Chapter 5 Subprograms and Implementing Subprograms

Introduction :

- Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design.
- Two fundamental abstraction facilities can be included in a programming language: **process abstraction and data abstraction.**
- In the early history of high-level programming languages, only process abstraction was included.
- Process abstraction, in the form of subprograms, has been a central concept in all programming languages.
- In the 1980s, however, many people began to believe that data abstraction was equally important.
- The first programmable computer, Babbage's Analytical Engine, built in the 1840s, had the capability of reusing collections of instruction cards at several different places in a program. In a modern programming language, such a collection of statements is written as a subprogram. This reuse results in several different kinds of savings, primarily memory space and coding time. Such reuse is also an abstraction, for the details of the subprogram's computation are replaced in a program by a statement that calls the subprogram.
- The methods of object-oriented languages are closely related to the subprograms. The primary way methods differ from subprograms is the way they are called and their associations with classes and objects.

Fundamentals of Subprograms:

a) General Subprogram Characteristics:

1. Each subprogram has a single-entry point.
2. The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
3. Control always returns to the caller when the subprogram execution terminates.

b) Basic Definitions:

1. A **subprogram definition** describes the interface to and the actions of the subprogram abstraction.
2. A **subprogram call** is the explicit request that a specific 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 kinds of subprograms, procedures and functions, are defined.
3. A **subprogram header**, which is the first part of the definition, serves several purposes. First, it specifies that the following syntactic unit is a subprogram definition of some particular kind. In languages that have more than one kind of subprogram, the kind of the subprogram is usually specified with a special word. Second, if the subprogram is not anonymous, the header provides a name for the subprogram. Third, it may optionally specify a list of parameters.

Some of the header examples are :
In Python ---- def adder (parameters):
In C ---------- void adder (parameters)

4. **The body of subprograms** defines its actions. In the C-based languages (and some others—for example, JavaScript) the body of a subprogram is delimited by braces. In Ruby, an end statement terminates the body of a subprogram. Statements in the body of a Python function must be indented and the end of the body is indicated by the first statement that is not indented.
5. The **parameter profile** of a subprogram contains the number, order, and types of its formal parameters.
6. **The protocol of a subprogram** is its parameter profile plus, if it is a function, its return type. In languages in which subprograms have types, those types are defined by the subprogram's protocol.
7. Subprograms can have declarations as well as definitions. This form parallels the variable declarations and definitions in C, in which the declarations can be used to provide type information but not to define variables. **Subprogram declarations** provide the subprogram's protocol but do not include their bodies. They are necessary in languages that do not allow forward references to subprograms. In both the cases of variables and subprograms, declarations are needed for static type checking. In the case of subprograms, it is the type of the parameters that must be checked. Function declarations are common in C and C++ programs, where they are called prototypes. Such declarations are often placed in header files. In most other languages (other than C and C++), subprograms do not need declarations, because there is no requirement that subprograms be defined before they are called.

c) Parameters:

- Subprograms typically describe computations.
- There are two ways that a non-method subprogram can gain access to the data that it is to process: through direct access to nonlocal variables (declared elsewhere but visible in the subprogram) or through parameter passing.
- Data passed through parameters are accessed through names that are local to the subprogram. Parameter passing is more flexible than direct access to nonlocal variables (global variables).
- The parameters in the subprogram header are called **formal parameters**. They are sometimes thought of as dummy variables because they are not variables in the usual sense: In most cases, they are bound to storage only when the subprogram is called, and that binding is often through some other program variable.
- Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called **actual parameters**. They must be distinguished from formal parameters, because the two usually have different restrictions on their forms, and their uses are quite different.
- In nearly all programming languages, the correspondence between actual and formal parameters—or the binding of actual parameters to formal parameters—is done by position: The first actual parameter is bound to the first formal parameter and so forth. Such parameters are

called **positional parameters**. This is an effective and safe method of relating actual parameters to their corresponding formal parameters, as long as the parameter lists are relatively short.

- When lists are long, however, it is easy for a programmer to make mistakes in the order of actual parameters in the list. One solution to this problem is to provide **keyword parameters**, in which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call.
- The advantage of keyword parameters is that they can appear in any order in the actual parameter list. For e.g. Python functions can be called using this technique.

  Sum_func(length = my_length, list = my_array, sum = my_sum)
- The disadvantages to keyword parameters are:
  a) User of the subprogram must know the names of formal parameters.
  b) In this approach, after a keyword parameter appears in the list, all remaining parameters must be keyworded. This restriction is necessary because a position may no longer be well defined after a keyword parameter has appeared.
  c) In Python, Ruby, C++, Fortran 95+ Ada, and PHP, formal parameters can have **default values**. A default value is used if no actual parameter is passed to the formal parameter in the subprogram header. Consider the following Python function header in which exemptions parameter has default value as 1:

     def compute_pay(income, exemptions = 1, tax_rate)

d) Procedures and Functions:

- There are two distinct categories of subprograms—procedures and functions—both of which can be viewed as approaches to extending the language.
- All subprograms are collections of statements that define parameterized computations.
- Functions return values and procedures do not.
- In most languages that do not include procedures as a separate form of subprogram, functions can be defined not to return values and they can be used as procedures.
- The computations of a procedure are enacted by single call statements. In effect, procedures define new statements. For example, if a particular language does not have a sort statement, a user can build a procedure to sort arrays of data and use a call to that procedure in place of the unavailable sort statement.
- In Ada, procedures are called just that; in Fortran, they are called subroutines.
- Most other languages do not support procedures.

Design Issues for subprograms operations:

- Subprograms are complex structures in programming languages, and therefore there is a lengthy list of issues is involved in their design.

1. Are local variables statically or dynamically allocated?
2. Can subprogram definitions appear in other subprogram definitions?
3. What parameter-passing method or methods are used?

4. Are the types of the actual parameters checked against the types of the formal parameters?
5. If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
6. Can subprograms be overloaded?
   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. A **closure** is a nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program.
7. Can subprograms be generic?
8. If the language allows nested subprograms, are closures supported?

Local Referencing Environments:

a) Local Variables: (1. Are local variables statically or dynamically allocated?)

- Subprograms can define their own variables, thereby defining local referencing environments.
- 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.
- **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.
  1. The primary one being the flexibility they provide to the subprogram. For e.g. recursive subprograms have stack-dynamic local variables.
  2. 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:
  1. First, there is the cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call to the subprogram.
  2. Second, accesses to stack-dynamic local variables must be indirect, whereas accesses to static variables can be direct. This indirectness is required because the place in the stack where a particular local variable will reside can be determined only during execution.
  3. 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. It is sometimes convenient to be able to write history-sensitive subprograms. A common example of a need for a history-sensitive subprogram is one whose task is to generate pseudorandom numbers. Each call to such a subprogram computes one pseudorandom number, using the last one it computed. It must, therefore, store the last one in a static local variable. Coroutines and the subprograms used in iterator loop constructs are other examples of subprograms that need to be history sensitive.
- 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. Also, if

accessed directly, these accesses are obviously more efficient. And, of course, they allow subprograms to be history sensitive.

- The **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, locals are stack dynamic unless specifically declared to be static.
  For example, in the following C (or C++) function, the variable sum is static and count is stack dynamic.

```
int adder(int list[], int listlen) {
        static int sum = 0;
        int count;
         for (count = 0; count < listlen; count ++)
                sum += list [count];
        return sum;
}
```

  The methods of C++, Java, and C# have only stack-dynamic local variables.
  In Python, the only declarations used in method definitions are for globals. Any variable declared to be global in a method must be a variable defined outside the method. A variable defined outside the method can be referenced in the method without declaring it to be global, but such a variable cannot be assigned in the method. If the name of a global variable is assigned in a method, it is implicitly declared to be a local and the assignment does not disturb the global. All local variables in Python methods are stack dynamic.
  Only variables with restricted scope are declared in Lua. Any block, including the body of a function, can declare local variables with the local declaration, as in the following:

```
local sum
```

  All nondeclared variables in Lua are global.

b) Nested Subprograms: (2. Can subprogram definitions appear in other subprogram definitions?)

- The idea of nesting subprograms originated with Algol 60.
- The main aim was to be able to create a hierarchy of both logic and scopes. 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 only languages that allowed nested subprograms were Algol 60, Algol 68, Pascal, and Ada. Many other languages, including all of the direct descendants of C, do not allow subprogram nesting. Recently, some new languages such as JavaScript, Python, Ruby, and Lua. Also, most functional programming languages allow subprograms to be nested.

Parameter-Passing Methods (3. What parameter-passing method or methods are used?)

- Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms.
- Formal parameters are characterized by one of three distinct semantics models:
  1. They can receive data from the corresponding actual parameter (in mode).
  2. They can transmit data to the actual parameter (out mode).
  3. or they can do both (inout mode).
- Different parameter passing methods are:

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 variable in the subprogram, thus implementing in-mode semantics.
- Pass-by-value is normally implemented by copy, because accesses often are more efficient with this approach. It could be implemented by transmitting an access path to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell (one that can only be read). Enforcing the write protection is not always a simple matter. For example, suppose the subprogram to which the parameter was passed passes it in turn to another subprogram. C++ provides a convenient and effective method for specifying write protection on pass-by-value parameters that are transmitted by access path.
- The **advantage** of pass-by-value is that for scalars it is fast, in both linkage cost and access time.
- The main **disadvantage** of the pass-by-value method if copies are used is that 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. In addition, 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.
  For e.g. consider following C code:

```
int add(int x, int y) {
   x = x +y;
}
```

  Variable x is passed by value, so changes made in subprogram cannot be transferred to callee.

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 obviously must be a variable.
- **The pass-by-result method has the advantages and disadvantages of pass-by-value, plus some additional 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. As with pass-by-value, the difficulty of implementing pass-by-result by transmitting an access path usually results in it being implemented by copy. In this case, the problem is in ensuring that the initial value of the actual parameter is not used in the called

subprogram. One additional problem with the pass-by-result model is that there can be an actual parameter collision, for e.g.

```
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 with the out specifier on its formal parameter.5

```
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.Because the order can be implementation dependent for some languages, different implementations can produce different results.

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. In fact, pass-by-value-result formal parameters must have local storage associated with the called subprogram. At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.
- 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.
- The **advantages** of pass-by-value-result are relative to pass-by-reference.

4) Pass-by-Reference:

- Pass-by-reference is a second implementation model for inout-mode parameters.
- Rather than copying data values back and forth, however, as in pass-byvalue-result, the pass-by-reference 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. Thus, the called subprogram is allowed to access the actual parameter in the calling program unit. In effect, the actual parameter is shared with the called subprogram.

- The **advantage** of pass-by-reference is that the passing process itself is efficient, in terms of both time and space. Duplicate space is not required, nor is any copying required.
- There are several **disadvantages** to the pass-by-reference method.
  a) First, access to the formal parameters will be slower than pass-by-value parameters, because of the additional level of indirect addressing that is required.
  b) Second, if only one-way communication to the called subprogram is required, inadvertent and erroneous changes may be made to the actual parameter.
  c) Another problem of pass-by-reference is that aliases can be created. This problem should be expected, because pass-by-reference makes access paths available to the called subprograms, thereby providing access to nonlocal variables. The problem with these kinds of aliasing is the same as in other circumstances: It is harmful to readability and thus to reliability. It also makes program verification more difficult.
- Consider a C or C++ function that has two parameters that are to be passed by reference
  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:

- Pass-by-name is an inout-mode parameter transmission method 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.
- This method is quite different from other ways, formal parameters are bound to actual values or addresses at the time of the subprogram call.
- A pass-by-name 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.
- Implementing a pass-by-name parameter requires a subprogram to be passed to the called subprogram to evaluate the address or value of the formal parameter. The referencing environment of the passed subprogram must also be passed. This subprogram/referencing environment is a closure
- 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.
- Pass-by-name is not part of any widely used language.

Type Checking Parameters: (4. Are the types of the actual parameters checked against the types of the formal parameters?)

1. 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.
2. 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.
3. 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. Although an integer 1 and a floating-point 1 have the same value, the representations of these two are very different. sub1 cannot produce a correct result given an integer actual parameter value when it expects a floating-point value.

4. Early programming languages, such as Fortran 77 and the original version of C, did not require parameter type checking; most later languages require it. However, the relatively recent languages Perl, JavaScript, and PHP do not.
5. In C99 and C++, all functions must have their formal parameters in prototype form. However, type checking can be avoided for some of the parameters by replacing the last part of the parameter list with an ellipsis, as follows:

   int printf(const char* format_string, . . .);
6. In Python and Ruby, there is no type checking of parameters, because typing in these languages is a different concept. Objects have types, but variables do not, so formal parameters are typeless. This disallows the very idea of type checking parameters.

Parameters that are Subprograms:

(5. If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?)

7. In programming, a number of situations occur that are most conveniently handled if subprogram names can be sent as parameters to other subprograms.
8. One common example of these occurs when a subprogram must sample some mathematical function. 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. When such a subprogram is written, it should be usable for any given function; it should not need to be rewritten for every function that must be integrated. It is therefore natural that the name of a program function that evaluates the mathematical function to be integrated be sent to the integrating subprogram as a parameter.
9. To pass subprogram as a parameter, there are two complications:
   1. 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. Fortran 95+ has a mechanism for providing types of parameters for subprograms that are passed as parameters, and they must be checked.
   10. 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:
       a. The environment of the call statement that enacts the passed subprogram (shallow binding)
       b. The environment of the definition of the passed subprogram (deep binding)
       c. The environment of the call statement that passed the subprogram as an actual parameter (ad hoc binding)

- For e.g, subprogram as parameter in javascript,

```
function sayHello(param) {
        console.log("hello", param);
        param();
        return "Hiii all"
}

// Function address
function smaller() {
        console.log("Is everything alright")
}

// Function call
const returnHello = sayHello(smaller)
console.log(returnHello)
```

Output :

```
hello [Function: smaller]
Is everything alright
Hiii all
```

Overloaded Subprograms: (6. Can subprograms be overloaded?)

- 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). It is not necessary, overloaded subprograms usually implement the same process.
- C++, Java, Ada, and C# include predefined overloaded subprograms. For example, many classes in C++, Java, and C# have overloaded constructors.
- Users are also allowed to write multiple versions of subprograms with the same name in Ada, Java, C++, C#, and F#. Once again, in C++, Java, and C# the most common user-defined overloaded methods are constructors.

Generic Subprograms: (7. an subprograms be generic?)

- One way to increase the reusability of software is to lessen the need to create different subprograms that implement the same algorithm on different types of data. For example, a programmer should not need to write four different sort subprograms to sort four arrays that differ only in element type.
- A polymorphic subprogram takes parameters of different types on different activations.
- Overloaded subprograms provide a particular kind of polymorphism called ad hoc polymorphism.
- Overloaded subprograms need not behave similarly.
- Languages that support object-oriented programming usually support subtype polymorphism. Subtype polymorphism means that a variable of type T can access any object of type T or any type derived from T.
- A more general kind of polymorphism is provided by the methods of Python and Ruby. Recall that variables in these languages do not have types, so formal parameters do not have types. Therefore, a method will work for any type of actual parameter, as long as the operators used on the formal parameters in the method are defined.
- Parametric polymorphism is provided by a subprogram that takes generic parameters that are used in type expressions that describe the types of the parameters of the subprogram.
- Different instantiations of such subprograms can be given different generic parameters, producing subprograms that take different types of parameters. Parametric definitions of subprograms all behave the same.
- Parametrically polymorphic subprograms are often called generic subprograms. Ada, C++, Java 5.0+, C# 2005+, and F# provide a kind of compile-time parametric polymorphism.

Generic Functions in C++:

- Generic functions in C++ have the descriptive name of template functions.
- The definition of a template function has the general form
  template <template parameters>
  —a function definition that may include the template parameters
- A template parameter (there must be at least one) has one of the forms
  class identifier
  typename identifier
- The class form is used for type names. The typename form is used for passing a value to the template function. For example, it is sometimes convenient to pass an integer value for the size of an array in the template function.
- As an example of a template function, consider the following:

```
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.

This template function can be instantiated for any type for which the operator > is defined. 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;
}
```

Generic Methods in Java 5.0:

- Support for generic types and methods was added to Java in Java 5.0.
- The name of a generic class in Java 5.0 is specified by a name followed by one or more type variables delimited by pointed brackets. For example,

  generic_class<T>

  where T is the type variable.
- Java's generic methods differ from the generic subprograms of C++ in several important ways.
  1. First, generic parameters must be classes—they cannot be primitive types. But in C++, the component types of arrays are generic and can be primitives.
  2. Second, although Java generic methods can be instantiated any number of times, only one copy of the code is built. The internal version of a generic method, which is called a raw method, operates on Object class objects. At the point where the generic value of a generic method is returned, the compiler inserts a cast to the proper type.
  3. Third, in Java, 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.
- For e.g. method definition:

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

  This defines a method named showElement that takes an array of elements of a generic type. The name of the generic type is T and it must be an array. Following is an example call to showElement:

  ```
  showElement<String>(myList);
  ```

Design Issues for Functions:

The following design issues are specific to functions:

1. Are side effects allowed?
2. What types of values can be returned?
3. How many values can be returned?

1. Functional Side Effects:

- Due the problems of side effects of functions that are called in expressions,, parameters to functions should always be in-mode parameters.

- In fact, some languages require this; for example, Ada functions can have only in-mode formal parameters. This requirement effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals.
- In most other imperative languages, however, functions can have either pass-by-value or pass-by-reference parameters, thus allowing functions that cause side effects and aliasing.
- Pure functional languages, such as Haskell, do not have variables, so their functions cannot have side effects.

2. 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.
- Ada, Python, Ruby, and Lua are the only languages among current imperative languages whose functions (and/or methods) can return values of any type. In the case of Ada, however, because functions are not types in Ada, they cannot be returned from functions. Of course, pointers to functions can be returned by functions.
- In some programming languages, subprograms are first-class objects, which means that they can be passed as parameters, returned from functions, and assigned to variables. Methods are first-class objects in some imperative languages, for example, Python, Ruby, and Lua. The same is true for the functions in most functional languages.
- Neither Java nor C# can have functions, but their methods are similar to functions. In both, any type or class can be returned by methods. Because methods are not types, they cannot be returned.

3. Number of Returned Values:

- In most languages, only a single value can be returned from a function.
- Ruby allows the return of more than one value from a method. If a return statement in a Ruby method is not followed by an expression, nil is returned. If followed by one expression, the value of the expression is returned. If followed by more than one expression, an array of the values of all of the expressions is returned.
- 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
```

- The form of the statement that calls the function determines the number of values that are received by the caller. If the function is called as a procedure, that is, as a statement, all return values are ignored. 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.

User-Defined Overloaded Operators:

- Operators can be overloaded by the user in Ada, C++, Python, and Ruby.
- Suppose that a Python class is developed to support complex numbers and arithmetic operations on them. A complex number can be represented with two floating-point values. The Complex class would have members for these two named real and imag.
- In Python, binary arithmetic operations are implemented as method calls sent to the first operand, sending the second operand as a parameter. For addition, the method is named __add__. For example, the expression x + y is implemented as x.__add__(y). To overload + for the addition of objects of the new Complex class, we only need to provide Complex with a method named __add__ that performs the operation. Following is such a method:

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

- In most languages that support object-oriented programming, a reference to the current object is implicitly sent with each method call. In Python, this reference must be sent explicitly; therefore self is the first parameter to our method, __add__.
- The example add method could be written for a complex class in C++ as follows:

```
Complex operator +(Complex &second) {
        return Complex(real + second.real, imag + second.imag);
    }
```

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. They also have the means to maintain their status between activations. 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. 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. Rather than executing to its end, a coroutine often partially executes and then transfers control to some other coroutine, and when restarted, a coroutine resumes execution just after the statement it used to transfer control elsewhere. In the case of coroutines, this is called as quasi-concurrency.
- Typically, coroutines are created in an application by a program unit called the master unit, which is not a coroutine. When created, coroutines execute their initialization code and then return control to that master unit. When the entire family of coroutines is constructed, the master program resumes one of the coroutines, and the members of the family of coroutines then resume each other in some order until their work is completed. If the execution of a coroutine reaches the end of its code section, control is transferred to the master unit that created it. This is the mechanism for ending execution of the collection of coroutines, when that is desirable.
- In some programs, the coroutines run whenever the computer is running.
- One example of a problem that can be solved with this sort of collection of coroutines is a card game simulation. Suppose the game has four players who all use the same strategy. Such a game can be simulated by having a master program unit create a family of coroutines, each with a collection, or hand, of cards. The master program could then start the simulation by resuming one of the player coroutines, which, after it had played its turn, could resume the next player coroutine, and so forth until the game ended.

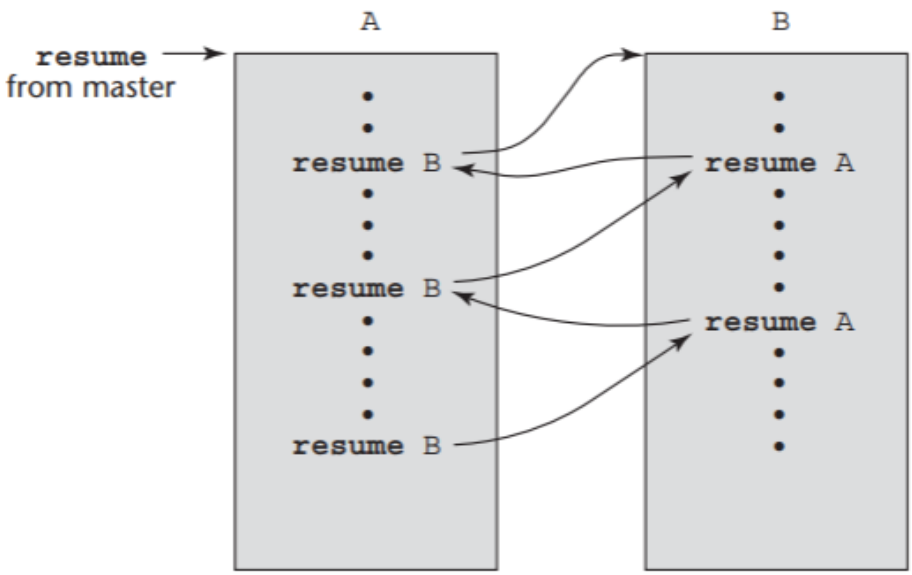


Figure a)

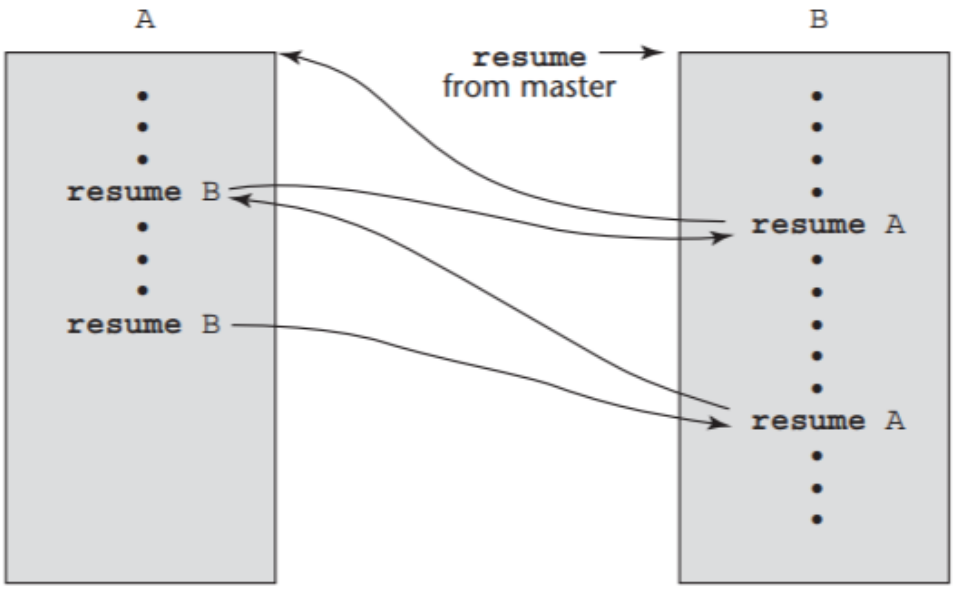


Figure b)

Suppose program units A and B are coroutines. Above figure shows two ways an execution sequence involving A and B might proceed. In Figure a), the execution of coroutine A is started by the master unit. After some execution, A starts B. When coroutine B in a) 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 b) shows an alternative execution sequence of coroutines A and B. In this case, B is started by the master unit.

Implementing Subprograms:

a) The General Semantics of Calls and Returns:

- The subprogram call and return operations of a language are together called its subprogram linkage.
- A subprogram call in a typical language has numerous actions associated with it.
- The call must include the mechanism for whatever parameter-passing method is used.
- If local vars are not static, the call must cause storage to be allocated for the locals declared in the called subprogram and bind those vars to that storage.
- It must save the execution status of the calling program unit.
- It must arrange to transfer control to the code of the subprogram and ensure that control to the code of the subprogram execution is completed.
- Finally, if the language allows nested subprograms, the call must cause some mechanism to be created to provide access to non-local vars that are visible to the called subprogram.

b) Implementing "Simple" Subprograms:

- Simple means that subprograms cannot be nested and all local vars are static.
- The semantics of a call to a simple subprogram requires the following actions:
    1. Save the execution status of the caller.
    2. Carry out the parameter-passing process.
    3. Pass the return address to the callee.
    4. Transfer control to the callee.

▪ The semantics of a return from a simple subprogram requires the following actions:

1. If pass-by-value-result parameters are used, move the current values of those parameters to their corresponding actual parameters.
2. If it is a function, move the functional value to a place the caller can get it.
3. Restore the execution status of the caller.
4. Transfer control back to the caller.

The call and return actions require storage for the following:

– Status information of the caller,
– parameters,
– return address, and
– functional value (if it is a function)

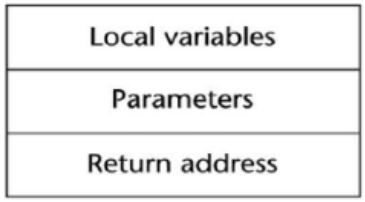
▪ These, along with the local vars and the subprogram code, form the complete set of information a subprogram needs to execute and then return control to the caller.

▪ A simple subprogram consists of two separate parts:

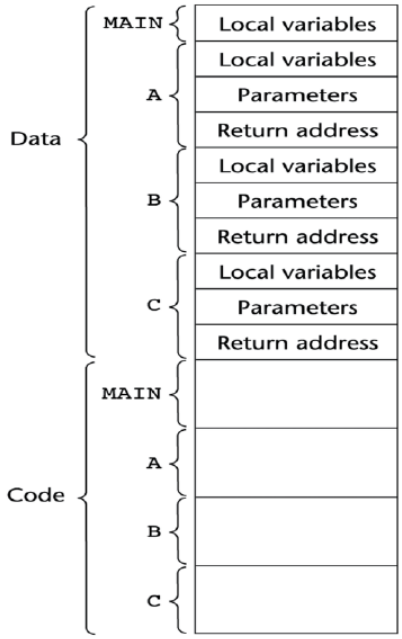
– The actual code of the subprogram, which is constant, and

– The local variables and data, which can change when the subprogram is executed. “Both of which have fixed sizes.”

- The format, or layout, of the non-code part of an executing subprogram is called an activation record, b/c the data it describes are only relevant during the activation of the subprogram.
- The form of an activation record is static.
- An activation record instance is a concrete example of an activation record (the collection of data for a particular subprogram activation).
- Because languages with simple subprograms do not support recursion; there can be only one active version of a given subprogram at a time.
- Therefore, there can be only a single instance of the activation record for a subprogram.
- One possible layout for activation records is shown below.



- Because an activation record instance for a simple subprogram has a fixed size, it can be statically allocated.
- The following figure shows a program consisting of a main program and three subprograms: A, B, and C.



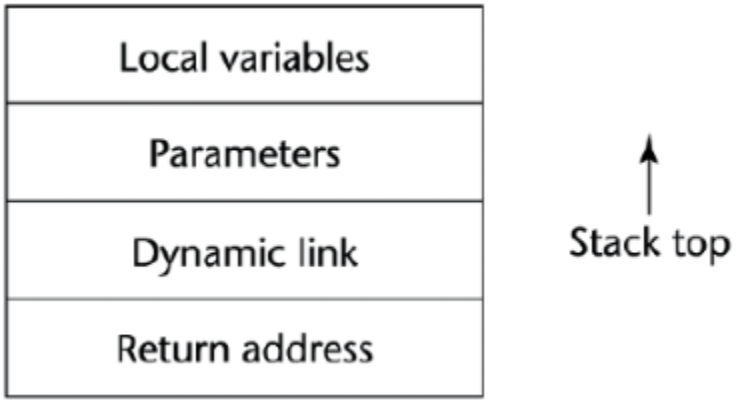
- The construction of the complete program shown above is not done entirely by the compiler.
- In fact, b/c of independent compilation, MAIN, A, B, and C may have been compiled on different days, or even in different years.
- At the time each unit is compiled, the machine code for it, along with a list of references to external subprograms is written to a file.
- The executable program shown above is put together by the linker, which is part of the O/S.
- The linker was called for MAIN, and the linker had to find the machine code programs A, B, and C, along with their activation record instances, and load them into memory with the code for MAIN.

c) Implementing Subprograms with Stack-Dynamic Local Variables:

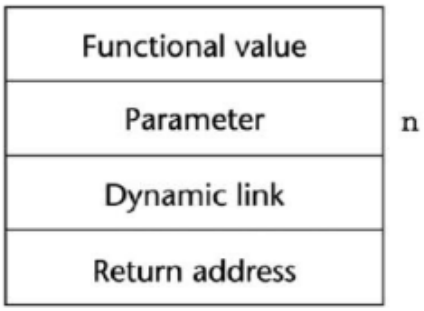
- One of the most important advantages of stack-dynamic local vars is support for recursion.
- Subprogram linkage in languages that use stack-dynamic local vars are more complex than the linkage of simple subprograms for the following reasons:
    1. The compiler must generate code to cause the implicit allocation and deallocation of local variables
    2. Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram), which means there can be more than one instance of a subprogram at a given time, with one call from outside the subprogram and one or more recursive calls.
    3. Recursion, therefore, requires multiple instances of activation records, one for each subprogram activation that can exist at the same time.

4. Each activation requires its own copy of the formal parameters and the dynamically allocated local vars, along with the return address.

- The format of an activation record for a given subprogram in most languages is known at compile time.
- In many cases, the size is also known for activation records b/c all local data is of fixed size.
- In languages with stack-dynamic local vars, activation record instances must be created dynamically. The following figure shows the activation record for such a language.



- Because the return address, dynamic link, and parameters are placed in the activation record instance by the caller, these entries must appear first.
- The return address often consists of a ptr to the code segment of the caller and an offset address in that code segment of the instruction following the call.
- The dynamic link points to the top of an instance of the activation record of the caller.
- In static-scoped languages, this link is used in the destruction of the current activation record instance when the procedure completes its execution.
- The stack top is set to the value of the old dynamic link.
- The actual parameters in the activation record are the values or addresses provided by the caller.
- Local scalar vars are bound to storage within an activation record instance.
- Local structure vars are sometimes allocated elsewhere, and only their descriptors and a ptr to that storage are part of the activation record.
- Local vars are allocated and possibly initialized in the called subprogram, so they appear last.
- For the recursive functions, there is an additional entry in the activation record - the return value of the function. The format of the activation record is as follows:

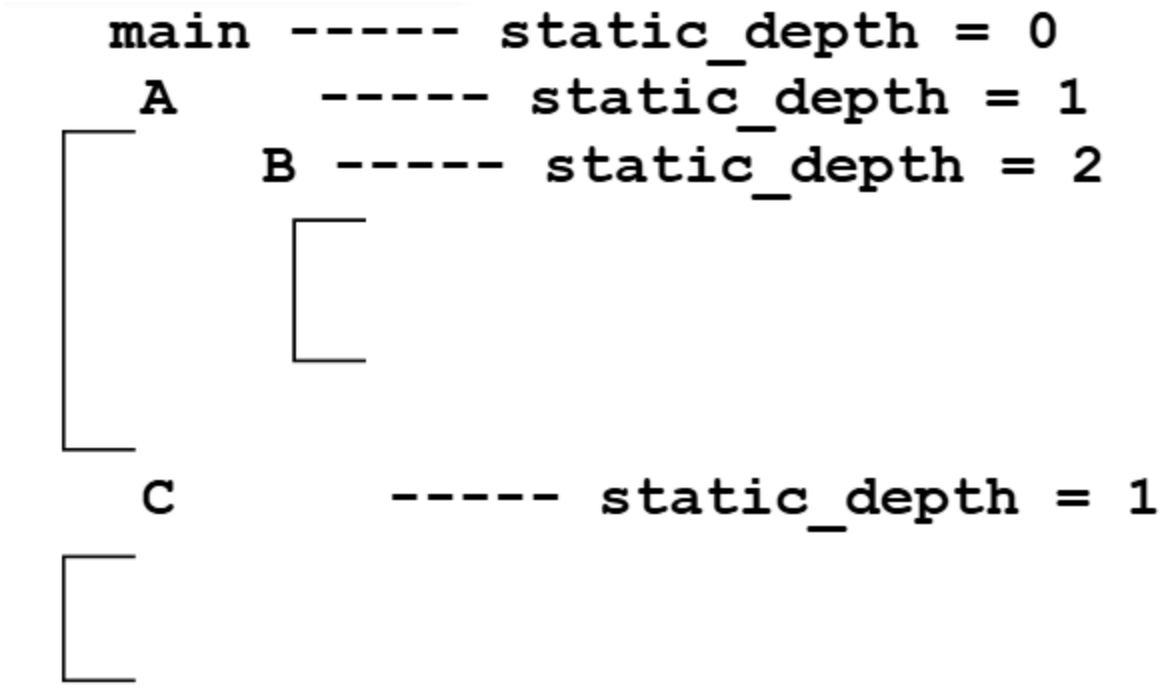


d) Nested Subprograms:

- Some of the non-C-based static-scoped languages (e.g., Fortran 95, Ada, JavaScript) use stack-dynamic local variables and allow subprograms to be nested.
- All variables that can be non-locally accessed reside in some activation record instance in the stack.
- The process of locating a non-local reference is as follows:
    1. Find the correct activation record instance in the stack in which the var was allocated.
    2. Determine the correct offset of the var within that activation record instance to access it.
- Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made.
- A subprogram is callable only when all of its static ancestor subprograms are active.
- The semantics of non-local references dictates that the correct declaration is the first one found when looking through the enclosing scopes, most closely nested first.

• Static Chains

- A static chain is a chain of static links that connects certain activation record instances in the stack.
- The static link, static scope pointer, in an activation record instance for subprogram 'A' points to one of the activation record instances of A's static parent.
- The static link appears in the activation record below the parameters.
- The static chain from an activation record instance connects it to all of its static ancestors.
- During the execution of a procedure P, the static link of its activation record instance points to an activation of P's static program unit.
- That instance's static link points, in turn, to P's static grandparent program unit's activation record instance, if there is one. Therefore, the static chain links all the static ancestors of an executing subprogram, in order of static parent first.
- This chain can obviously be used to implement the access to non-local vars in static-scoped languages.
- When a reference is made to a non-local var, the ARI containing the var can be found by searching the static chain until a static ancestor ARI is found that contains the var.
- Because the nesting scope is known at compile time, the compiler can determine not only that a reference is non-local but also the length of the static chain must be followed to reach the ARI that contains the non-local object.
- A static_depth is an integer associated with a static scope whose value is the depth of nesting of that scope.



- The length of the static chain needed to reach the correct ARI for a non-local reference to a var X is exactly the difference between the static_depth of the procedure containing the reference to X and the static_depth of the procedure containing the declaration for X .
- The difference is called the nesting_depth, or chain_offset, of the reference.
- The actual reference can be represented by an ordered pair of integers (chain_offset, local_offset), where chain_offset is the number of links to the correct ARI.
- Consider the following procedure:

```
procedure A is
        procedure B is
                procedure C is
                …
                end; // C
                …
        end; // B
        …
end; // A
```

- The static_depths of A, B, and C are 0, 1, 2, respectively. If procedure C references a var in A, the chain_offset of that reference would be 2 (static_depth of C minus the static_depth of A).
- If procedure C references a var in B, the chain_offset of that reference would be 1 (static_depth of C minus the static_depth of B).
- References to locals can be handled using the same mechanism, with a chain_offset of 0.

Blocks:

In programming, a **block** is a set of statements that are grouped and treated as a single unit. Blocks are used to define the scope of variables, control the flow of execution in conditional statements and loops, and encapsulate code in functions, methods, or classes.

Characteristics of Blocks:

1. **Encapsulation**: Blocks encapsulate a set of statements, allowing them to be treated as a single unit.
2. **Scope**: Blocks define a scope, which is a region of code where a variable can be accessed and manipulated. Variables declared within a block are typically only accessible within that block.
3. **Control Structures**: Blocks are used with control structures such as if, else, for, while, do-while, and switch to group multiple statements and control the flow of execution.
4. **Functions and Methods**: In programming languages that support functions and methods, blocks are used to define the body of the function or method.

Types of Blocks in Programming:

Here are some common types of blocks in programming:

1. Basic Block

A basic block is a sequence of instructions in a program with a single entry point and a single exit point. It usually doesn't contain any jump or branch instructions.

```
x = 10
y = 20
z = x + y
```

2. Function Block

A function block contains a set of instructions that perform a specific task. It starts with a function definition and ends with a return statement.

```
def add_numbers(a, b):
  result = a + b
  return result
```

3. Conditional Block

A conditional block contains code that is executed based on a certain condition. It is usually defined using if, elif, and else statements.

```
x = 10
if x > 5:
  print("x is greater than 5")
else:
  print("x is less than or equal to 5")
```

4. Loop Block

A loop block contains code that is executed repeatedly as long as a certain condition is true. It is defined using for and while loops.

For Loop

```
for i in range(5):
    print(i)
```

While Loop

```
x = 0
while x < 5:
    print(x)
    x += 1
```

5. Try-Except Block

A try-except block is used for exception handling. The code inside the try block is executed, and if an exception occurs, the code inside the except block is executed.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

6. Class Block

A class block contains the definition of a class in object-oriented programming. It can contain attributes and methods.

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print(f"{self.name} says Woof!")
```

7. Scope Block

A scope block defines the visibility and accessibility of variables within a program. In Python, indentation is used to define the scope of variables.

```
x = 10  # Global variable

def my_function():
    y = 20  # Local variable
    print(x)  # Access global variable
    print(y)  # Access local variable

my_function()
```

In the example above, x is a global variable, and y is a local variable defined within the scope of the my_function() block.

8. Nested Blocks

Nested blocks refer to blocks that are defined within another block. They can be found within loops, conditional statements, or function blocks.

Nested Loops

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

Nested Conditional Statements

```
x = 10
if x > 5:
    if x < 15:
        print("x is between 5 and 15")
```

Implementing Dynamic Scoping:

- Scoping is how you search for a variable with a given name.
- A variable has a *scope* which is the whole area in which that variable can be accessed by name. If there is a reference to a variable "a" then how does the compiler or interpreter find it?
- In lexical scoping , it searches in the local function (the function which is running now), then it searches in the function (or scope) in which that function was *defined,* then it searches in the function (scope) in which *that* function was defined, and so forth. "Lexical" here refers to *text,* in that interpreter can find out what variable is being referred to by looking at the nesting of scopes in the program text.
- In *dynamic* scoping, by contrast, search is made in the local function first, then search in the function that *called* the local function, then search in the function that called *that* function, and so on, up the call stack. "Dynamic" refers to *change,* in that the call stack can be different every

time a given function is called, and so the function might hit different variables depending on where it is called from.

- Dynamic scoping is useful as a substitute for globally scoped variables. It can also have a use similar to a ContextObject, as in JavaServlets and EJB APIs, holding state for a series of subroutines.
- **The two most popular methods for implementing Dynamic Scoping are Deep Binding and Shallow Binding.**
- In deep binding:
    - Nonlocal references are found by searching the activation record instances on the dynamic chain
    - Length of chain cannot be statically determined
    - Every activation record instance must have variable names
- In shallow binding:
    - Put locals in a central place
    - Methods:
        - One stack for each variable name
        - Central table with an entry for each variable name
