

Chapter 9

Subprograms

9.1 Introduction	366
9.2 Fundamentals of Subprograms	366
9.3 Design Issues for Subprograms	374
9.4 Local Referencing Environments	375
9.5 Parameter-Passing Methods	376
9.6 Parameters That Are Subprograms	392
9.7 Calling Subprograms Indirectly	394
9.8 Design Issues for Functions	396
9.9 Overloaded Subprograms	397
9.10 Generic Subprograms	398
9.11 User-Defined Overloaded Operators	404
9.12 Closures	405
9.13 Coroutines	407
Summary • Review Questions • Problem Set • Programming Exercises	410

Chapter 9

Subprograms

9.1 Introduction 366

- Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design
- This **reuse** results in savings, including memory space and coding time

9.2 Fundamentals of Subprograms 366

9.2.1 General Subprogram Characteristics

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

9.2.2 Basic Definitions

- A subprogram definition describes the interface to and 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 purposes:
 - It specifies that the following syntactic unit is a subprogram definition of some particular kind
 - If the subprogram is not anonymous, the header provides a name for the subprogram
 - It may optionally specify a list of parameters.
- Consider the following header examples:
 - In Python, the header of a subprogram named `adder`

```
def adder(parameters):
```
 - Ruby subprogram headers also begin with `def`
 - The header of a JavaScript subprogram begins with `function`

- Ada

```
procedure adder(parameters)
```

- In C, the header of a function named `adder` might be as follows:

```
void adder(parameters)
```

- One characteristic of **Python** functions that sets them apart from the functions of other common programming languages is that function `def` statements are **executable**
 - Consider the following skeletal example:

```
if . . .
    def fun(. . . ):
        . . .
else
    def fun(. . . ):
        . . .
```

- When a `def` statement is executed, it assigns the given name to the given function body
- If the then clause of this selection construct is executed, that version of the function `fun` can be called, but not the version in the else clause. Likewise, if the else clause is chosen, its version of the function can be called but the one in the then clause cannot
- 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
 - 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

9.2.3 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:
 - Through **direct access** to non-local variables
 - Declared elsewhere but **visible** in the subprogram
 - 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:
 - **Positional**: The first actual parameter is bound to the first formal parameter and so forth
 - **Keyword**: The name of the formal parameter is to be bound with the actual parameter.
 - They can appear in any order in the actual parameter list. Python functions can be called using this technique, as in


```
sumer(length = my_length, list = my_array, sum = my_sum)
```
 - Where the definition of `sumer` has the formal parameters `length`, `list`, and `sum`
 - Advantage: parameter order is **irrelevant**
 - Disadvantage: user of the subprogram must know the **names** of formal parameters
- In Python, Ruby, C++, and PHP, formal parameters can have **default** values (if no actual parameter is passed)
 - In C++, which has no keyword parameters, the rules for default parameters are necessarily different
 - The default parameters must appear last; 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);
```
- In most languages that do **not** have default values for formal parameters, the number of actual parameters in a call must match the number of formal parameters in the subprogram definition header

9.2.4 Procedures and Functions

- There are **two** distinct categories of subprograms, procedures and functions
- Subprograms are collections of statements that define parameterized computations. Functions return values and procedures do **not**
- **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**: Functions structurally resemble procedures but 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 **returned** value is its only effect
- The functions in **most** programming languages have side effects
- The **methods** of Java are syntactically similar to the **functions** of C

9.3 Design Issues for Subprograms 374

- 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?

9.4 Local Referencing Environments 375

- Variables that are defined inside subprograms are called **local** variables.
- Local variables can be either static or stack dynamic “bound to storage when the program begins execution and are unbound when execution terminates”
- Local variables can be **stack dynamic**
 - Advantages
 - Support for recursion
 - Storage for locals is shared among some subprograms
 - Disadvantages:
 - Allocation/deallocation time
 - Indirect addressing “only determined during execution”
 - Subprograms cannot be history sensitive “can’t retain data values between calls”
- Local variables can be **static**
 - Advantages
 - Static local variables can be accessed faster because there is no indirection
 - No run-time overhead for allocation and deallocation
 - Allow subprograms to be history sensitive
 - Disadvantages
 - Inability to support recursion
 - Their storage can’t be shared with the local variables 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 variable  
    for (count = 0; count < listlen; count++)  
        sum += list[count];  
    return sum;  
}
```

- The methods of C++, Java, and C# have **only** stack-dynamic local variables
- Nested Subprograms
 - All of the direct descendants of C, do **not** allow subprogram nesting.
 - Recently, some new languages again allow it. Among these are JavaScript, Python, and Ruby.
 - Most functional programming languages allow subprograms to be nested.

9.5 Parameter-Passing Methods 376

9.5.1 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

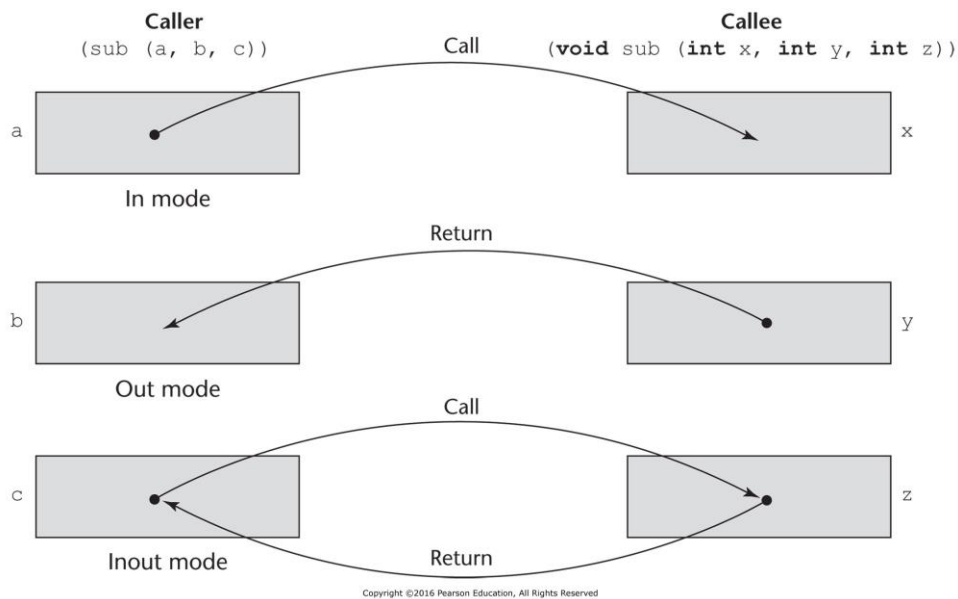


Figure 9.1 The three semantics models of parameter passing when physical moves are used

9.5.2 Implementation Models of Parameter Passing

- Five implementation models of parameter passing:
 - Pass-by-value
 - Pass-by-result
 - Pass-by-value-result
 - Pass-by-reference
 - Pass-by-name

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. The storage and the copy operations can be costly if the parameter is large, such as an **array** with many elements

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 variable, 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 variable
- 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 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 variable
- 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**, usually 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 variable 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 does not 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** part of any widely used language, it is not discussed further here

9.5.3 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 of 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 variable 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
- 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**

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

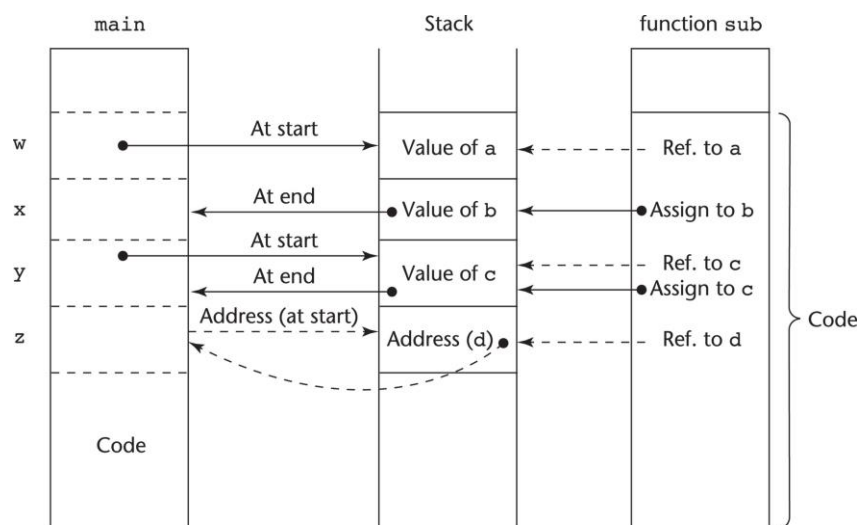


Figure 9.2 One possible stack implementation of the common parameter-passing methods

9.5.4 Parameter-Passing Methods of Some Common 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. Reference parameters are **implicitly dereferenced** in the function or method, and their semantics is pass-by-reference
 - C++ also allows reference parameters to be defined to be **constants**. For example, we could have

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

- p1 is pass-by-reference: p1 **cannot** be changed in the function fun
 - p2 is pass-by-value
 - p3 is pass-by-reference
 - Neither p1 nor p3 need be explicitly dereference in fun
- Java
 - All parameters are passed are **passed by value**
 - However, because objects can be accessed only through reference variables, object parameters are in effect **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**, **inout**; in is the default mode
 - Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; inout parameters can be referenced and assigned
- C#
 - Default method: pass-by-value
 - Pass-by-reference can be specified by preceding both a formal parameter and its actual parameter with **ref**

```
void sumer(ref int oldSum, int newOne) { . . . }  
. . .  
sumer(ref sum, newValue);
```

- The first parameter to `sumer` is passed-by-reference; the second is passed-by-value
- PHP: very similar to C#
- 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 process of changing the value of a variable with an assignment statement, as in

```
x = x + 1
```

- does **not** change the object referenced by `x`. Rather, it takes the object referenced by `x`, increments it by 1, thereby creating a **new** object (with the value `x + 1`), and then `x` to reference the new object.

9.5.5 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, **Java**, and Ada: it is always **required**
- Relatively new languages Perl, JavaScript, and PHP do **not** require type checking

9.6 Parameters That Are Subprograms 392

- The issue is what referencing environment for executing the passed subprogram should be used
- The three choices are:
 1. **Shallow binding:** The environment of the call statement that enacts the passed subprogram
 - Most natural for **dynamic-scoped** languages
 2. **Deep binding:** The environment of the definition of the passed subprogram
 - Most natural for **static-scoped** languages
 3. **Ad hoc binding:** The environment of the call statement that passed the subprogram as an actual parameter
 - It has **never** been used because, one might surmise, the environment in which the procedure appears as a parameter has no natural connection to the passed subprograms
- Ex: JavaScript

```
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 binding: the binding is to the local **x** in **sub3**, and the output is **3**.

9.7 Calling Subprograms Indirectly 394

- There are situations in which subprograms must be called indirectly
- These most often occur when the specific subprogram to be called is not known until **run time**
- The call to the subprogram is made through a pointer or reference to the subprogram, which has been set during execution before the call is made
- C and C++ allow a program to define a **pointer to function**, through which the function can be called
- For example, the following declaration defines a pointer (`pfun`) that can point to any function that takes a `float` and an `int` as parameters and returns a `float`

```
float (*pfun) (float, int);
```

- **Both** following are legal ways of giving an initial value or assigning a value to pointer to a function

```
int myfun2(int, int); // A function declaration
int (*pfun2)(int, int) = myfun2; // Create a pointer and initialize
                                // it to point to myfun2
```

```
int myfun2(int, int); // A function declaration
int (*pfun2)(int, int); // Create a pointer
pfun2 = myfun2;         // Assigning a function's address to a pointer
```

- The function `myfun2` can now be called with either of the following statements:

```
(*pfun2)(first, second);
```

```
pfun2(first, second);
```

- The first of these explicitly dereferences pointer `pfun2`, which is legal, but **unnecessary**

- In C#, the power and flexibility of method pointers is increased by making them **objects**
- These are called **delegates**, because instead of calling a method, a program delegates that action to a delegate
- For example, we could have the following:

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

- This delegate type, named `Change`, can be instantiated with any method that takes an `int` as a parameter and returns an `int`
- For example, consider the following method:

```
static int fun1 (int x) {. . .}
```

- The delegate `change` can be instantiated by sending the name of this method to the delegate's constructor, as in the following:

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

- This can be shortened to the following:

```
Change chgfun1 = fun1;
```

- Following is an example call to `fun1` through the delegate `chgfun1`:

```
chgfun1(12);
```

- Objects of a delegate class can store more than one method. A second method can be added using the operator `+=`, as in the following

```
Change chgfun1 += fun2;
```

- Ada 95 has pointers to subprograms, but Java does **not**

9.8 Design Issues for Functions 396

- The following design issues are specific to functions:
 - Are side effects allowed?
 - What types of values can be returned?
 - How may 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
 - For example, 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
 - Most imperative programming languages restrict the types that can be returned by their functions
 - C allows **any** type to be returned by its functions **except** arrays and functions. Both of these can be handled by pointer type return values
 - C++ is like C but also allows user-defined types, or classes, to be returned from its functions
 - Java and C# methods can return any type (but because methods are not types, methods cannot be returned)
 - Python, Ruby, and Lua treat methods as first-class objects, so they can be returned, as well as any other class
 - JavaScript functions can be passed as parameters and returned from functions
- Number of return values
 - In most of languages, only a **single** value can be returned from a function
 - Ruby allows the return of more than one value from a method
 - Lua also allows functions to return **multiple** values
 - Such values follow the return statement as a comma-separated list, as in the following:

```
return 3, sum, index
```
 - If the function returned three values and all are to be kept by the caller, the function would be called as in the following example:

```
a, b, c = fun()
```
 - In F#, multiple values can be returned by placing them in a tuple and having the tuple be the last expression in the function

9.9 Overloaded Subprograms 397

- An overloaded **operator** is one that has multiple meanings
- The meaning of a particular instance of an overloaded operator is determined by its types of its operands
- For example, if the * operator
 - It 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
- Ada, Java, C++, and C# include predefined overloaded subprograms
 - For examples, overloaded constructors
 - Users are also allowed to write multiple versions of subprograms with the same
- 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
```

9.10 Generic Subprograms 398

- 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
- Subtype polymorphism means that a variable of type T can access any object of type T or any type derived from T (OOP languages)
- 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
 - 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;
}
```

- where `Type` is the parameter that specifies the type of data on which the function will operate
- For example, if it were instantiated with `int` as the parameter, it would be:

```
int max(int first, int second) {
    return first > second ? first : second;
}
```

- 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);
```

Generic Methods in Java 5.0

- Java's generic methods differ from the generic subprogram of C++ in several important ways:
 - Generic parameters in Java 5.0 must be **classes** – they cannot be primitive type
 - Java 5.0 generic methods are instantiated just once as truly generic methods
 - Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters. Such restrictions are called bounds
- As an example of a generic Java 5.0 method:

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

- This defines a method named `doIt` that takes an array of elements of a generic type
- The name the generic type is `T` and it must be an array
- An example call to `doIt`:

```
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 Methods in C# 2005

- The generic method of C# 2005 are similar in capability 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
- For example, consider the following skeletal class definition:

```
class MyClass {  
    public static T DoIt<T>(T p1) { . . . }  
    . . .  
}
```

- For example, both following class are legal:

```
int myInt = MyClass.DoIt(17);           // Calls DoIt<int>  
string myStr = MyClass.DoIt('apples'); // Calls DoIt<string>
```

9.11 User-Defined Overloaded Operators 404

- Operators can be overloaded in Ada, C++, Python, and Ruby (**not** carried over into Java)
- A Python example:

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

- The method is named `__add__`
- For example, the expression `x + y` is implemented as

```
x.__add__(y)
```

9.12 Closures 405

- 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 does not 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)
- Following is an example of a closure written in JavaScript:

```
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`
- The output of this code, assuming it was embedded in an HTML document and displed with a browser, is as follows:

```
Add 10 to 20: 30  
Add 5 to 20: 25
```

- In this example, the closuer is the **anonymous** function defined inside the `makeAdder` function, which `makceAdder` return
- The variable `x` referenced in the closure function is bound to the parameter what was sent to `makeAdder`
- The `makeAdder` function is called twice, once with a parameter of 10 and once with 5
- Each of these call returns a different version of the closure because they are bound to different values of `x`
- The first call to `makeAdder` creates a function that adds 10 to it parameter; the second creates a function that adds 5 to its parameter

- In C#, this same closure function can be written in C# using nested anonymous delegate
 - The type of the nesting method is specified to be a function that takes an `int` as a parameter and returns an anonymous **delegate**
 - The return type is specified with the special notation for such delegates, `Func<int, int>`
 - The first type in the angle brackets is the parameter type
 - The second type is the return type of the method encapsulated by the delegate

```
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));
```

- The output of this code is exactly the same as for the previous JavaScript closure example

```
Add 10 to 20: 30
Add 5 to 20: 25
```

9.13 Coroutines 407

- A coroutine is a subprogram that has **multiple** entries and controls them itself – supported directly in Lua
- The coroutine control mechanism is often called the symmetric control: caller and called coroutines are more equitable
- It also has the means to maintain their status between activation
- This means that coroutines must be **history sensitive** and thus have static local variables
- Secondary executions of a coroutine often begin at points other than its beginning
- The invocation of a coroutine is named a **resume** rather than a call
- 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

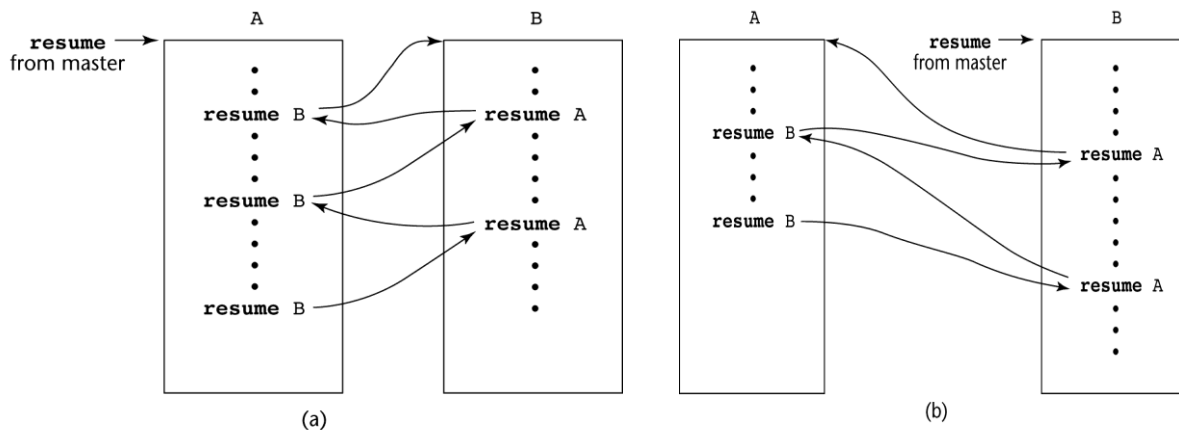


Figure 9.3 Two possible execution control sequences for two coroutines without loops

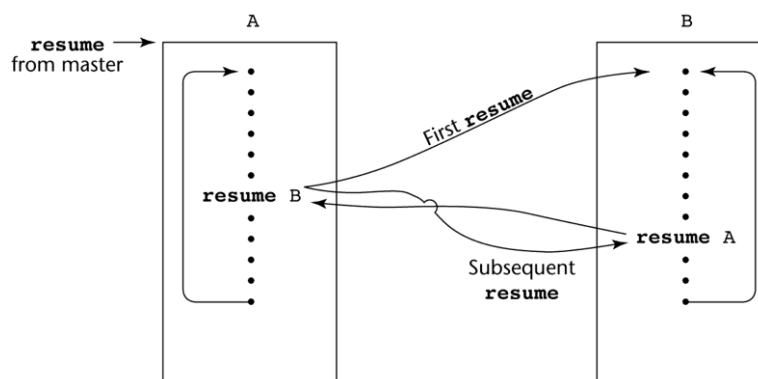


Figure 9.4 Coroutine execution sequence with loops

Summary 410

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
 - Functions return values and procedures do **not**
- Local variables in subprograms can be stack-dynamic or static
- JavaScript, Python, Ruby, and Lua allow subprogram definitions to be **nested**
- Three models of parameter passing: **in**-mode, **out**-mode, and **inout**-mode
- Five implementation models of parameter passing:
 - **Pass-by-value**: in-mode
 - **Pass-by-result**: out-mode
 - **Pass-by-value-result**: inout-mode
 - **Pass-by-reference**: inout-mode
 - **Pass-by-name**: inout-mode
- C and C++ support **pointers to functions**. C# has **delegates**, which are object that can store references to methods
- Ada, C++, C#, Ruby, and Python allow both subprogram and **operator overloading**
- Subprograms in C++, Java 5.0, and C# 2005 can be **generic**, using parametric polymorphism, so the desired types of their data objects can be passed to the **compiler**, which then can construct unit for the required types
- A closure is a subprogram and its reference environment
 - Closures are useful in languages that allow **nested** subprograms, are static-scoped, and allow subprograms to be returned from functions and assigned to variables
- A coroutine is a special subprogram with **multiple** entries