = new int[5][10];

Chapter 9 Topics

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

Parameters that are Subprogram Names

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

Note: ch5: *Referencing environment* of a statement is the collection of all names that are visible in the statement

Parameters that are Subprogram Names: Referencing Environment

- *Environment for executing the passed subprogram:*
  - *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

```
function   sub1()  {
    var  x;
    function  sub2()  {
        alert(x); // print x
    };
    function  sub3()  {
        var  x;
        x = 3;   // ad hoc
        sub4(sub2);
    };
    function  sub4(subx) {
        var  x;
        x = 4;   // shadow
        subx();
    };
    x = 1;   // deep
    sub3();
};
```

- Shadow binding
  - x in sub2() is bound to the local x = 4 in sub4()
- Deep binding
  - x in sub2() is bound to the local x = 1 in sub1()
- Ad hoc binding
  - x in sub2() is bound to the local x = 3 in sub3()

Notes: **JavaScript**: **Deep** binding; print x in **sub2**()

Call sequences: **sub1() -> sub3() -> sub4()** -> sub2()

Chapter 9 Topics

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

Calling Subprograms Indirectly

- when several possible subprograms to be called
  - the correct one on a particular run of the program is not know until execution
  - In C and C++, such calls are made through function pointers
  - E.g., in C:

```
int func1(int, int);
int (*ptrFunc1)(int, int) = func1;
ptrFunc1(5, 10); //call func1
```

Calling Subprograms Indirectly

- In C#, method pointers are implemented as objects called *delegates*
  - E.g., delegate declaration:

```
public delegate int Change(int x);
```

  - This delegate type, named `Change`
    - can be instantiated with any method that takes an `int` parameter returns an `int` value

E.g., A method: `static int fun1(int x) { … }`

Instantiate: `Change chgfun1 = new Change(fun1);`

Can be called with: `chgfun1(12);`

Chapter 9 Topics

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

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of returned values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except **arrays** and **functions**
  - C++ is like C but also allows user-defined types
  - Java and C# methods can return any type (but because methods are **not** types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
  - Lua allows functions to return multiple values

Chapter 9 Topics

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

Overloaded Subprograms

- An *overloaded subprogram*
  - has the same name as another subprogram in the same referencing environment
  - every version of an overloaded subprogram has a unique **protocol**.
  - E.g., Java: class CS { void foo(int a) {..} foo(double d) {...} }
- C++, Java, C#, and Ada
  - include predefined overloaded subprograms
- In Ada
  - 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

Chapter 9 Topics

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

Generic Subprograms

- A *generic* or *polymorphic subprogram:*
  - takes parameters of different types on different activations
- *Subtype polymorphism*:
  - a variable of type T can access any object of type T or any type derived from T (OOP languages)
- *Parametric polymorphism:*
  - a subprogram takes a generic parameter that is used in a type expression
  - E.g., Java:

```
static<T> void arrToCol(T[]  arr, Collection<T> col){ … };
```

Generic Subprograms: C++

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

E.g.,
```
template <class Type>
Type max(Type first, Type second) {
   return first > second ? first : second;
}
int maxVal = max(100, 200);
```

Generic Subprograms: Java

Java:

- Differences btw generics in Java and those of C++:
  1. Generic parameters in Java must be classes
  2. Java 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

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

Generic Subprograms: Java

- Java (continued)

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

  - parameter T: 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: Java

- Java (continued)
  - 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: C#

- C#
  - Supports generic methods similar to Java
  - Difference:
    1. **actual** type parameters in a call can be omitted if the compiler can infer the unspecified type
       - E.g., void Exclass.ExMethod(int required; [string optStr =" defaultStr"], [int optInt=10])
       - Exclass anEx = new Exclass ();
       - anEx.ExMethod(**1, "one", 1**);
       - anEx.ExMethod(**2, "two"**);
       - anEx.ExMethod(**3**);
    2. C# does not support wildcards

Generic Subprograms: F#

- 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: F#

- 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
  - E.g., let adder x y = x + y;;   // return int

    adder 2.5 3.6;;   // illegal
  - Because of type inferencing and the lack of type coercions, F# generic functions are far less useful than those of C++, Java, and C#

Chapter 9 Topics

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

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)`

Chapter 9 Topics

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

Closures

- A *closure*
  - is a subprogram and the referencing environment where it was defined
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
    - A static-scoped language that does not permit nested subprograms doesn’t need closures
  - 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: JavaScript

- A JavaScript closure:

```
function makeAdder(x) {
  return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);  // x = 10
var add5 = makeAdder(5);    // x = 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: C#

- C#
  - We can write the same closure in C# using an anonymous delegate.

e.g.,

```
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));
```

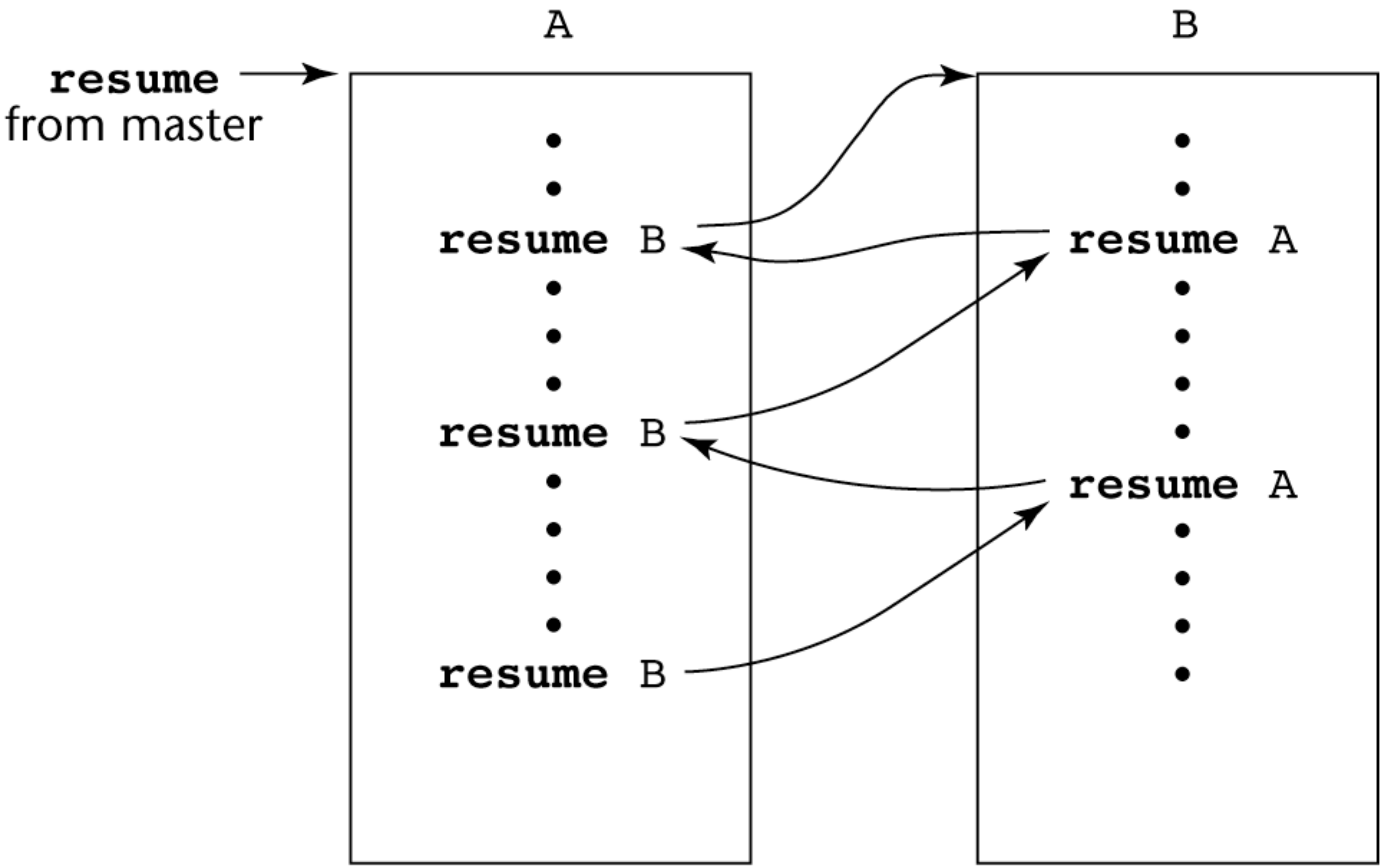
Chapter 9 Topics

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

Coroutines

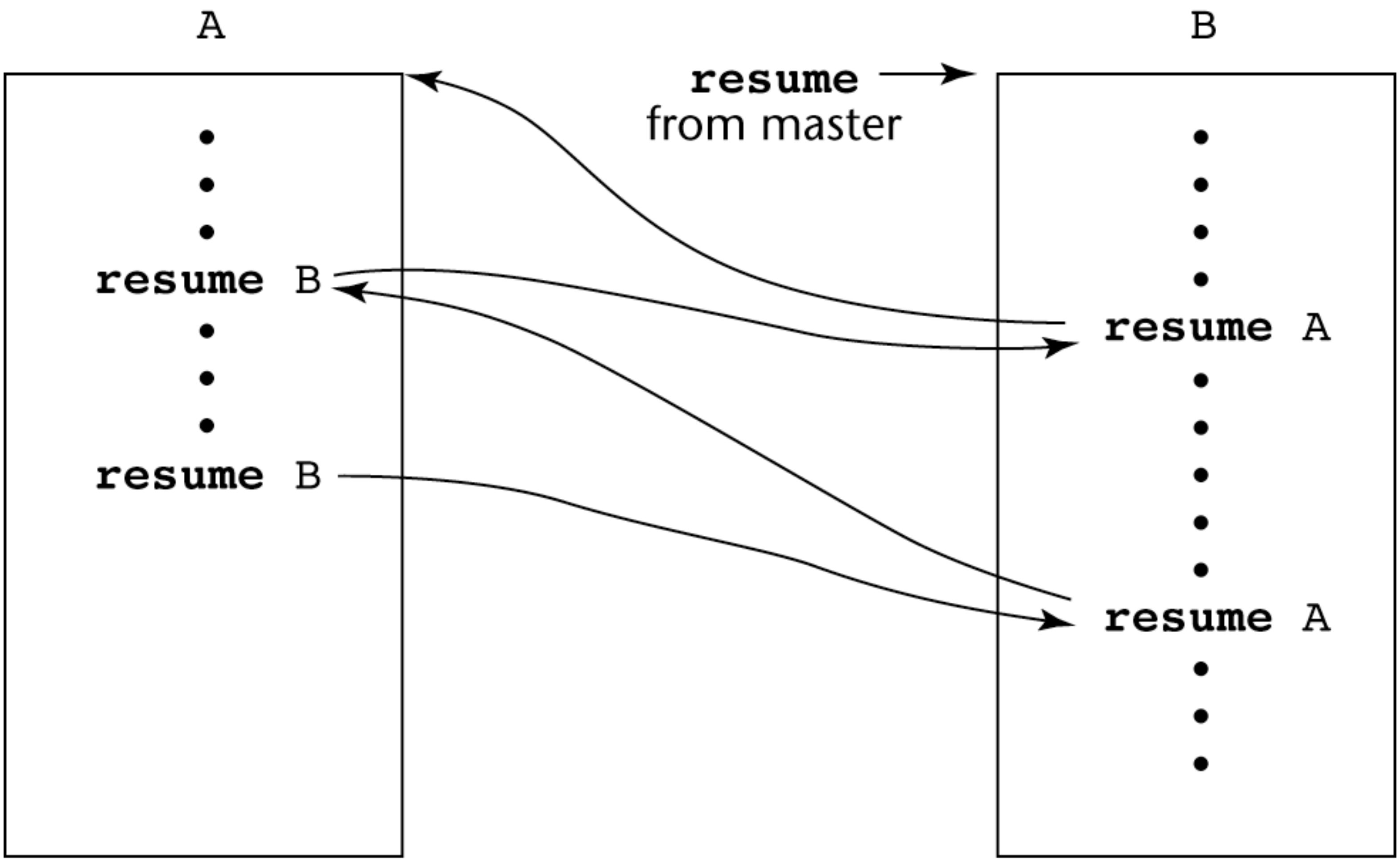
- A *coroutine*
  - a subprogram has multiple entries and controls them itself – supported directly in Lua
  - also called *symmetric control:* caller and callee coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent 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