Chapter 9

Subprograms

Chapter 9 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

Chapter 9

Subprograms

Introduction

- Subprograms are the fundamental building blocks of programs and are therefore among the most import concepts in programming language design.
- The **reuse** results in several different kinds of savings, including memory space and coding time.

Fundamentals of Subprograms

General Subprogram Characteristics

a. A subprogram has a single entry point.
b. The caller is suspended during execution of the called subprogram, which implies that there is **only one** subprogram in execution **at any given time**.
c. Control always returns to the caller when the called subprogram's execution terminates

Basic Definitions

- A subprogram definition is a description of the actions of the subprogram abstraction.
- A subprogram call is an explicit request that the called subprogram be executed.
- A subprogram is said to be **active** if, after having been called, it has begun execution but has not yet completed that execution.
- The **two** fundamental types of the subprograms are:
  - Procedures
  - Functions
- A **subprogram header** is the first line of the definition, serves several definitions:
  - It specifies that the following syntactic unit is a subprogram definition of some particular kind.
  - The header provides a name for the subprogram.
  - May optionally specify a list of parameters.
- Consider the following header examples:
- Fortran

```
Subroutine Adder(parameters)
```

- Ada

```
procedure Adder(parameters)
```

- C

```
void Adder(parameters)
```

- No special word appears in the header of a C subprogram to specify its kind.
- The parameter profile (sometimes called the signature) of a subprogram is the number, order, and types of its formal parameters.
- The protocol of a subprogram is its parameter profile plus, if it is a function, its return type.
- 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.
- Function declarations are common in C and C++ programs, where they are called **prototypes**.
- **Java and C# do not need declarations of their methods**, because there is no requirement that methods be defined before they are called in those languages.

Parameters

- Subprograms typically describe computations. There are two ways that a non-local method program can gain access to the data that it is to process:
  1. Through **direct access** to non-local variables.
     - The only way the computation can proceed on different data is to assign new values to those non-local variables between calls to the subprograms.
     - Extensive access to non-locals can reduce reliability.
  2. Through **parameter passing** "more flexible".
     - Data passed through parameters are accessed through names that are **local** to the subprogram.
     - A subprogram with parameter access to the data it is to process is a parameterized computation.
     - It can perform its computation on whatever data it receives through its parameters.
- A **formal parameter** is a dummy variable listed in the **subprogram header** and used in the subprogram.
- Subprograms call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram.
- An **actual parameter** represents **a value or address** used in the subprogram call statement.
- Actual/Formal Parameter Correspondence:
  1. **Positional**: The first actual parameter is bound to the first formal parameter and so forth. "Practical if list is short."
  2. **Keyword**: the name of the formal parameter is to be bound with the actual parameter. "Can appear in any order in the actual parameter list."

     ```
     SORT(LIST => A, LENGTH => N);
     ```

    - Advantage: order is **irrelevant**
    - Disadvantage: user **must** know the formal parameter's **names**.
- Default Values:

```
procedure SORT(LIST : LIST_TYPE;
               LENGTH : INTEGER := 100);
...

SORT(LIST => A);
```

- In C++, which has no keyword parameters, the rules for default parameters are necessarily different.
- The default parameters must appear last, for parameters are positionally associated.
- Once a default parameter is omitted in a call, all remaining formal parameters must have default values.

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

- An example call to the **C++** `compute_pay` function is:

```
pay = compute_pay(20000.0, 0.15);
```

Procedures and Functions

- There are **two** distinct categories of subprograms, procedures and functions.
- **Procedures**: provide user-defined parameterized computation statements.
- The computations are enacted by single call statements.
- Procedures can produce results in the calling program unit by two methods:
    - If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them.
    - If the subprogram has formal parameters that allow the transfer of data to the caller, those parameters **can be changed**.
- **Functions:** provide user-defined operators which are semantically modeled on mathematical functions.
    - If a function is a faithful model, it produces no side effects.
    - It **modifies neither its parameters nor any variables** defined outside the function.
- The **methods** of Java, C++, and C# are syntactically similar to the functions of C.

Design Issues for Subprograms

1. What parameter passing methods are provided?
2. Are parameter types checked?
3. Are local variables static or dynamic?
4. Can subprogram definitions appear in other subprogram definitions?
5. What is the referencing environment of a passed subprogram?
6. Can subprograms be overloaded?
7. Are subprograms allowed to be generic?

Local Referencing Environments

- Vars that are defined inside subprograms are called **local vars**.
- Local vars can be either static or stack dynamic "bound to storage when the program begins execution and are unbound when execution terminates."
- Advantages of using **stack dynamic**:
  a. Support for recursion.
  b. Storage for locals is shared among some subprograms.
- Disadvantages:
  a. Allocation/deallocation time.
  b. Indirect addressing "only determined during execution."
  c. Subprograms cannot be history sensitive "can't retain data values of local vars between calls."
- Advantages of using **static vars**:
  a. Static local vars can be accessed faster because there is no indirection.
  b. No run-time overhead for allocation and deallocation.
  c. Allow subprograms to be history sensitive.
- Disadvantages:
  a. Inability to support recursion.
  b. Their storage can't be shared with the local vars of other inactive subprograms.
- In **C** functions, locals are stack-dynamic unless specifically declared to be static.
- Ex:

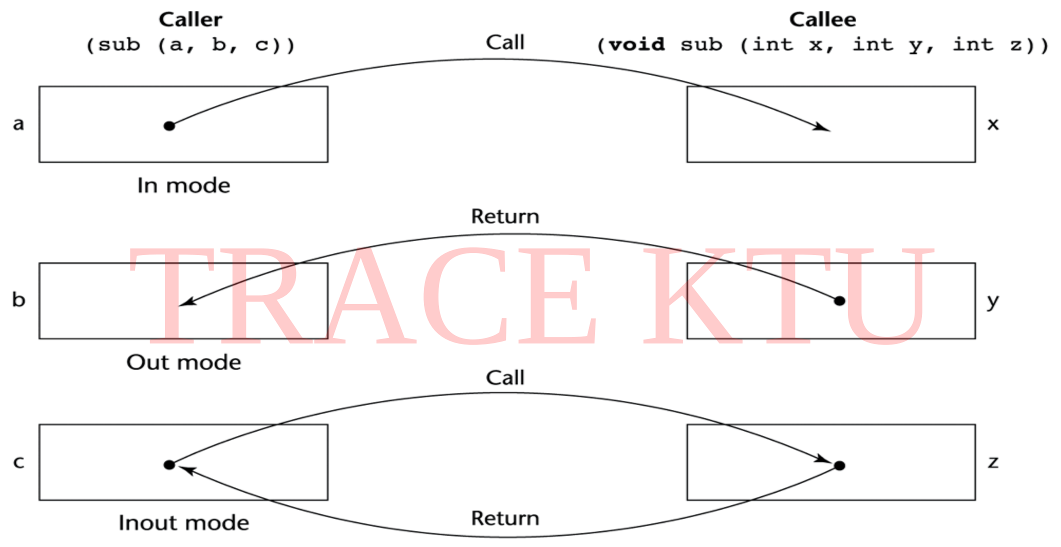
```
int adder(int list[ ], int listlen) {
   static int sum = 0;   //sum is static variable
   int count;            //count is stack-dynamic
   for (count = 0; count < listlen; count++)
      sum += list[count];
   return sum;
}
```

- Ada subprograms and the **methods** of C++, Java, and C# have **only stack-dynamic** local variables.

Parameter Passing Methods

Semantic Models of Parameter Passing

- Formal parameters are characterized by one of three distinct semantic models:
  - **in mode**: They can receive data from corresponding actual parameters.
  - **out mode**: They can transmit data to the actual parameter.
  - **inout mode**: They can do both.
- There are two conceptual models of how data transfers take places in parameter transmission:
  - Either an **actual value** is copied (to the caller, to the callee, or both ways), or
  - An **access path** is transmitted.
- Most commonly, the access path is a simple **pointer** or **reference**.
- Figure below illustrates the three semantics of parameter passing when values are copied.



Implementation Models of Parameter Passing

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, which then acts as a local var in the subprogram, thus implementing **in-mode** semantics.
- Disadvantages:
  Additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram. The actual parameter must be **copied** to the storage area for the corresponding formal parameter. "If the parameter is large such as an **array**, it would be costly.

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.
- The corresponding formal parameter acts as a **local var**, but just before control is transferred back to the caller, its value is transmitted back to the **caller's actual parameter**, which must be a var.
- One 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 that 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.

3. Pass-by-Value-Result

- It is an implementation model for **inout-mode** parameters in which actual values are copied.
- It is 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 var.
  At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.
- It 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.

4. Pass-by-reference

- Pass-by-reference is a second implementation model for inout-mode parameters.
- Rather than **copying** data values back and forth. This method transmits an **access path**, sometimes just **an address**, to the called subprogram. This provides the access path to the cell storing the actual parameter.
- The actual parameter is **shared** with the called subprogram.
- Advantages:
  The passing process is **efficient** in terms of time and space. Duplicate space is not required, nor is any copying.
- Disadvantages:
  Access to the formal parameters will be **slower** than pass-by-value, because of additional level of **indirect addressing** that is required.
- Inadvertent and erroneous **changes** may be made to the **actual parameter**.
- **Aliases** can be created as in C++.

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

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

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

Then first and second in fun will be aliases.

5. **Pass-by-Name**

- The method is an **inout-mode** parameter transmission that doesn't correspond to a single implementation model.
- 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.
- A formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.
- Because pass-by-name is **not used** in any widely used language, it is not discussed further here

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

    ```
    void fun(const int &p1, int p2, int &p3)  { … }
    ```

- Java
  - All parameters are passed are **passed by value**
  - Object parameters are **passed by reference**
    - Although an object reference passed as a parameter **cannot** itself be changed in the called subprogram, the referenced object can be changed if a method is available to cause the change.
- 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#
- Perl: all actual parameters are implicitly placed in a predefined array named @_

Type-Checking Parameters

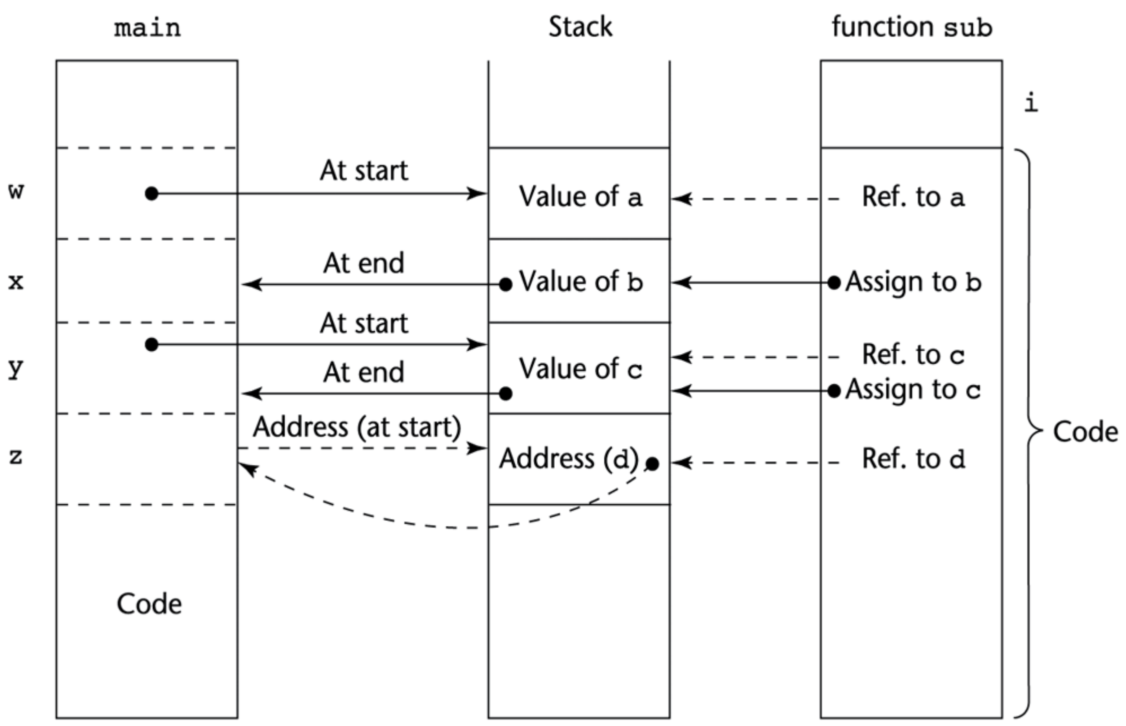
- It is now widely accepted that software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters.
- Ex:

```
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.
- Early languages, such as Fortran 77 and the original version of C, **did not** require parameter type checking.
- Pascal, FORTRAN 90, **Java**, and Ada: it is always **required**
- Perl, PHP, and JavaScript **do not** have type checking.

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 is a system program that manages the execution of programs.
- The run-time stack is used extensively for subprogram control linkage and parameter passing.
- **Pass-by-value** parameters have their values copied into stack locations.
- The stack location then serves 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 pf **pass-by-value** and **pass-by-result**.
- The stack location for the parameters is initialized by the call and it then used like a local var in the called subprogram.
- **Pass-by-reference** parameters are the simplest to implement.
- Only its address must be placed in the stack.
- Access to the formal parameters in the called subprogram is by indirect addressing from the stack location of the address.
- Figure below illustrates the previous parameters' passing methods.



- The 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**.
- A subtle but fatal error can occur with pass-by-reference and pass-by-value-result parameters if care is not take in their implementation.
- Suppose a program contains two references to the constant **10**, the first as an actual parameter in a call to a subprogram.
- Further suppose that the subprogram mistakenly changes the formal parameter that corresponds to the 10 to the value **5**.
- The compiler may have built a single location for the value 10 during compilation, as compilers often do, and use that location for all references to the constant 10 in the program.
- But after the return from the subprogram, all subsequent occurrences of **10** will actually be references to the value **5**.
- If this is allowed to happen, it creates a programming problem that is very difficult to diagnose.
- This happened in many implementations of Fortran IV.

Parameters that are Subprogram Names

- In languages that allow nested subprograms, such as JavaScript, there is another issue related to subprogram names that are passed as parameters.
- The question is what referencing environment for executing the passed subprogram should be used.
- The three choices are:
  1. It is the environment of the call statement that enacts the passed subprogram "**Shallow binding**."
  2. It is the environment of the definition of the passed subprogram "Deep binding."
  3. It is the environment of the call statement that passed the subprogram as an actual parameter "**Ad hoc binding**; has never been used"
- Ex: "written in the syntax of Java"

```
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.
- **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**.
- **Deep Binding**: the referencing environment of sub2's execution is that of sub1, so the reference so the reference to x in sub2 is bound to the local x in sub1 and the output is **1**.
- **Ad hoc**: the binding is to the local x in sub3, and the output is **3**.
- Shallow binding is not appropriate for **static-scoped languages** with nested subprograms.

Overloaded Subprograms

- An overloaded operator is one that has multiple meanings. The types of its operands determine the meaning of a particular instance of an overloaded operator.
- 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, or in its return if it is a function**.
- The meaning of a call to an overloaded subprogram is determined by the actual parameter list.
- Users are also allowed to write multiple versions of subprograms with the same name in Ada, Java, C++, and C#.
- Overloaded subprograms that have default parameters can lead to **ambiguous** subprogram calls.

```
void fun(float b = 0.0);
void fun( );
…

fun( );  // The call is ambiguous and will cause a compilation error.
```

Generic Subprograms

- A programmer should not need to write four different sort subprograms to sort four arrays that differ only in element type.
- A generic or polymorphic subprogram takes parameters of **different types** on different activations.
- Overloaded subprograms provide a particular kind of polymorphism called ad hoc polymorphism.
- Parametric polymorphism is provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameters of the subprogram.

Generic Functions in C++

- Generic functions in C++ have the descriptive name of template functions.
- The following is the C++ version of the generic sort subprogram.

```
template <class Type>
void generic_sort (Type list [ ], int len) {
   int top, bottom;
   Type temp;
   for (top = 0, top < len –2; top ++)
        for (bottom = top + 1; bottom <  len – 1; bottom++)
     if (list [top] > list [bottom]) {
         temp = list [top];
         list[top] = list[bottom];
     } // end for bottom
} // end for generic
```

- The instantiation of this template function is:

```
float flt_list [100];
…
generic_sort (flt_list, 100);
```

Design Issues for Functions

- Are side effects allowed?
- What types of values can be returned?

Functional Side Effects

- Because of the problems of side effects of functions that are called in expressions, parameters to functions should always be in-mode parameters.
- Ada functions can have only in-mode formal parameters.
- This effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals.
- In most languages, however, functions can have either **pass-by-value or pass-by-reference parameters**, thus allowing functions that cause side effects and aliasing.

Types of Returned Values

- C allows **any type** to be returned by its functions except arrays and functions.
- C++ is like C but also allows user-defined types, or classes, to be returned from its functions.
- JavaScript functions can be passed as parameters and returned from functions.

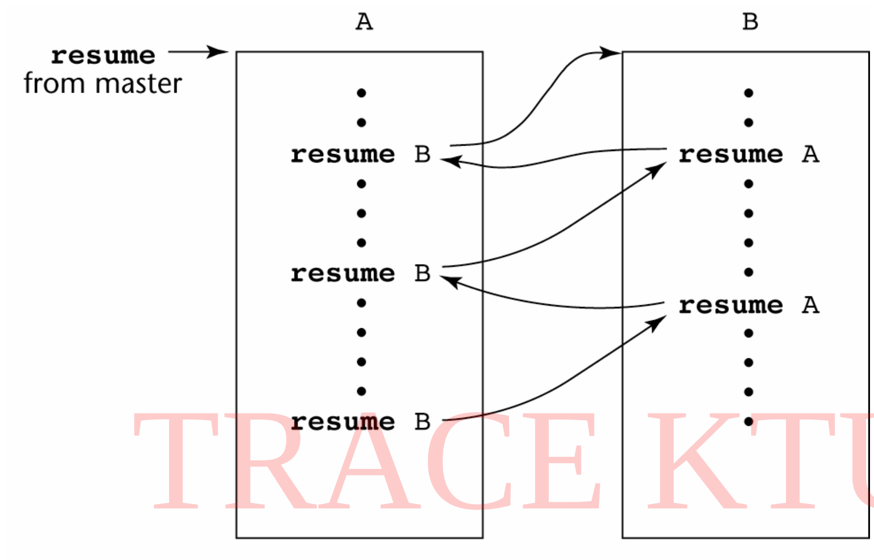
User-Defined Overloaded Operators

- Nearly all programming languages have overloaded operators.
- Users can further overload operators in **C++ and Ada** (**Not** carried over into **Java**)
- How much operator overloading is good, or can you have too much?

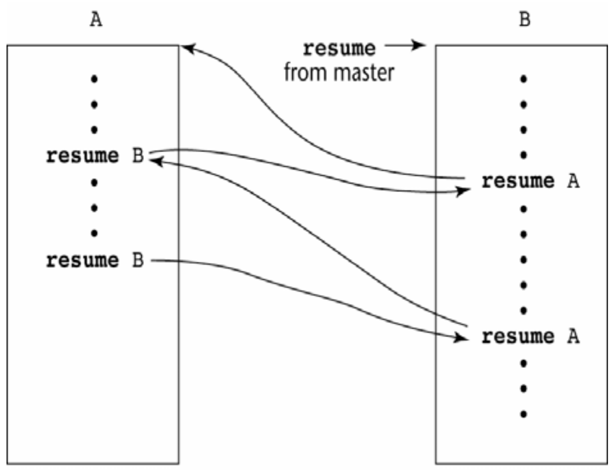
```
Function "*"(A,B: in Vec_Type): return Integer is
    Sum: Integer := 0;
    begin
    for Index in A'range loop
            Sum := Sum + A(Index) * B(Index)
    end loop
    return sum;
end "*";
…
c = a * b; -- a, b, and c are of type Vec_Type
```

Coroutines

- A coroutine is a subprogram that has multiple entries and controls them itself.
- It is also called symmetric control.
- It also has the means to maintain their status between activation.
- This means that coroutines must be history sensitive and thus have static local vars.
- Secondary executions of a coroutine often begin at points other than its beginning.
- 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.



(a)



(b)