MODULE 3
Subprograms

Subprograms

- Introduction
- Design Issues of Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Subprograms as Parameters
- Overloaded Subprograms
- Closures
- Co-routines

Introduction

- Two fundamental abstraction facilities can be included in a programming language: process abstraction and data abstraction.
- In the early history of highlevel programming languages, only process abstraction was included.
- Process abstraction, in the form of **subprograms**, has been a central concept in all programming languages.
- Reusing collections of statements is written as a subprogram.
- This reuse results in several different kinds of savings, primarily memory space and coding time.
- This increases the readability of a program by emphasizing its logical structure while hiding the low-level details.

- Fundamental Characteristics of Subprograms:

  1. A subprogram has a single entry point

  2. The caller is suspended during execution of the called subprogram

  3. Control always returns to the caller when the called subprogram's execution terminates

Design Issues for Subprograms

- Subprograms are complex structures in programming languages

1. What parameter passing methods are provided?

- The choice of one or more parameter-passing methods that will be used.

2. Are parameter types checked?

- Whether the types of actual parameters will be type checked against the types of the corresponding formal parameters.

3. Are local variables are statically or dynamically allocated

- The nature of the local environment of a subprogram dictates to some degree the nature of the subprogram.

4. What is the referencing environment of a passed subprogram?

- If subprogram names can be passed as parameters and the language allows subprograms to be nested, there is the question of the correct referencing environment of a subprogram that has been passed as a parameter.

5. Whether subprograms can be overloaded or generic.

- An **overloaded subprogram** is one that has the same name as another subprogram in the same referencing environment.
- A **generic subprogram** is one whose computation can be done on data of different types in different calls.

6. If the language allows nested subprograms, are closures supported?

- A **closure** is a nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program.

Local Referencing Environments

- The issues related to variables that are defined within subprograms.

➢ Local Variables

- Variables that are defined inside subprograms are called **local variables**, because their scope is usually the body of the subprogram in which they are defined.
- Subprograms can define their own variables, thereby defining local referencing environments.
- local variables can be either **static or stack dynamic**.

- If local variables are stack dynamic, they are bound to storage when the subprogram begins execution and are unbound from storage when that execution terminates.
- There are several advantages of stack-dynamic local variables, the primary one being the flexibility they provide to the subprogram.
  - It is essential that recursive subprograms have stack-dynamic local variables.
  - Another advantage of stack-dynamic locals is that the storage for local variables in an active subprogram can be shared with the local variables in all inactive subprograms.

- The main disadvantages of stack-dynamic local variables are the following:
  - First, there is the cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call to the subprogram.
  - Second, accesses to stack-dynamic local variables must be indirect, whereas accesses to static variables can be direct.
  - Finally, when all local variables are stack dynamic, subprograms cannot be history sensitive; that is, they cannot retain data values of local variables between calls.

- The primary advantage of **static local variables** over stack-dynamic local variables is that they are slightly more efficient—they require no run-time overhead for allocation and deallocation.
- They allow subprograms to be history sensitive.
- The greatest disadvantage of static local variables is their inability to support recursion.
- Also, their storage cannot be shared with the local variables of other inactive subprograms.

- In most contemporary languages, local variables in a subprogram are by default stack dynamic.
- In C and C++ functions, local variables are stack dynamic unless specifically declared to be static.
- The methods of C++, Java, and C# have only stack-dynamic local variables.
- All local variables in Python methods are stack dynamic.

Nested Subprograms

- If a subprogram is needed only within another subprogram, why not place it there and hide it from the rest of the program?
- Because static scoping is usually used in languages that allow subprograms to be nested, this also provides a highly structured way to grant access to nonlocal variables in enclosing subprograms.
- The idea of nesting subprograms originated with Algol 60.
- The only languages that allowed nested subprograms were those directly descending from Algol 60, which were Algol 68, Pascal, and Ada.
- Recently, some new languages again allow it. Among these are JavaScript, Python, Ruby, and Lua.
- Also, most functional programming languages allow subprograms to be nested.
- Many other languages, including all of the direct descendants of C, do not allow subprogram nesting.

```
def outer_function():
    print("This is the outer function.")

    def inner_function():
        print("This is the inner function.")

    inner_function()

# Calling the outer function
outer_function()
```

Parameter-Passing Methods

- Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms.

➢ Semantics Models of Parameter Passing

- Formal parameters are characterized by one of three distinct semantics models:

  (1) They can receive data from the corresponding actual parameter;

  (2) they can transmit data to the actual parameter; or

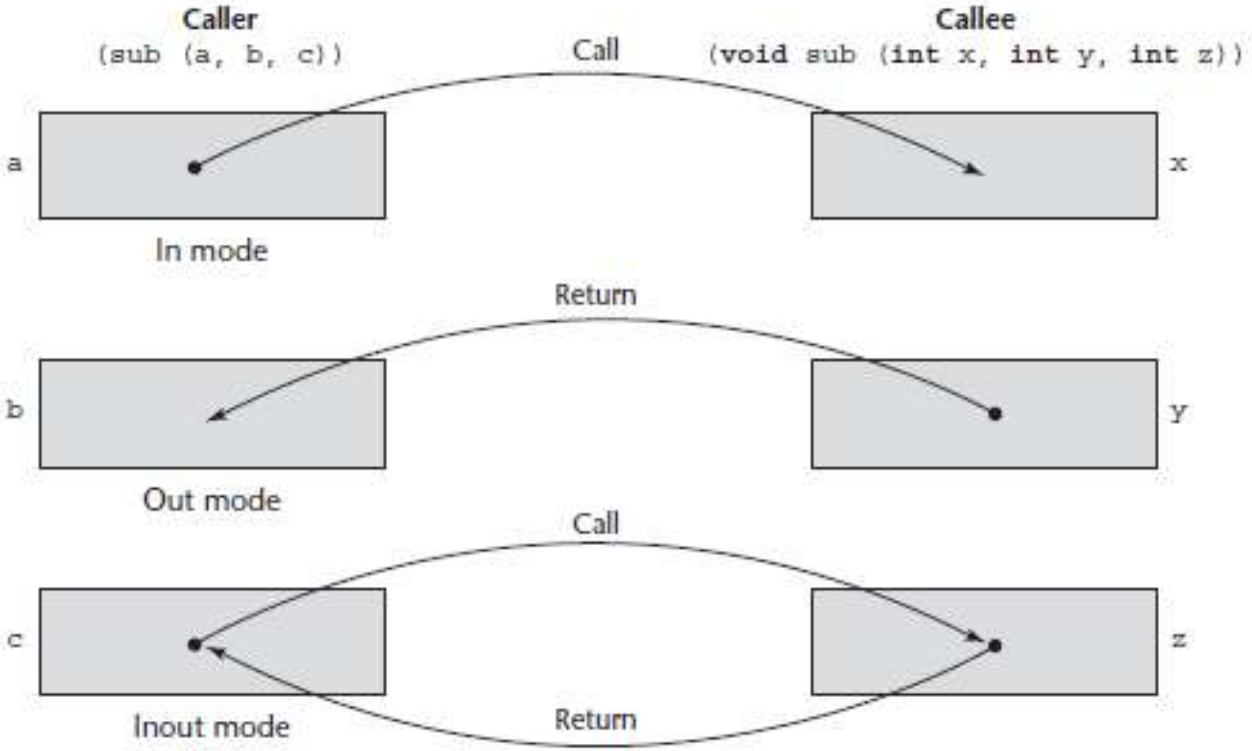
  (3) they can do both.

These models are called **in mode**, **out mode**, and **inout mode**, respectively.

- For **example**, consider a subprogram that takes two arrays of int values as parameters—list1 and list2.
- The subprogram must add list1 to list2 and return the result as a revised version of list2. Furthermore, the subprogram must create a new array from the two given arrays and return it.
  - For this subprogram, list1 should be in mode, because it is not to be changed by the subprogram.
  - list2 must be inout mode, because the subprogram needs the given value of the array and must return its new value.
  - The third array should be out mode, because there is no initial value for this array and its computed value must be returned to the caller.

Figure 9.1

The three semantics models of parameter passing when physical moves are used



Implementation Models of Parameter Passing

- A variety of models have been developed by language designers to guide the implementation of the three basic parameter transmission modes.

1. Pass-by-value

- When a parameter is **passed by value**, the value of the actual parameter is used to initialize the corresponding formal parameter.
- Local variable in the subprogram, thus implementing **in-mode** semantics.

- Either by physical move or access path
- Pass-by-value is normally implemented by copy the value or it could be implemented by transmitting an access path to the value of the actual parameter in the caller.

- Disadvantages of access path method:

  - Must write-protect(one that can only be read) in the called subprogram

  - Accesses cost more (indirect addressing)
- Disadvantages of physical move:

  - Requires more storage

  - Cost of the moves

2. Pass-by-result

- Pass-by-result is an implementation model for **out-mode** parameters.
- When a parameter is passed by result, no value is transmitted to the subprogram.
- Local variable's value is passed back to the caller
- Physical move is usually used
- The pass-by-result method has the advantages and disadvantages of passby-value, plus some additional disadvantages.

- **Disadvantages:**
- If values are returned by copy (as opposed to access paths), as they typically are, pass-by-result also requires the extra storage and the copy operations that are required by pass-by-value(time and space).

- One additional problem with the pass-by-result model is that there can be *an actual parameter collision*, such as the one created with the call

  sub(p1, p1)
- In sub, assuming the two formal parameters have different names, the two can obviously be assigned different values. Then, whichever of the two is copied to their corresponding actual parameter last becomes the value of p1 in the caller.
- Thus, the order in which the actual parameters are copied determines their value.

- For example, consider the following C# method, which specifies the pass-by-result method

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

  - If, at the end of the execution of Fixer, the formal parameter x is assigned to its corresponding actual parameter first, then the value of the actual parameter a in the caller will be 35.
  - If y is assigned first, then the value of the actual parameter a in the caller will be 17.
  - Value of a in the caller depends on order of assignments at the return

3. Pass-by-value-result

- Pass-by-value-result is an implementation model for **inout-mode** parameters in which actual values are copied.
- It is in effect a combination of pass-by-value and pass-by-result.
- The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable.
- Pass-by-value-result is sometimes called **pass-by-copy**, because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

- Pass-by-value-result shares with pass-by-value and pass-by-result the **disadvantages**
  - of requiring multiple storage for parameters and time for copying values.
  - It shares with pass-by-result the problems associated with the order in which actual parameters are assigned.

4. Pass-by-reference

- Pass-by-reference is a second implementation model for **inout-mode** parameters.
- The pass-by-reference method transmits an access path, usually just an address, to the called subprogram.
- The advantage of pass-by-reference is that the passing process itself is efficient, in terms of both time and space.
- **Disadvantages:**

✓ access to the formal parameters will be slower than pass-by-value parameters

✓ if only one-way communication to the called subprogram is required, inadvertent and erroneous changes may be made to the actual parameter.

✓ Another problem of pass-by-reference is that aliases can be created.

- There are several ways pass-by-reference parameters can create aliases.
- First, collisions can occur between actual parameters.

Consider a C++ function that has two parameters that are to be passed by reference, as in

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

If the call to fun happens to pass the same variable twice, as in

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

then first and second in fun will be aliases.

- Second, collisions between array elements can also cause aliases.

For example, suppose the function fun is called with two array elements that are specified with variable subscripts, as in

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

If these two parameters are passed by reference and i happens to be equal to j, then first and second are again aliases.

- Third, if two of the formal parameters of a subprogram are an element of an array and the whole array, and both are passed by reference, then a call such as

  fun1(list[i], list)

could result in aliasing in fun1, because fun1 can access all elements of list through the second parameter and access a single element through its first parameter

- Still another way to get aliasing with pass-by-reference parameters is through collisions between formal parameters and nonlocal variables(global) that are visible.

- For example, consider the following C code:

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

Inside sub, param and global are aliases.

- All these possible aliasing situations are eliminated if pass-by-value-result is used instead of pass-by-reference.

5. Pass-by-name

- Pass-by-name is an **inout-mode** parameter transmission method
- When parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram.
- Pass-by-name parameters are both complex to implement and inefficient.
- They also add significant complexity to the program, thereby lowering its readability and reliability.

Implementing Parameter-Passing Methods

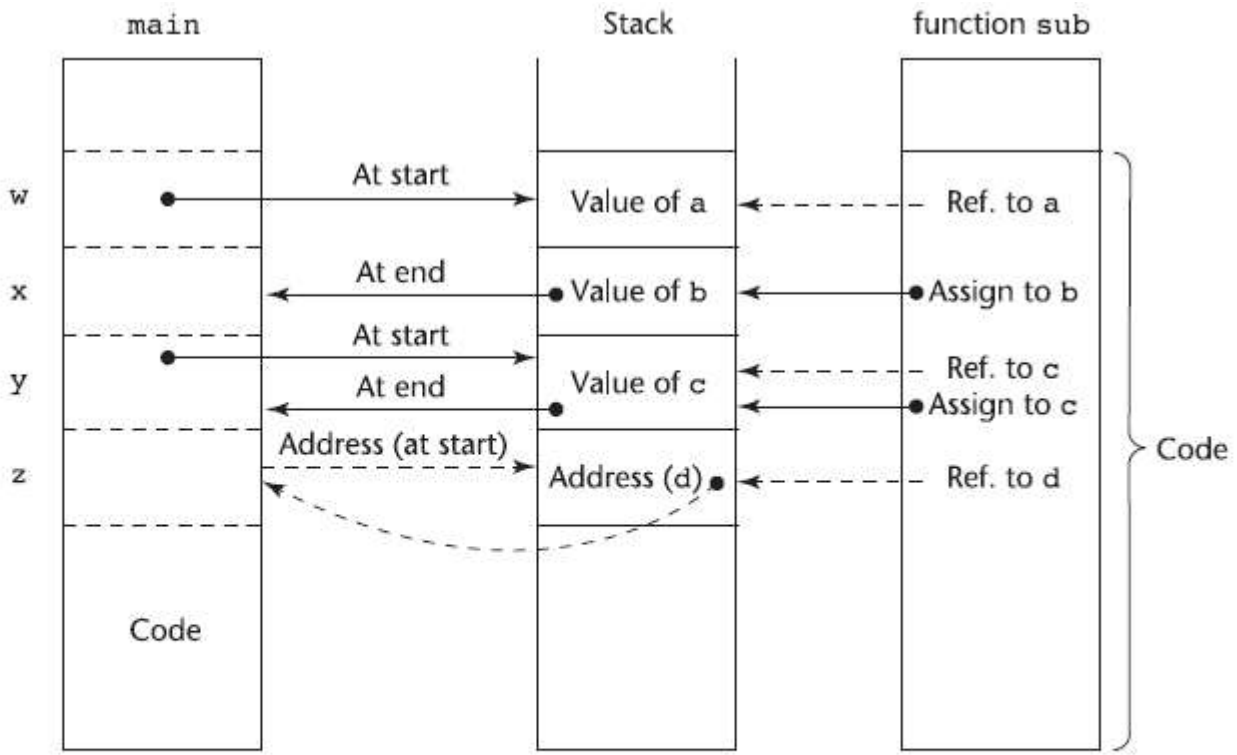
- In most contemporary languages, parameter communication takes place through the **run-time stack**.
- The run-time stack is initialized and maintained by the run-time system, which manages the execution of programs.
- Pass-by-value parameters have their values copied into stack locations.
- The stack locations then serve as storage for the corresponding formal parameters.
- Pass-by-result parameters are implemented as the opposite of pass-by-value.
- The values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram.

- Pass-by-value-result parameters can be implemented directly from their semantics as a combination of pass-by-value and pass-by-result.
- The stack location for such a parameter is initialized by the call and is then used like a local variable in the called subprogram.
- Pass-by-reference parameters are perhaps the simplest to implement.
- Regardless of the type of the actual parameter, only its address must be placed in the stack.
- In the case of literals, the address of the literal is put in the stack.
- In the case of an expression, the compiler must build code to evaluate the expression just before the transfer of control to the called subprogram.
- The address of the memory cell in which the code places the result of its evaluation is then put in the stack.

- The implementation of passby-value, -result, -value-result, and -reference, where the run-time stack is used, is shown in Figure 9.2. Subprogram sub is called from main with the call sub(w, x, y, z), where w is passed by value, x is passed by result, y is passed by value-result, and z is passed by reference.

Figure 9.2

One possible stack implementation of the common 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)

➢ Type Checking Parameters

- Software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters.
- Without such type checking, small typographical errors can lead to program errors that may be difficult to diagnose because they are not detected by the compiler or the run-time system.
- For example, in the function call

  result = sub1(1)

the actual parameter is an integer constant.

If the formal parameter of sub1 is a floating-point type, no error will be detected without parameter type checking.

Multidimensional Arrays as Parameters

- In some languages, such as C and C++, when a multidimensional array is passed as a parameter to a subprogram, the compiler must be able to build the mapping function for that array while seeing only the text of the subprogram (not the calling subprogram). This is true because the subprograms can be compiled separately from the programs that call them.
- Consider the problem of passing a matrix to a function in C. Multidimensional arrays in C are really arrays of arrays, and they are stored in row major order.
- Following is a storage-mapping function for row major order for matrices when the lower bound of all indices is 0 and the element size is 1:

  address(mat[i, j]) = address(mat[0,0]) + i *
  number_of_columns + j

- Notice that this mapping function needs the number of columns but not the number of rows.

- Therefore, in C and C++, when a matrix is passed as a parameter, the formal parameter must include the number of columns in the second pair of brackets. This is illustrated in the following skeletal C program:

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

- The problem with this method of passing matrixes as parameters is that it does not allow a programmer to write a function that can accept matrixes with different numbers of columns; a new function must be written for every matrix with a different number of columns.
- This, in effect, disallows writing flexible functions that may be effectively reusable if the functions deal with multidimensional arrays.

Examples of Parameter Passing

- Consider the following C function:

```
void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

Suppose this function is called with

```
swap1(c, d);
```

- Recall that C uses pass-by-value. The actions of swap1 can be described by the following pseudocode:

```
a = c          – Move first parameter value in
b = d          – Move second parameter value in
temp = a
a = b
b = temp
```

- Although a ends up with d's value and b ends up with c's value, the values of c and d are unchanged because nothing is transmitted back to the caller.
- We can modify the C swap function to deal with pointer parameters to achieve the effect of pass-by-reference:

```
void swap2(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

- swap2 can be called with

  swap2(&c, &d);
- The actions of swap2 can be described with

```
a = &c          – Move first parameter address in
b = &d          – Move second parameter address in
temp = *a
*a = *b
*b = temp
```

- In this case, the swap operation is successful: The values of c and d are in fact interchanged.

Subprograms as Parameters

- subprogram names can be sent as parameters to other subprograms.
- For example, a subprogram that does numerical integration estimates the area under the graph of a function by sampling the function at a number of different points.
- If only the transmission of the subprogram code was necessary, it could be done by passing a single pointer.
- However, two complications arise.
  - First, there is the matter of type checking the parameters of the activations of the subprogram that was passed as a parameter.
- In C and C++, functions cannot be passed as parameters, but pointers to functions can.
- The type of a pointer to a function includes the function's protocol. Because the protocol includes all parameter types, such parameters can be completely type checked.

- The second complication with parameters that are subprograms appears only with languages that allow nested subprograms.

- The issue is what referencing environment for executing the passed subprogram should be used.
- There are three choices:

• The environment of the call statement that enacts the passed subprogram (**shallow binding)**

(It is that of the subprogram that enacts it.)

• The environment of the definition of the passed subprogram (**deep binding)**

(It is that of the subprogram that declared it.)

• The environment of the call statement that passed the subprogram as an actual parameter (**ad hoc binding)**

(It is that of the subprogram that passed it.)

- The following example program, written with the syntax of JavaScript, illustrates these choices:

```
function sub1() {
  var x;
  function sub2() {
    alert(x);  // Creates a dialog box with the value of x
    };
  function sub3() {
    var x;
    x = 3;
    sub4(sub2);
    };
  function sub4(subx) {
    var x;
    x = 4;
    subx();
    };
  x = 1;
  sub3();
  };
```

Consider the execution of sub2 when it is called in sub4.

For shallow binding, the referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4, and the output of the program is 4.

For deep binding, the referencing environment of sub2's execution is that of sub1, so the reference to x in sub2 is bound to the local x in sub1, and the output is 1.

For ad hoc binding, the binding is to the local x in sub3, and the output is 3.

- For static-scoped languages, deep binding is most natural
- For dynamic-scoped languages, shallow binding is most natural
- Shallow binding is not appropriate for static-scoped languages with nested subprograms.

Overloaded Subprograms

- An overloaded operator is one that has multiple meanings.
- The meaning of a particular instance of an overloaded operator is determined by the types of its operands.
- For example, if the * operator has two floating-point operands in a Java program, it specifies floating-point multiplication. But if the same operator has two integer operands, it specifies integer multiplication.

- An **overloaded subprogram** is a subprogram that has the same name as another subprogram in the same referencing environment.
- Every version of an overloaded subprogram must have a unique protocol; that is, it must be different from the others in the number, order, or types of its parameters, and possibly in its return type if it is a function.
- The meaning of a call to an overloaded subprogram is determined by the actual parameter list (and/or possibly the type of the returned value, in the case of a function).
- C++, Java, Ada, and C# include predefined overloaded subprograms. For example, many classes in C++, Java, and C# have overloaded constructors.

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.
- Explaining a closure is not so simple.
- If a static-scoped programming language does not allow nested subprograms, closures are not useful, so such languages do not support them.
- There is an associated problem:
  - The subprogram could be called after one or more of its nesting subprograms has terminated, which normally means that the variables defined in such nesting subprograms have been deallocated—they no longer exist.

- For the subprogram to be callable from anywhere in the program, its referencing environment must be available wherever it might be called.
- Therefore, the variables defined in nesting subprograms may need lifetimes that are of the entire program, rather than just the time during which the subprogram in which they were defined is active.

Coroutines

- A **coroutine** is a special kind of subprogram. Rather than the master-slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are more equitable.
- The coroutine control mechanism is often called the symmetric unit control model.
- Coroutines can have multiple entry points, which are controlled by the coroutines themselves.
- coroutines must be history sensitive and thus have static local variables.
- Secondary executions of a coroutine often begin at points other than its beginning. Because of this, the invocation of a coroutine is called a **resume** rather than a call.

- For example, consider the following skeletal coroutine:

```
sub co1(){
   ...
   resume co2();
   ...
   resume co3();
   ...
}
```

The first time co1 is resumed, its execution begins at the first statement and executes down to and including the resume of co2, which transfers control to co2.

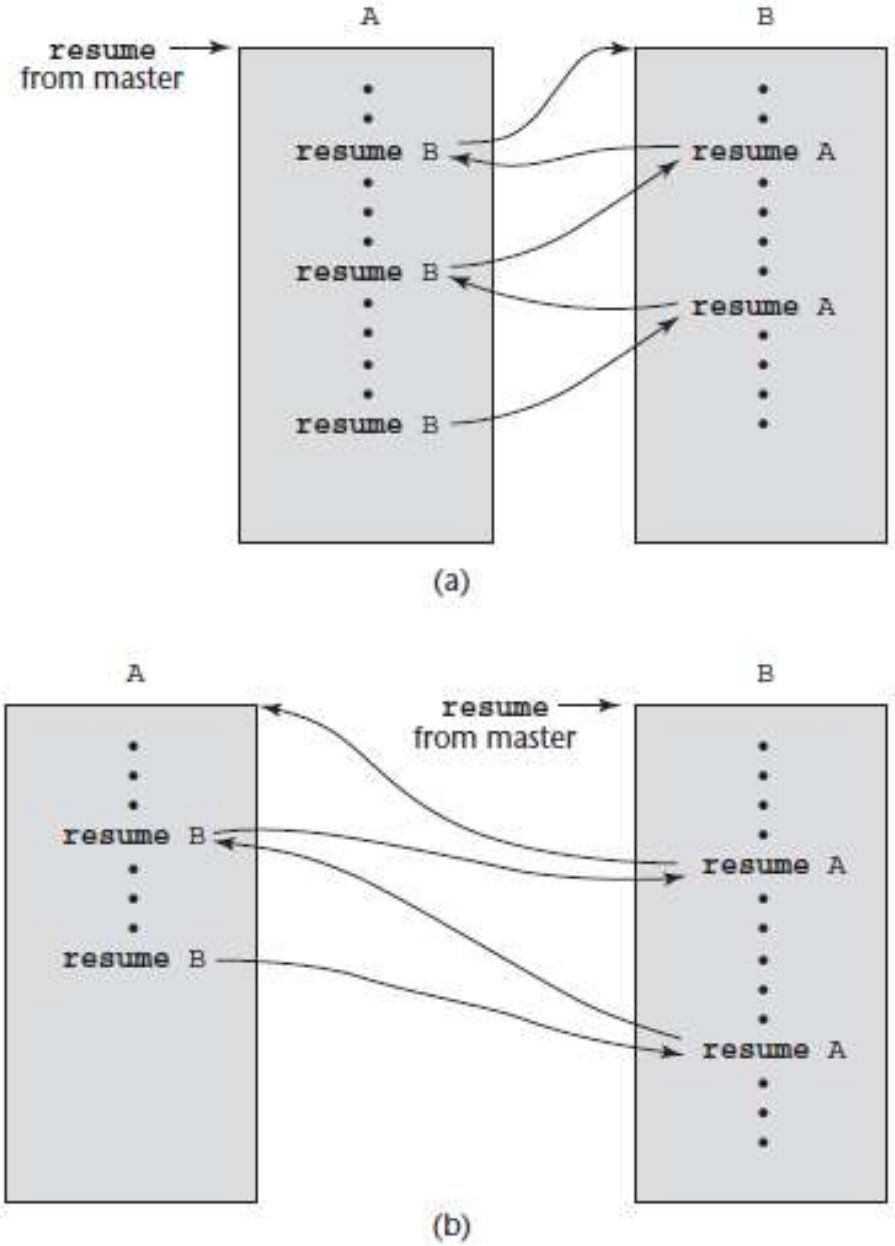
The next time co1 is resumed, its execution begins at the first statement after its call to co2.

When co1 is resumed the third time, its execution begins at the first statement after the resume of co3.

- Only one coroutine is actually in execution at a given time.

Figure 9.3

Two possible execution control sequences for two coroutines without loops



Suppose program units A and B are coroutines. Figure 9.3 shows two ways an execution sequence involving A and B might proceed.

In Figure 9.3a, the execution of coroutine A is started by the master unit. After some execution, A starts B. When coroutine B in Figure 9.3a first causes control to return to coroutine A, the semantics is that A continues from where it ended its last execution. In particular, its local variables have the values left them by the previous activation.

Figure 9.3b shows an alternative execution sequence of coroutines A and B. In this case, B is started by the master unit.

- Rather than have the patterns shown in Figure 9.3, a coroutine often has a loop containing a resume.
- Figure 9.4 shows the execution sequence of this scenario.
- In this case, A is started by the master unit. Inside its main loop, A resumes B, which in turn resumes A in its main loop.

Figure 9.4

Coroutine execution sequence with loops