

# Subprograms

# Topics

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

# Introduction

- Two fundamental abstraction facilities
  - Process abstraction
    - Emphasized from early days
    - Details of how a computation or process is done are “abstracted away” by the procedure call
  - Data abstraction
    - Emphasized in the 1980s

# Fundamentals/Definitions of Subprograms (quick review)

function signature: → int doCalc(int aNumber);      called *prototype* in C/C++  
number, order &  
types of parameters  
function *protocol*:  
includes return type  
(aka declaration)

int main() {  
...  
result=doCalc(5);  
...  
}

actual argument      function call, program suspends during execution

subprogram header

single entry point → int doCalc(int aNumber) {  
int x=aNumber\*2;  
return x+3;  
}

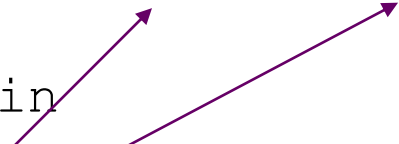
} subprogram definition

# Actual/Formal Parameter Correspondence

- Positional
  - Safe and effective, convenient if list is short

```
int doIt(int a, double b);
```

```
// in main  
doIt(5, 4.6);
```



- Keyword
  - Parameters can appear in any order

```
summer(length=my_length,  
        list=my_array,  
        sum=my_sum)
```

- May be able to mix:

```
summer(my_length,  
        sum=my_sum,  
        list=my_array)
```

# Formal Parameter Default Values

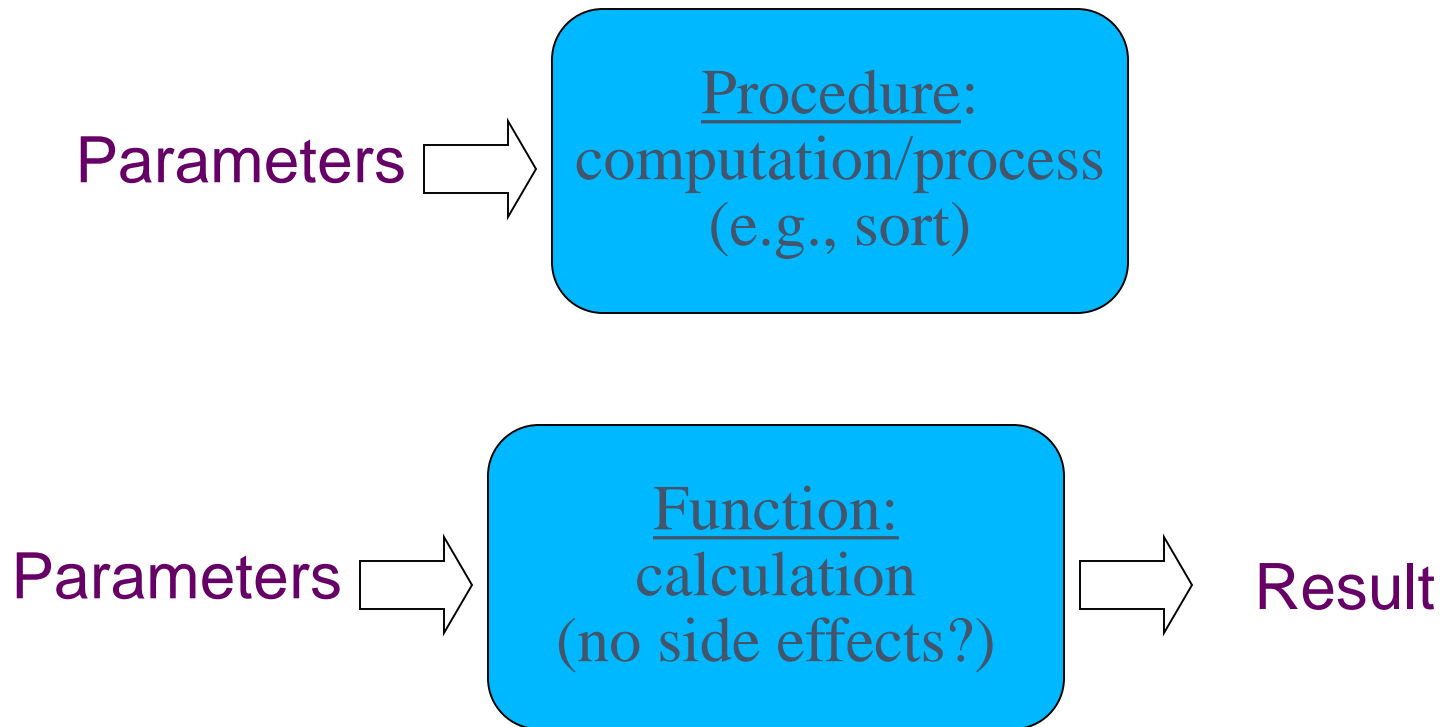
- In certain languages (e.g., C++, Ada, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed)
  - In C++, default parameters must appear last because parameters are positionally associated
- C# methods can accept a variable number of parameters as long as they are of the same type

# Formal Parameters (cont...)

- In Ruby the initial parameters which correspond to the formal parameters can be followed by a list of key->value pairs placed in an anonymous hash. This is a substitute for keyword parameters. The hash item can be followed by an Array object.
- Python also has formal parameters which can be followed by an array and a hash.

# Procedures and Functions

- There are two categories of subprograms





# Design Issues for Subprograms

- Can subprograms be nested?
- What parameter passing methods are provided?
- Are parameter types checked?
- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- Can subprograms be overloaded?
- Can subprograms be generic?

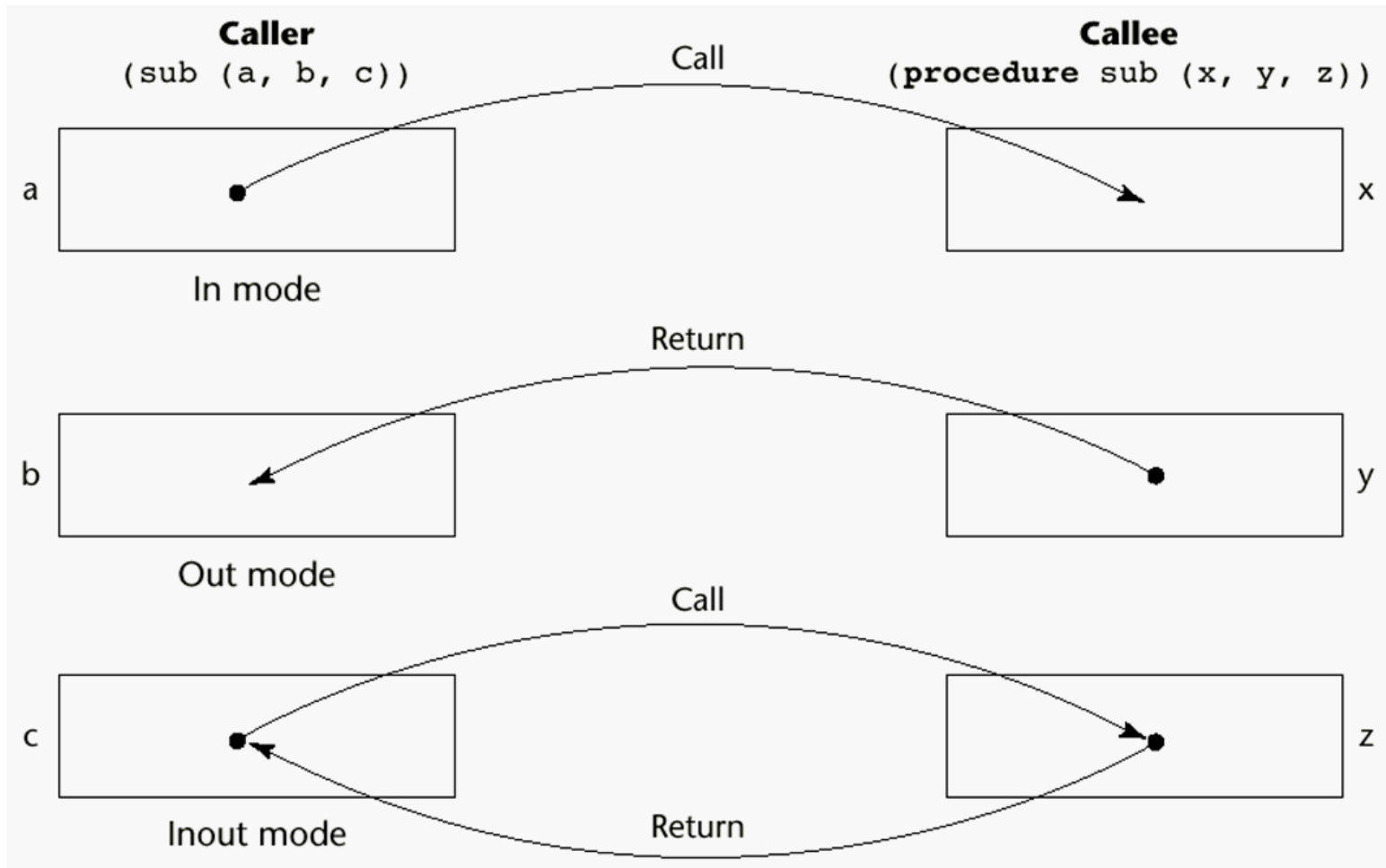
# Nested Subprograms

- Not allowed in C and descendants
- Allowed in Algol 68, Pascal, Ada
- More recently included in JavaScript, Python and Ruby

# Parameter Passing Methods

- Ways in which parameters are transmitted to and/or from called subprograms
  - Pass-by-value
  - Pass-by-result
  - Pass-by-value-result
  - Pass-by-reference
  - Pass-by-name
- Three semantic models for passing:
  - in, out, inout mode

# Semantic Models of Parameter Passing



# 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 but not recommended (enforcing write protection is not easy)
  - When copies are used, additional storage is required
  - Storage and copy operations can be costly

# 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
  - Require extra storage location and copy operation on return

# Pass-by-Result (Out Mode)


- Potential problem: `sub(p1, p1);` **whichever formal parameter is copied back will represent the current value of p1**

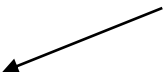
```
void Fixer(out int x, out int y) {  
    x = 17;  
    y = 35;  
}
```

```
f.Fixer(out a, out a);  
// what will a contain?
```

# Pass-by-Result (Out Mode)

- Must also consider when address of out parameter is determined

```
void DoIt(out int x, int index) {  
    x = 17;  
    index = 42;  
}  is address calculated here?
```

```
...  
sub = 21;  is address calculated here?  
f.DoIt(list[sub], sub); // what is updated?
```



# 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, copy is made of actual parameter. Value is transmitted back at end of subprogram.
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

# Pass-by-Reference (Inout Mode)

- Pass an access path
- Also called pass-by-sharing
- Passing process is efficient (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters due to indirect addressing
  - Potentials for unwanted side effects
  - Unwanted aliases (access broadened)

How to prevent?  
(in C++)

Why do we care?

# Pass by Reference – alias examples

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

```
fun (total, total);
```

```
// first & second now aliases
```

```
fun (list[i], list[j]);
```

```
// if i == j, aliases
```

# Pass-by-Name (Inout Mode)

- By textual substitution
- Not part of any widely used language
- Used at compile time by macros in assembly languages and for generic parameters of generic subprograms

Sub (X)	Sub (Y)	textual substitution:
Z $\leftarrow$ X * 2		Z $\leftarrow$ Y * 2
X $\leftarrow$ Z + 3		Y $\leftarrow$ Z + 3

Must use referencing environment for Y.  
Treat parameters as parameterless subprograms called *thunks*.

# Implementing Parameter-Passing Methods

- In most languages parameter communication takes place through the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack
- A subtle but fatal error can occur with pass-by-reference and pass-by-value-result: a formal parameter corresponding to a constant can mistakenly be changed

# Parameter Passing Methods of Major Languages

- Fortran
  - Always used the inout semantics model
  - Before Fortran 77: pass-by-reference
  - Fortran 77 and later: scalar variables are often passed by value-result
- 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 (no changes can be made to primitives via reference parameters)
  - Object parameters are references

# Parameter Passing Methods of Major Languages (continued)

- 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
- 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#

# Parameter Passing Methods of Major Languages (continued)

- Python and Ruby
  - pass-by-assignment. Actual parameter value is assigned to the formal parameter. All actual parameters are references, so similar to pass-by-reference, except that many objects are immutable.
  - `x = x + 1`; creates a new object `x` with value `(x + 1)`. So no effect on actual parameter in caller.
  - Changes to an array element will be seen.



# 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 C99 and C++: function parameters are type-checked (avoid with ellipsis in parameter list)
- Relatively new languages Perl, JavaScript, Python, Ruby and PHP do not require type checking

# 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
  - Example: numerical integration method samples function at different points... need function to evaluate
- Issues:
  - 1.Are parameter types checked?
  - 2.What is the correct referencing environment for a subprogram that was sent as a parameter? (only matters if nested subprograms allowed)

# Parameters that are Subprogram Names:

## Parameter Type Checking

- C and C++: functions cannot be passed as parameters but pointers to functions can be passed; parameters can be type checked because type of pointer is function's protocol
- FORTRAN 95 type checks
- Ada does not allow subprogram parameters; a similar alternative is provided via Ada's generic facility

# Parameters that are Subprogram Names: Referencing Environment

- *Deep/shallow binding makes sense only when a procedure can be passed as an argument to a function.*
- *Shallow binding:* The Shallow binding binds the environment at the time a procedure is actually called.
- *Deep binding:* Deep binding binds the environment at the time a procedure is passed as an argument.

# Environment Binding Example

```
def sub1():  
    x = 1  
    def sub2():  
        print x  
    def sub3():  
        x = 3  
        sub4(sub2)  
    def sub4(f):  
        x = 4  
        f()  
    sub3()
```

Now if **sub1()** is called, what will be the three different cases generated for the three different types of binding:

**Deep binding** : Takes the environment of the parent function. So in the example, no matter which sub-function gets called the value of x would be 1.

**Shallow binding** : Takes the environment of the “final” calling function. Here, ultimately the final function to call sub2 is sub4, so sub2 would take the value of x initialized in sub4 and would print 4.

**Python** supports **deep binding**, so the answer of the above code will be **1**.

# 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

# Overloaded Subprograms (cont)

- What happens if two or more methods can be matched through coercions?
- Try the following:

```
#include <iostream>
using namespace std;
Class A
{private:
    A() {}
public:
    A(int x = 0) {}
};
int main()
{   A a;
    return 0;}
```

- Having a constructor with default arguments and a default constructor (no arguments) results in ambiguity and compilation error.



# Generic Subprograms

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations.
- Overloaded subprograms provide **ad hoc polymorphism**.
- 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**.

# Parametric polymorphism: C++

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

- The above template can be instantiated for any type for which operator > is defined

```
int intMax = max(4, 6);
double dblMax = max(7.8, 3.8);
```

# Generics in Java

- Generic parameters must be classes
- Generics mean parameterized types. The idea is to allow type (Integer, String, ... etc, and user-defined types) to be a parameter to methods, classes, and interfaces.
- Using Generics, it is possible to create classes that work with different data types.
- Generics in Java is similar to templates in C++.
- Like C++, we use <> to specify parameter types in generic class creation.

# Generics in Java (cont)

Sample generic method definition:

```
public static <T> T DoIt(T[] list) { ... }
```

Sample generic method call:

```
DoIt<int>(myList) ;
```

Sample bounded method definition:

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

Can have multiple restrictions separated by &

# Generics in Java (cont)

- Also supports *wildcard types*:

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

```
Collection<?> c = new ArrayList<String>();  
// illegal to add any element except a String
```

- The question mark (?) is known as the wildcard in generic programming. It represents an unknown type.
- The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type.

*Generics in C# are similar but no wildcards.*

# Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions (can be done with pointers)
  - C++ is like C but also allows user-defined types
  - Ada, Python and Ruby allow any type
  - Java and C# do not have functions but methods can have any type

# User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python and Ruby.
- C++ Example (Ref. C++ Operator Overloading Slides).

# Coroutines

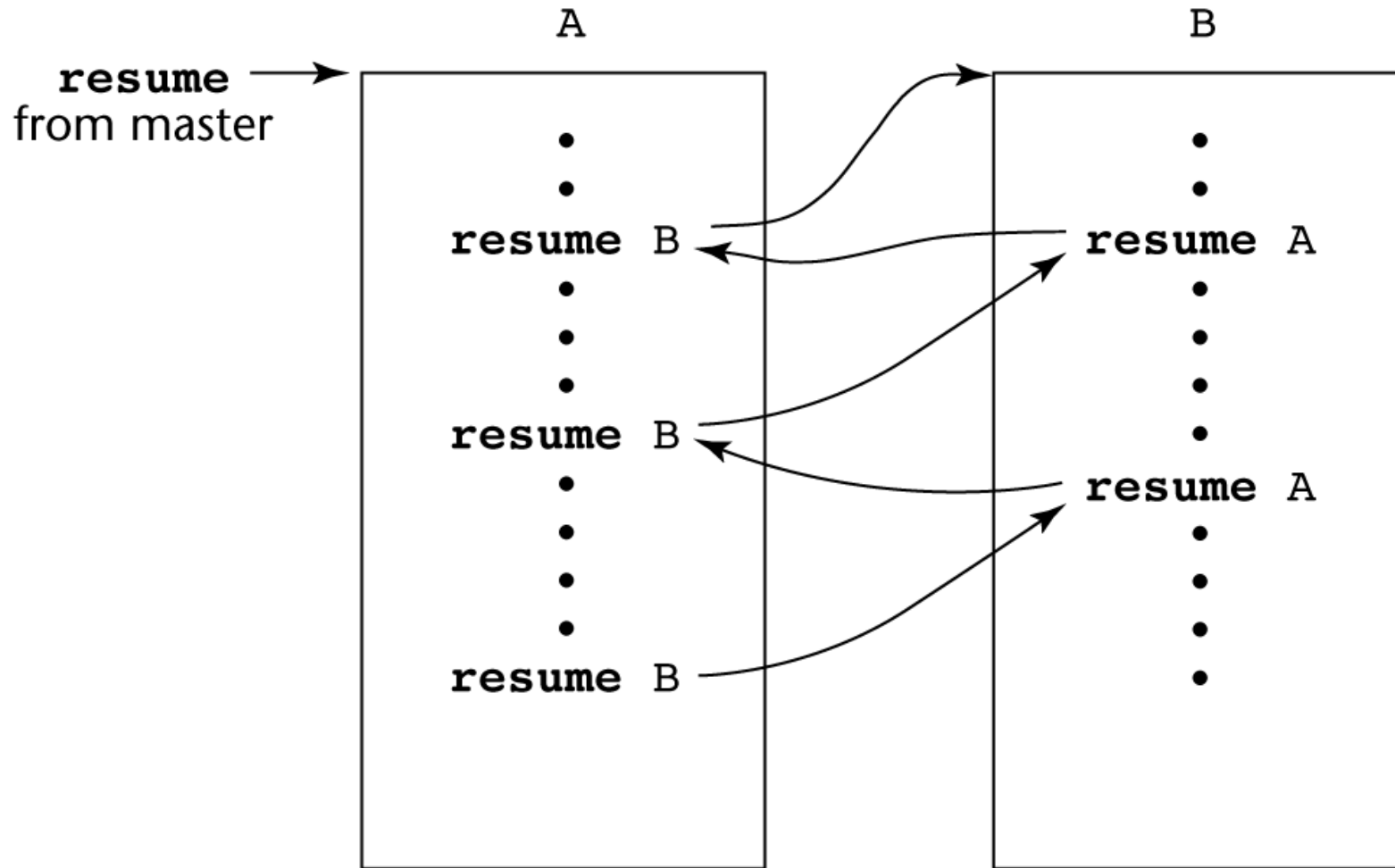
- Read about them...
  - <http://defoe.sourceforge.net/defoe/coroutines.html>
  - <http://www.csl.mtu.edu/cs4411/www/NOTES/non-local-goto/coroutine.html>
  - <http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html>
  - <http://c2.com/cgi/wiki?CoRoutine>
  - <http://msdn.microsoft.com/msdnmag/issues/03/09/CoroutinesinNET/default.aspx>
- What are they?
- What programs that you know implement coroutines?
- What types of problems might they be useful for?



# Coroutines

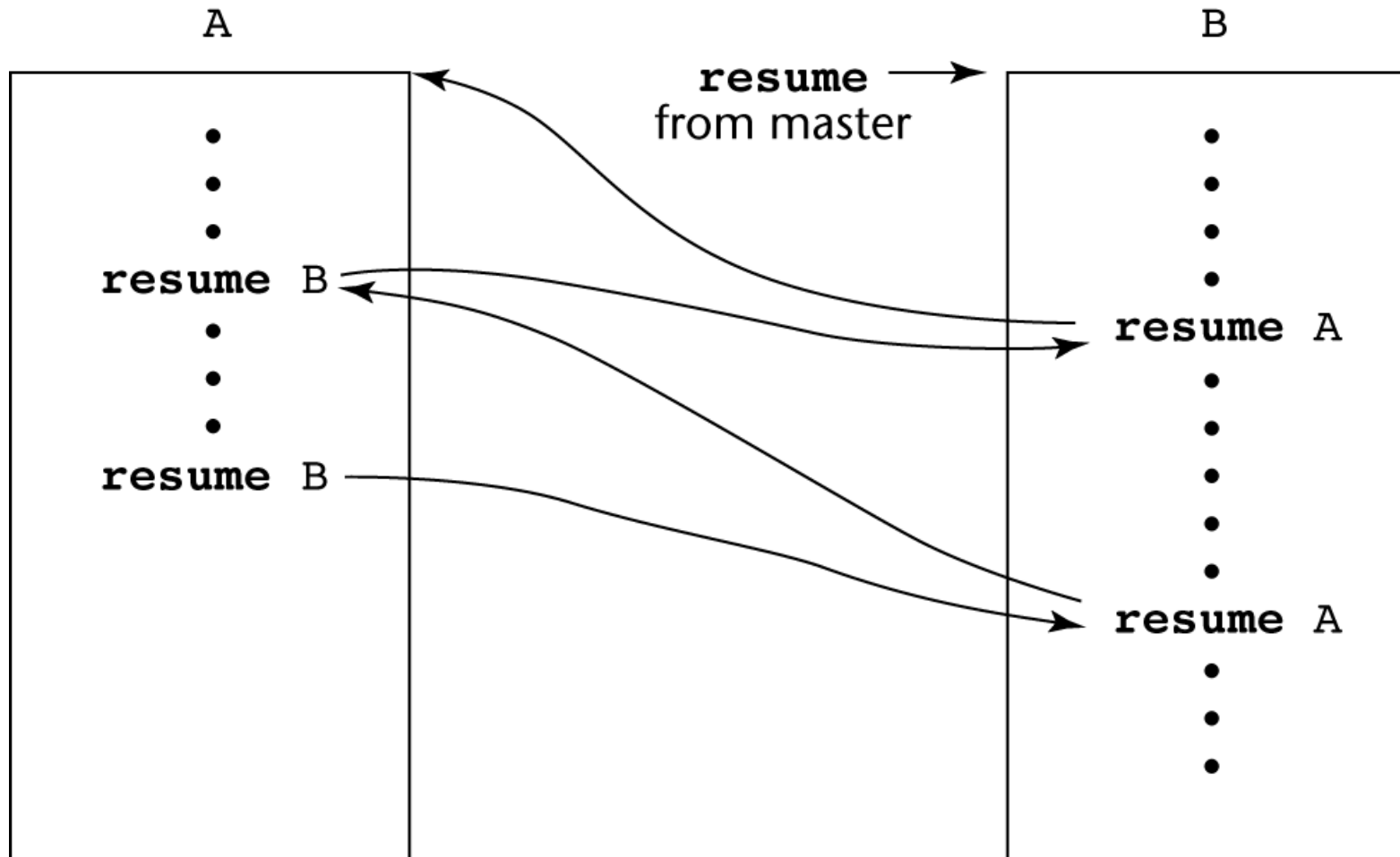
- A *coroutine* is a subprogram that has multiple entries and controls them itself (maintain status)
- Also called *symmetric control*: caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent calls 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



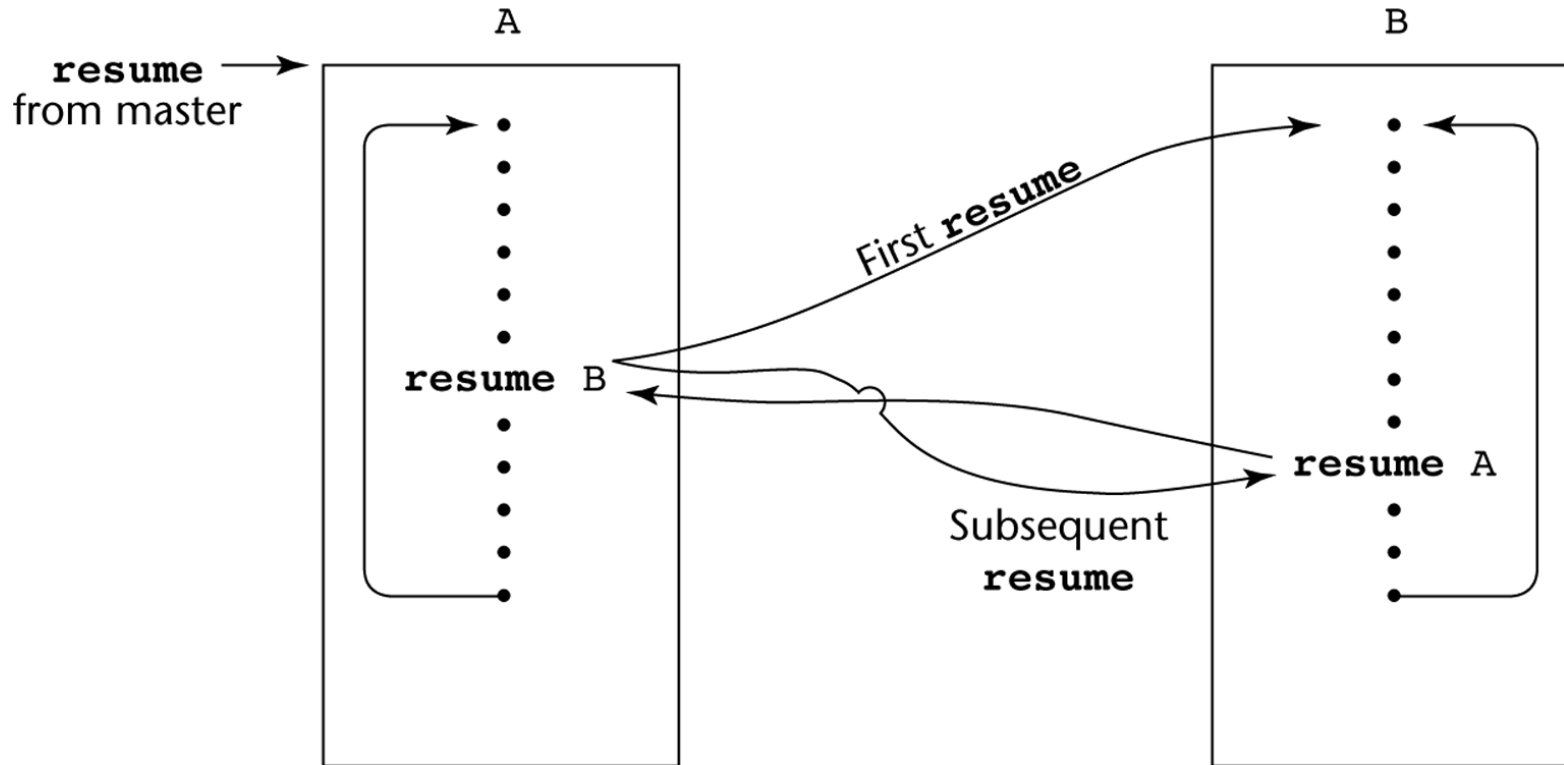
(a)

# Coroutines Illustrated: Possible Execution Controls



(b)

# Coroutines Illustrated: Possible Execution Controls with Loops



# Coroutines (cont...)

- Coroutines originated as an assembly-language technique.
- Supported in some high-level languages, Simula and Modula-2 being two early examples.
- Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, iterators, infinite lists, and pipes (as per Wikipedia).

# Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A coroutine is a special subprogram with multiple entries