

CSPC31: Principles^s of Programming Languages^s

Dr. R. Bala Krishnan

Asst. Prof.

Dept. of CSE

NIT, Trichy – 620 015

Ph: 999 470 4853

E-Mail: balakrishnan@nitt.edu

Books

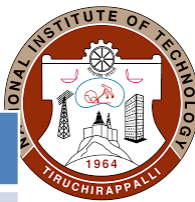
- **Text Books**

- ✓ Robert W. Sebesta, *“Concepts of Programming Languages”*, Tenth Edition, Addison Wesley, 2012.
- ✓ Michael L. Scott, *“Programming Language Pragmatics”*, Third Edition, Morgan Kaufmann, 2009.

- **Reference Books**

- ✓ Allen B Tucker, and Robert E Noonan, *“Programming Languages – Principles and Paradigms”*, Second Edition, Tata McGraw Hill, 2007.
- ✓ R. Kent Dybvig, *“The Scheme Programming Language”*, Fourth Edition, MIT Press, 2009.
- ✓ Jeffrey D. Ullman, *“Elements of ML Programming”*, Second Edition, Prentice Hall, 1998.
- ✓ Richard A. O'Keefe, *“The Craft of Prolog”*, MIT Press, 2009.
- ✓ W. F. Clocksin, C. S. Mellish, *“Programming in Prolog: Using the ISO Standard”*, Fifth Edition, Springer, 2003.

Chapters



Chapter No.	Title
1.	Preliminaries
2.	Evolution of the Major Programming Languages
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
5.	Names, Binding, Type Checking and Scopes
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages



Chapter 9 – Subprograms

Objectives

- Design of subprograms
 - Parameter passing methods
 - Local referencing environments
 - Overloaded subprograms
 - Generic subprograms
 - Aliasing
 - Side-effect problems

Miscellaneous

```

1000 #include<iostream.h>
1001 #include<conio.h>
1002 #include<stdlib.h>
1003 int add(int, int);
1004 int subtract(int, int);
1005 void main()
1006 {
1007     int a = 5, b = 10;
1008     int Summation = add(a, b);
1009 }

```

```

1011 int add(int x, int y)
1012 {
1013     int res;
1014     res = subtract(x, y);
1015     return res;
1016 }
1017 int subtract(int w, int v)
1018 {
1019     int res;
1020     res = w - v;
1021     return res;
1022 }

```

Stack M/o

x, y
y = 10
x = 5
1014
a, b
b = 10
a = 5
1008

add(a, b); -> {1008, [a = 5, b = 10], [a, b]}
 Subtract(a,b); -> {1014, [x = 5, y = 10], [x, y]}

Store the Return Address, Values of Local Variables, **Parameters in Stack Memory**

Miscellaneous

```

1000 #include<iostream.h>
1001 #include<conio.h>
1002 #include<stdlib.h>
1003 int add(int, int);
1004 int subtract(int, int);
1005 void main()
1006 {
1007     int a = 5, b = 10;
1008     int Summation = add(a, b);
1009 }
    
```

```

1011 int add(int x, int y)
1012 {
1013     int res;
1014     res = subtract(x, y);
1015     return res;
1016 }
1017 int subtract(int w, int v)
1018 {
1019     int res;
1020     res = w - v;
1021     return res;
1022 }
    
```

Stack M/o

v = 10
w = 5
Subtract(x, y)
y = 10
x = 5
add(a, b)
b = 5
a = 5
main()

add(a, b); -> {1008, [a = 5, b = 10], [a, b]}
 Subtract(a,b); -> {1014, [x = 5, y = 10], [x, y]}

Store the Return Address, Values of Local Variables, **Parameters in Stack Memory**

Introduction

- Two fundamental abstraction facilities -> Process Abstraction and Data Abstraction
- First programmable computer, Babbage's Analytical Engine (built in 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 -> Memory Space and Coding Time
- Instead of describing how some computation is to be done in a program, that description (the collection of statements in the subprogram) is enacted by a call statement, effectively abstracting away the details

Introduction

- This increases the readability of a program by emphasizing its logical structure while hiding the low-level details
- Methods of object-oriented languages are closely related to the subprograms
- Primary way methods differ from subprograms is the way they are called and their associations with classes and objects

Fundamentals of Subprogram

- Each subprogram has a single entry point
- Calling program unit is suspended during the execution of the called subprogram
 - Implies that there is only one subprogram in execution at any given time
- Control always returns to the caller when the subprogram execution terminates

Basic Definitions

- Subprogram Definition -> Describes the interface to and the actions of the subprogram abstraction
 - Subprogram Call -> 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
 - Subprogram Header
 - First, it specifies that the following syntactic unit is a subprogram definition of some particular kind
 - Second, if the subprogram is not anonymous, the header provides a name for the subprogram
 - Third, it may optionally specify a list of parameters
- `void addr (parameters)`

Basic Definitions

- Python -> **def** statements are executable
 - When a def statement is executed, it assigns the given name to the given function body
 - Until a function's def has been executed, the function cannot be called

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

- 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
- If the else clause is chosen, its version of the function can be called but the one in the then clause

Basic Definitions

- All Lua functions are anonymous

```
function cube(x) return x * x * x end
```

```
cube = function (x) return x * x * x end
```

- Parameter Profile of a subprogram contains the number, order and types of its formal parameters
- Protocol of a subprogram is its parameter profile plus, if it is a function, its return type
- Subprograms can have declarations as well as definitions

Parameters

- Subprograms typically describe computations
- 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)
 - 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
- Parameters in the subprogram header are called formal parameters
 - Dummy variables -> Not variables in the usual sense
 - Bound to storage only when the subprogram is called

Parameters

- Actual Parameters
- Binding of actual parameters to formal parameters is done by position -> Positional Parameters

`a = sum(a, b, c)`

`function sum (int x, int y, int z)`

- Keyword Parameters

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

- Default value is used, if no actual parameter is passed to the formal parameter in the subprogram header

```
def compute_pay(income, exemptions = 1, tax_rate)
```

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

Procedures and Functions

- Two distinct categories of subprograms -> Procedures and Functions
- 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 procedure has formal parameters that allow the transfer of data to the caller, those parameters can be changed. **Eg:** Sorting of array elements

Procedures and 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
- Functions define new user-defined operators

`float power(float base, float exp)`

`result = 3.4 * power(10.0, x)`

Design Issues for Subprograms

- Choice of one or more parameter-passing methods that will be used
- Whether the types of actual parameters will be type checked against the types of the corresponding formal parameters
- Whether local variables are statically or dynamically allocated
- Whether subprogram definitions can be nested
- Whether subprogram names can be passed as parameters
- Whether subprograms can be overloaded or generic

Design Issues for Subprograms

- Overloaded Subprogram -> Has the same name as another subprogram in the same referencing environment
- Generic Subprogram -> Computation can be done on data of different types in different calls
- Closure -> Nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program

Design Issues for Subprograms



- Are local variables statically or dynamically allocated?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter-passing method or methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprograms be generic?
- If the language allows nested subprograms, are closures supported?

Local Referencing Environments



- Local Variables
- Subprograms can define their own variables, thereby defining local referencing environments
- Variables that are defined inside subprograms are called local variables
 - 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
- **Adv of Stack Dynamic Local Variables:**
 - Flexibility they provide to the subprogram -> Recursive
 - Storage for local variables in an active subprogram can be shared with the local variables in all inactive subprograms

Local Variables

- **Disadv of Stack Dynamic Local Variable:**
 - Cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call to the subprogram
 - Accesses to stack-dynamic local variables must be indirect, whereas accesses to static variables can be direct -> Indirectness is required because the place in the stack where a particular local variable will reside can be determined only during execution
 - When all local variables are stack dynamic, subprograms cannot be history sensitive -> Cannot retain data values of local variables between calls

Local Variables



- History-sensitive subprograms
 - Generate pseudorandom numbers -> Store the last one in a static local variable
 - Coroutines and the subprograms used in iterator loop constructs
- Adv of static local variable:
 - More efficient—they require no run-time overhead for allocation and deallocation
 - If accessed directly, these accesses are obviously more efficient
 - Allow subprograms to be history sensitive
- Disadvantage of static local variables
 - Cannot support recursion
 - 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

Local Variables

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

- **Python:**
 - 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

Nested Subprograms

- Idea of nesting subprograms originated with Algol 60
- If a subprogram is needed only within another subprogram, why not place it there and hide it from the rest of the program?
- Descendants of Algol 60 -> Algol 68, Pascal and Ada
- Direct descendants of C -> Do not allow subprogram nesting
- JavaScript, Python, Ruby, and Lua -> Allow
- Most functional programming languages allow subprograms to be nested

```
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();  
};
```

Parameter Passing Methods

- Ways in which parameters are transmitted to and/or from called subprograms
- First, focus on the different semantics models of parameter-passing methods
- Then, discuss the various implementation models invented by language designers for these semantics models
- Next, survey the design choices of several languages and discuss the actual methods used to implement the implementation models
- Finally, consider the design considerations that face a language designer in choosing among the methods

Semantic Models of Parameter passing

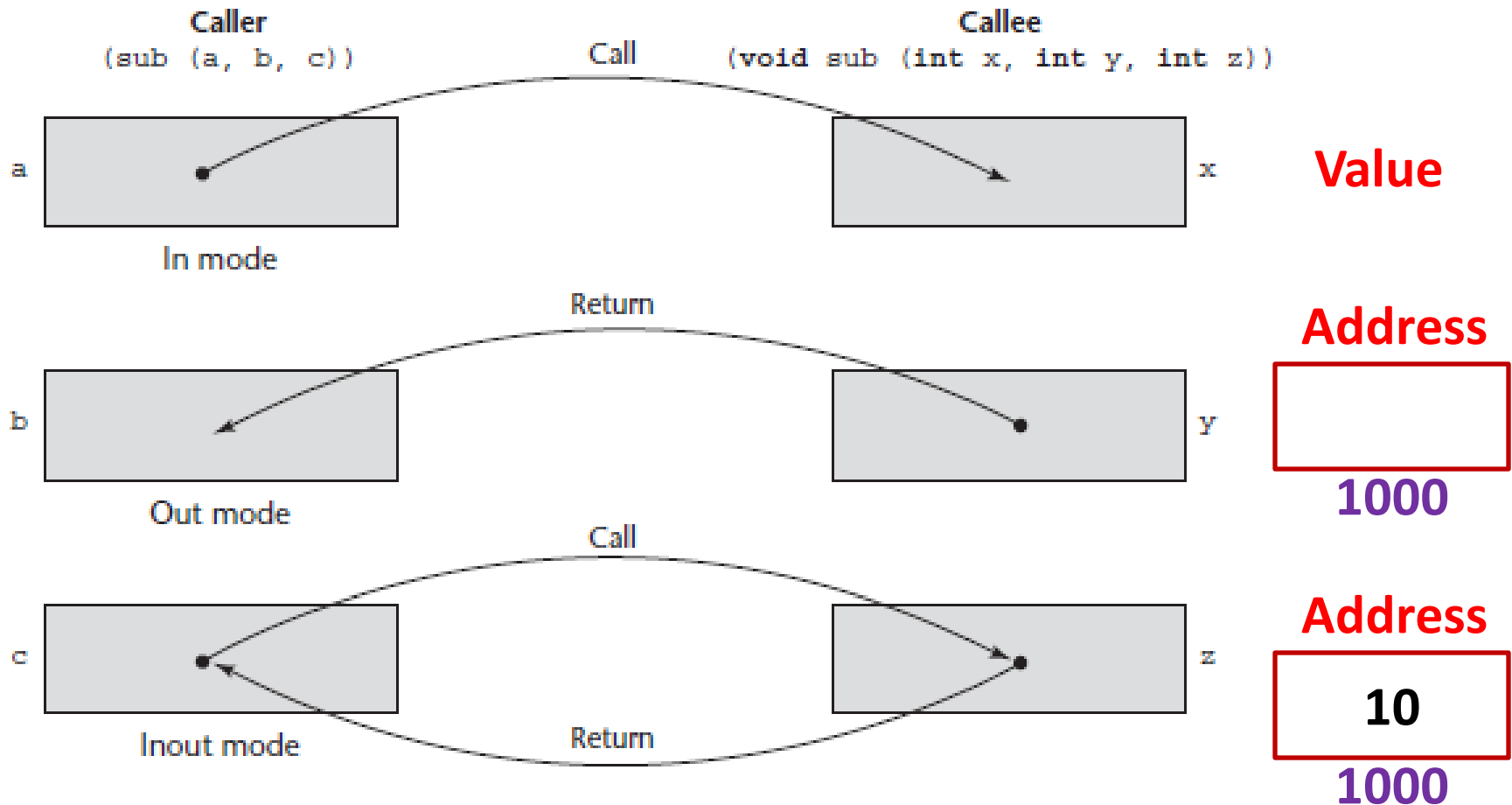


- Formal parameters are characterized by one of three distinct semantics models
 - Can receive data from the corresponding actual parameter (*in mode*)
 - Can transmit data to the actual parameter (*out mode*)
 - Can do both (*inout mode*)
- Eg: Two arrays of int values as parameters -> list1 and list2
 - Subprogram must add list1 to list2 and return the result as a revised version of list2
 - list1 (*in mode*); list2 (*inout mode*); Third array (*out mode*)
 - Third array should be *out mode*, because there is no initial value for this array and its computed value must be returned to the caller

Semantic Models of Parameter passing

- Two conceptual models of how data transfers take place in parameter transmission
 - An actual value is copied (to the caller, to the called, or both ways)
in mode *out mode*
inout mode
 - An access path is transmitted
- Most commonly, the access path is a simple pointer or reference

Implementation Models of Parameter Passing

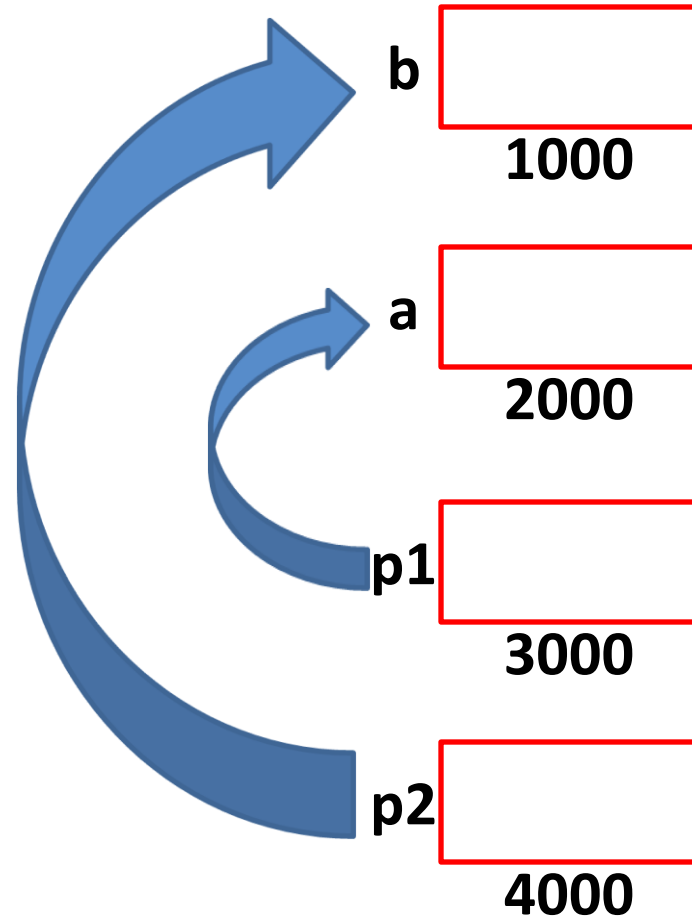


Pass-by-Value

- 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
- Normally implemented by copy
- 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
- **Adv:** For scalars it is fast, in both linkage cost and access time
- **Disadv:**
 - If copies are used, then an 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
 - Actual parameter must be copied to the storage area for the corresponding formal parameter
- Storage and the copy operations can be costly if the parameter is large, such as an array with many elements

Pass-by-Value

```
void sub(int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(p1, p2);
```

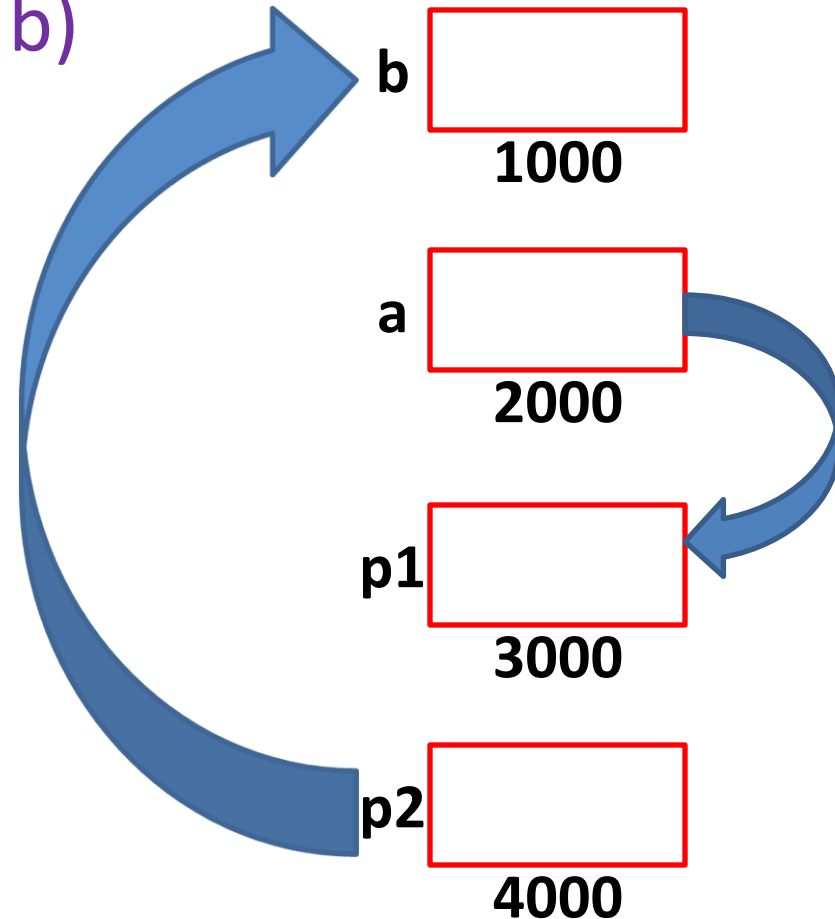


Pass-by-Result

- 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
- 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

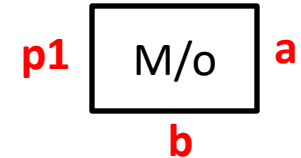
Pass-by-Result

```
void sub(out int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(out p1, p2);
```



Pass-by-Result

- Additional Problem: Actual Parameter Collision



`sub(p1, p1)`

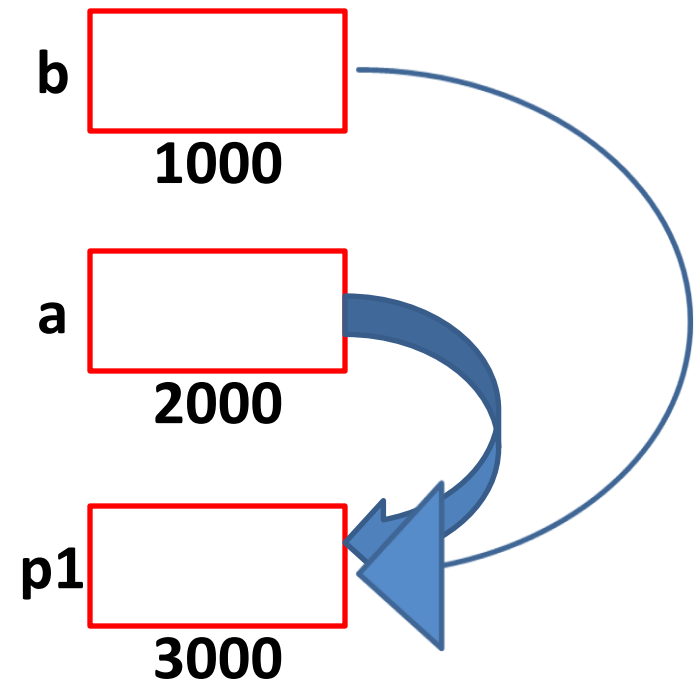
```
void sub(out int a, out int b)
{
    a = 17;
    b = 35;
}
int p1;
f.sub(out p1, out p1);
```

- If, at the end of the execution of sub, the formal parameter a is assigned to its corresponding actual parameter first, then the value of the actual parameter p1 in the caller will be 35 (***This happens when the parameters are evaluated from L -> R***)
- If b is assigned first, then the value of the actual parameter p1 in the caller will be 17 (***This happens when the parameters are evaluated from R -> L***)

- Order can be implementation dependent for some languages, different implementations can produce different results

Pass-by-Result

```
void sub(out int a, out int b)
{
    a = 17;
    b = 35;
}
int p1;
f.sub(out p1, out p1);
```



Pass-by-Result

- **Another Problem:** Implementor may be able to choose between two different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return

```
void DoIt(out int x, int index){  
    x = 17;  
    index = 42;  
}  
...  
sub = 21;  
f.DoIt(list[sub], sub);
```

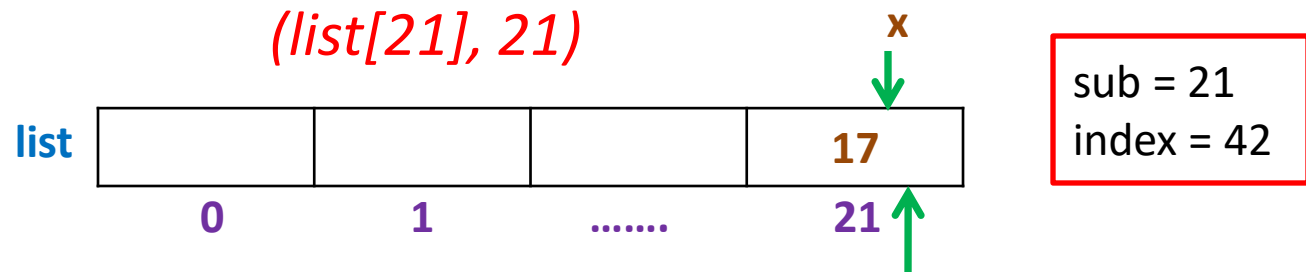
- If the address is computed on entry to the method, the value 17 will be returned to list[21]
- If computed just before return, 17 will be returned to list[42]

- Address of list[sub] changes between the beginning and end of the method
- Implementor must choose the time to bind this parameter to an address - at the time of the call or at the time of the return

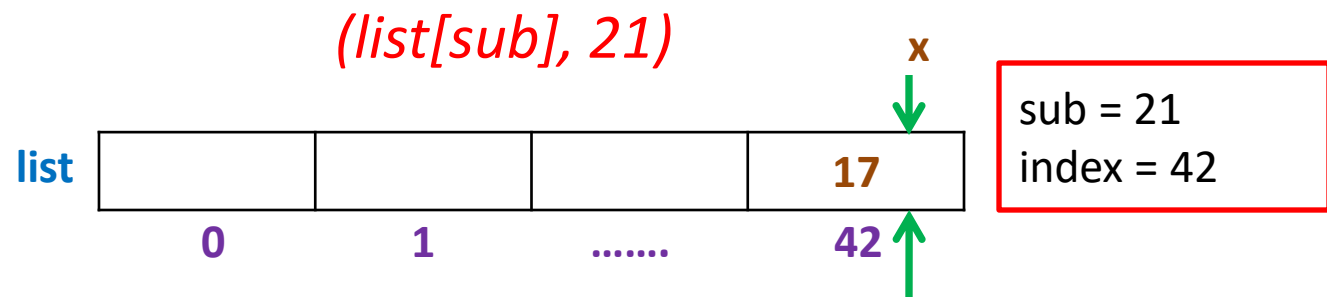
Pass-by-Result

```
void DoIt(out int x, int index) {
    x = 17;
    index = 42;
}
...
sub = 21;
f.DoIt(list[sub], sub);
```

- Address is computed on entry to the method:



- Address is computed just before return:

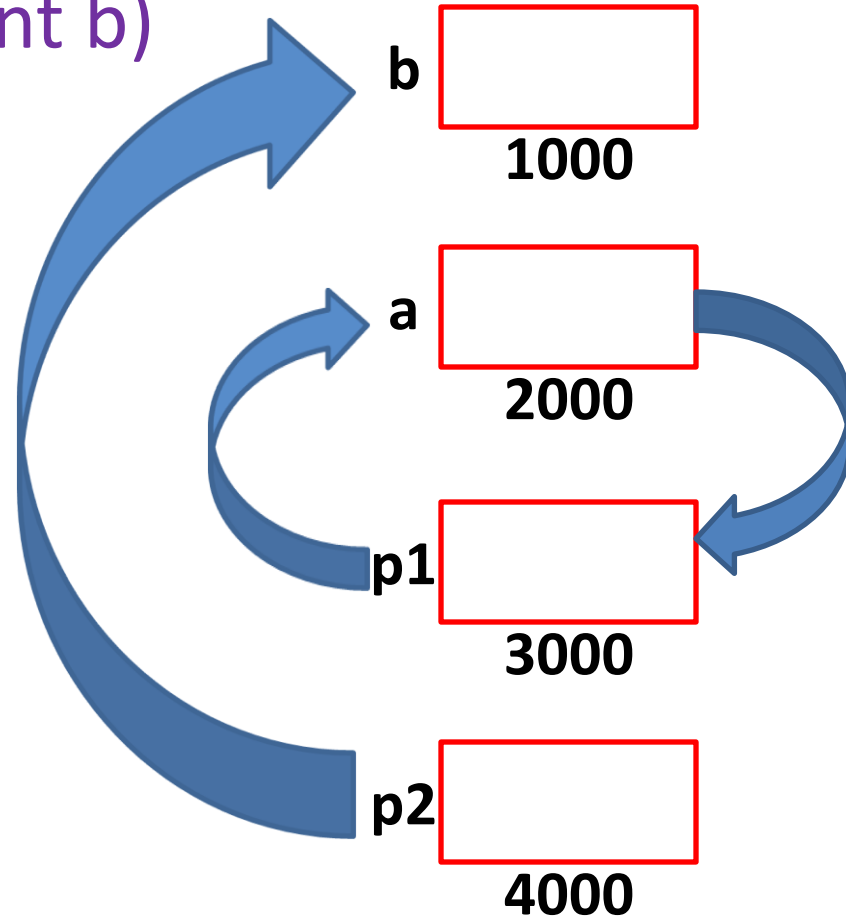


Pass-by-Result

- If the address is computed on entry to the method, the value 17 will be returned to list[21]
- If computed just before return, 17 will be returned to list[42]
- Makes programs unportable between an implementation that chooses to evaluate the addresses for out-mode parameters at the beginning of a subprogram and one that chooses to do that evaluation at the end

Pass-by-Value-Result

```
void sub(inout int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(inout p1, p2);
```



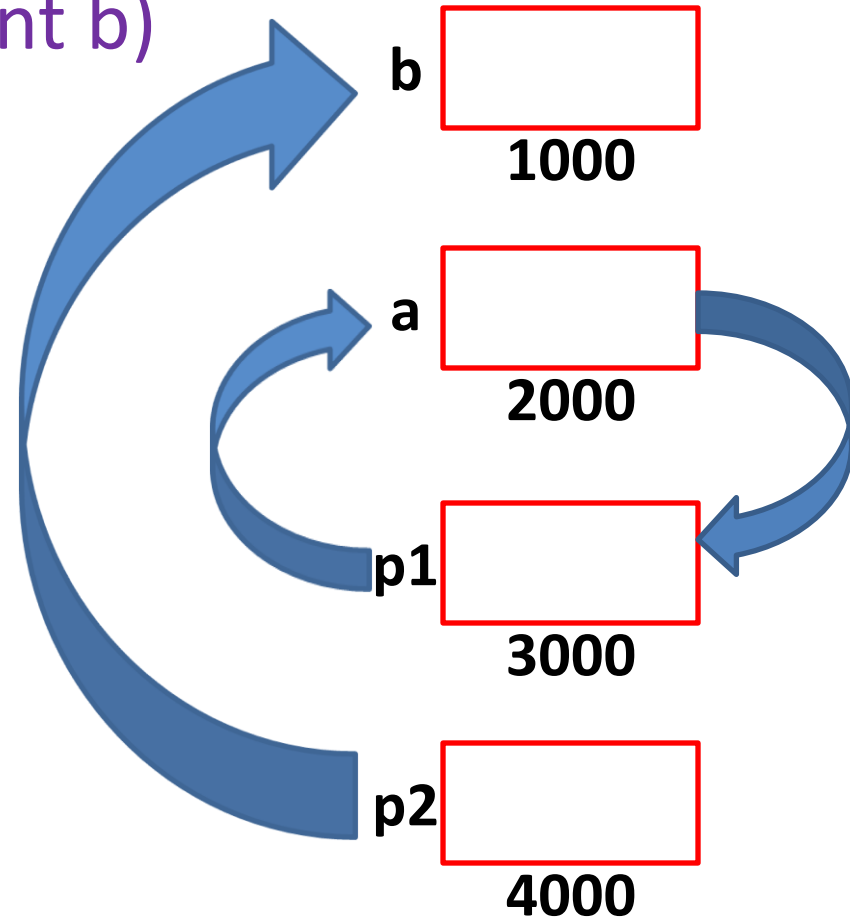
Pass-by-Value-Result

- Implementation model for inout-mode parameters in which actual values are copied
- Combination of both pass-by-value and pass-by-result
- The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable
- Pass-by-value-result 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

```
void sub(inout int a, int b)
{
    a = 17;
    b = 35;
}

int p1 = 5;
int p2 = 10;
f.sub(inout p1, p2);
```



Pass-by-Value Result

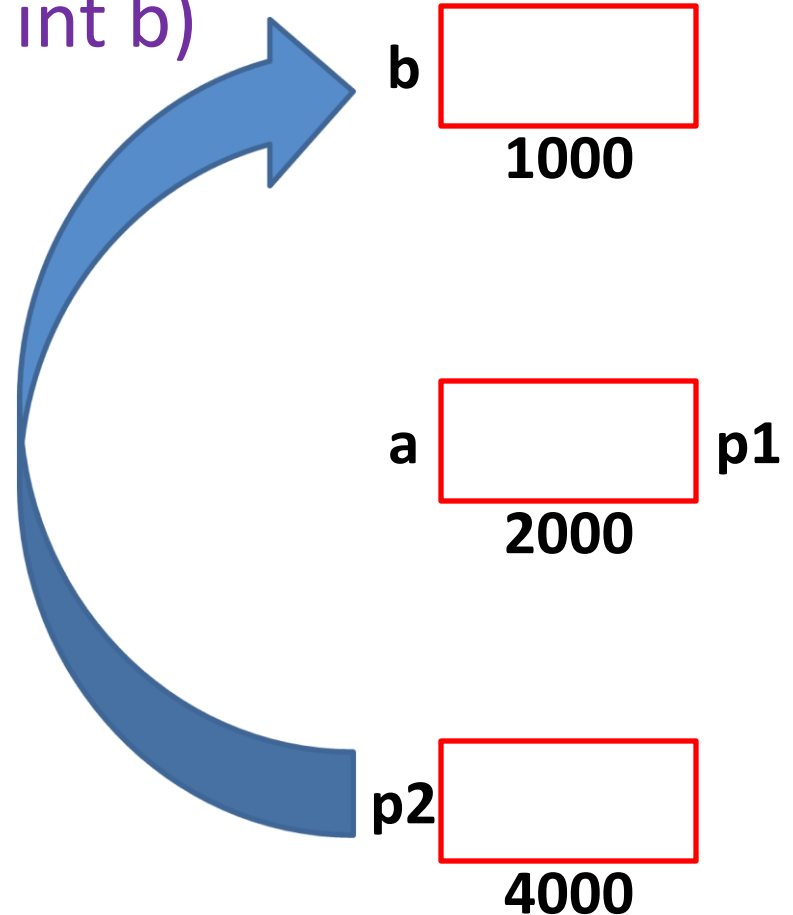
- **Disadv:**
 - Requires multiple storage for parameters and time for copying values
 - Shares with pass-by-result the problems associated with the order in which actual parameters are assigned
- **Adv:** Relative to pass-by-reference

Pass-by-Reference

- Second implementation model for inout-mode parameters
- 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
- **Adv:** Duplicate space is not required, nor is any copying required

Pass-by-Reference

```
void sub(inout int *a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(inout &p1, p2);
```



Pass-by-Reference

- Disadv:
 - Access time
 - Not possible -> One-way communication to the called subprogram is required
 - Aliases can be created -> Harmful to Readability and Reliability
 - Collisions can occur

Pass-by-Reference

- Collisions can occur between actual parameters

void fun(int &first, int &second)

fun(total, total) -> *first* and *second* are aliases

fun(list[i], list[j]) -> If $i == j$, then *first* and *second* are aliases

fun1(list[i], list) -> Aliasing in fun1

- Collisions can occur between formal parameters and nonlocal variables that are visible

- Inside sub -> *param* and *global* are aliases

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

Pass-by-Name

- 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
- 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

Pass-by-Name

- 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

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))  
x = min(a, b);           → x = ((a) < (b) ? (a) : (b));  
y = min(1, 2);           → y = ((1) < (2) ? (1) : (2));  
z = min(a + 28, *p);     → z = ((a + 28) < (*p) ? (a + 28) : (*p));
```


Pass-by-Name

```
void sub(name int a, int b)
{
    b = a + 17;    // Replace a by p1 or 5
}

int p1 = 5;
int p2 = 10;
f.sub(name p1, p2);
```

Passing Values

```
void sub(int a, int b)
{
    a = 17;
    b = 35;
}
int p1 = 5;
int p2 = 10;
f.sub(p1, p2);
```

1. Pass-by-Value (*in*)
2. Pass-by-Result (*out*)
3. Pass-by-Value-Result (*inout*)
4. Pass-by-Reference (*inout + [&, *]*)
5. Pass-by-Name (*name*)

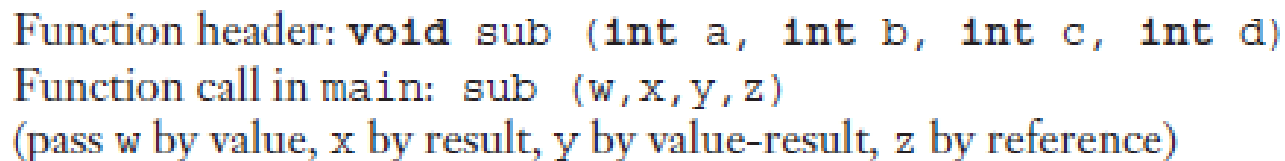
Implementing Parameter Passing

Methods

- Contemporary languages -> Parameter communication takes place through the run-time stack
- Run-time stack is initialized and maintained by the run-time system, which manages the execution of programs
- Runtime stack is used extensively for subprogram control linkage and parameter passing
- Following discussion assume that the stack is used for all parameter transmission

Pass-by-Value

- Parameters have their values copied into stack locations
- Stack locations then serve as storage for the corresponding formal parameters



Implementing Parameter Passing Methods



Pass-by-Result

- Parameters are implemented as the opposite of pass-by-value
- 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

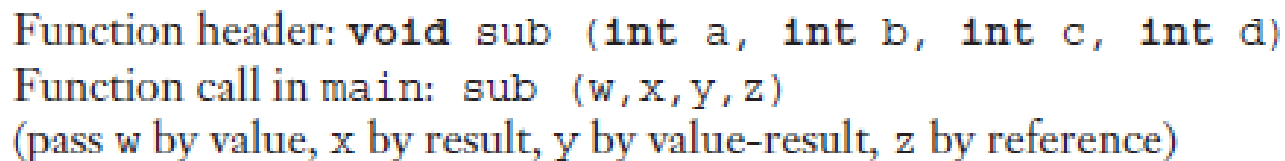
Pass-by-Value-Result

- Parameters can be implemented directly from their semantics as a combination of pass-by-value and pass-by-result
- Stack location for such a parameter is initialized by the call and is then used like a local variable in the called subprogram

Implementing Parameter Passing Methods

Pass-by-Reference

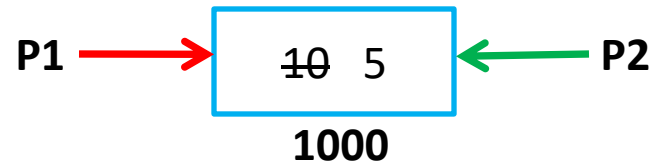
- Simplest to implement
- Regardless of the type of the actual parameter, only its address must be placed in the stack
- In the case of literals, the address of the literal is put in the stack
- In the case of an expression, the compiler must build code to evaluate the expression just before the transfer of control to the called subprogram
- Address of the memory cell in which the code places the result of its evaluation is then put in the stack
- Compiler must be sure to prevent the called subprogram from changing parameters that are literals or expressions
- Access to the formal parameters in the called subprogram is by indirect addressing from the stack location of the address



Implementing Parameter Passing Methods

- Error can occur with pass-by-reference and pass-by-value-result parameters, if care is not taken in their implementation
- Program contains two references to constant 10, the first as an actual parameter in a call to a subprogram
- Subprogram mistakenly changes the formal parameter that corresponds to the value 10 to the value 5
- Compiler for this program may have built a single location for the value 10 during compilation, as compilers often do, and uses that location for all references to the constant 10 in the program
- After the return from the subprogram, all subsequent occurrences of 10 will actually be references to the value 5
- Creates a programming problem that is very difficult to diagnose

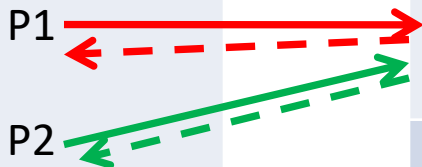
Implementing Parameter Passing Methods



Main

Stack

Subprogram



1000
(Address of Constant 10)

Reference to Constant 10
(Changes the value to 5)

Parameter Passing Methods of some Common Languages



- C uses pass-by-value
- Pass-by-reference (inout mode) semantics is achieved by using pointers as parameter
 - Value of the pointer is made available to the called function and nothing is copied back
 - However, because what was passed is an access path to the data of the caller, the called function can change the caller's data
- Formal parameters can be typed as pointers to constants
 - Corresponding actual parameters need not be constants, for in such cases they are coerced to constants
- Allows pointer parameters to provide the efficiency of pass-by-reference with the one-way semantics of pass-by-value
- Write protection of those parameters in the called function is implicitly specified

Miscellaneous

```
void sub(const int *a, int b)
```

```
{
```

```
    *a = 17;    // Throws error
```

```
    b = *a + 35;
```

```
}
```

```
int p1 = 5;
```

```
int p2 = 10;
```

```
f.sub(&p1, p2);
```

Error: Assignment of Read-only Location

Parameter Passing Methods of some Common Languages

- C++ includes a special pointer type, called a *reference type* which is often used for parameters
- 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

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

where p1 is pass-by-reference but cannot be changed in the function fun, p2 is pass-by-value, and p3 is pass-by-reference

- Neither p1 nor p3 need be explicitly dereferenced in fun
- Constant parameters and in-mode parameters are not exactly alike
- Constant parameters clearly implement in mode

Parameter Passing Methods of some Common Languages

- In a pure object-oriented language, the process of changing the value of a variable with an assignment statement, $x = x + 1$, 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 changes x to reference the new object
- So, when a reference to a scalar object is passed to a subprogram, the object being referenced cannot be changed in place
- Because the reference is passed by value, even though the formal parameter is changed in the subprogram, that change has no effect on the actual parameter in the caller
- Now, suppose a reference to an array is passed as a parameter
- If the corresponding formal parameter is assigned a new array object, there is no effect on the caller
- However, if the formal parameter is used to assign a value to an element of the array, as in `list[3] = 47`, the actual parameter is affected
- So, changing the reference of the formal parameter has no effect on the caller, but changing an element of the array that is passed as a parameter does

Type Checking Parameters

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

`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
- sub1 cannot produce a correct result given an integer actual parameter value when it expects a floating-point value

Type Checking Parameters

- C and C++ require some special discussion in the matter of parameter type checking
- In the original C, neither the number of parameters nor their types were checked
- In C89, the formal parameters of functions can be defined in two ways
 - Names of the parameters are listed in parentheses and the type declarations for them follow
 - Prototype method, in which the formal parameter types are included in the list

```
double sin(x)  
double x;  
{ ... }
```

```
double value;  
int count;  
...  
value = sin(count);
```

```
double sin(double x)  
{ ... }  
value = sin(count);
```

Type Checking Parameters



```
int printf(const char* format_string, ...);
```

- A call to printf must include at least one parameter, a pointer to a literal character string

```
printf();    //Error: Too few arguments
```

```
printf("")   //Warning: Zero-Length Format String
```

```
printf(" The String");
```

- Beyond that, anything (including nothing) is legal
- The way printf determines whether there are additional parameters is by the presence of format codes in the string parameter
- For example, the format code for integer output is %d
- This appears as part of the string, as in the following:

```
printf("The sum is %d\n", sum);
```

- The % tells the printf function that there is one more parameter

Multidimensional Arrays as Parameters

- Multidimensional arrays in C are really arrays of arrays, and they are stored in row major order

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

// Size of each element is 1

- Mapping function needs the number of columns but not the number of rows

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

Multidimensional Arrays as Parameters



- Pbm: Does not allow a programmer to write a function that can accept matrixes with different numbers of columns
- Matrix can be passed as a pointer, and the actual dimensions of the matrix also can be passed as parameters
- Then, the function can evaluate the user-written storage-mapping function using pointer arithmetic each time an element of the matrix must be referenced

```
void fun(float *mat_ptr,  
        int num_rows,  
        int num_cols);
```

```
*(mat_ptr + (row * num_cols) + col) = x;
```

Macro:

```
#define mat_ptr(r,c)  (*mat_ptr + ((r) *  
                          (num_cols) + (c)))  
  
mat_ptr(row,col) = x;
```

Design Considerations

- Two important considerations
 - Choosing parameter-passing methods: efficiency
 - Whether one-way or two-way data transfer is needed
- Contemporary software-engineering principles dictate that access by subprogram code to data outside the subprogram should be minimized
- in-mode parameters should be used whenever no data are to be returned through parameters to the caller
- Out-mode parameters should be used when no data are transferred to the called subprogram but the subprogram must transmit data back to the caller
- inout-mode parameters should be used only when data must move in both directions between the caller and the called subprogram

Design Considerations

- There is a practical consideration that is in conflict with this principle
- Sometimes it is justifiable to pass access paths for one-way parameter transmission
 - **Eg:** When a large array is to be passed to a subprogram that does not modify it, a one-way method may be preferred
 - However, pass-by-value would require that the entire array be moved to a local storage area of the subprogram
 - This would be costly in both time and space
- Because of this, large arrays are often passed by reference
- C++ constant reference parameters offer another solution
- Another alternative approach would be to allow the user to choose between the **methods**
- The choice of a parameter-passing method for functions is related to another design issue: functional side effects

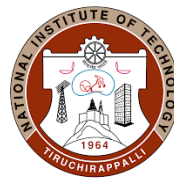
Static vs Dynamic Scoping

```
1  x <- 1
2  f <- function(a) x + a
3  g <- function() {
4    x <- 2
5    f(0)
6  }
7  g() # what does this return?
```

- Under *lexical scoping* (also known as *static scoping*), the scope of a variable is determined by the lexical (*i.e.*, textual) structure of a program
- Under *dynamic scoping*, a variable is bound to the most recent value assigned to that variable

- In Static Scoping, the final result will be 1
- In Dynamic Scoping, the final result will be 2

Static Scoping vs Dynamic Scoping



- Lexical scoping (sometimes known as *static scoping*) -> Sets the *scope* (range of functionality) of a variable so that it may only be *called* (referenced) from within the block of code in which it is defined
 - Scope is determined when the code is compiled
 - A variable declared in this fashion is sometimes called a *private* variable
- The opposite approach is known as *dynamic scoping* -> Creates variables that can be called from outside the block of code in which they are defined
 - A variable declared in this fashion is sometimes called a *public* variable

Private vs Public Variables

```
#include <iostream>
using namespace std;

// class definition
class Circle {
public: ←
    double radius;

    double compute_area()
    {
        return 3.14 * radius * radius;
    }
};

// main function
int main()
{
    Circle obj;

    // accessing public data member outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

O/P:

```
Radius is: 5.5
Area is: 94.985
```

```
#include <iostream>
using namespace std;

// class definition
class Circle {
private: ←
    double radius;

    double compute_area()
    {
        return 3.14 * radius * radius;
    }
};

// main function
int main()
{
    Circle obj;

    // accessing public data member outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

O/P: Error (Trying to access a variable and function which does not fall inside the scope)

Parameters that are Subprograms

- Subprogram names will be sent as parameters to other subprograms
- If only the transmission of the subprogram code was necessary, it could be done by passing a single pointer
- Two complications arise
 - 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
 - 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

Miscellaneous

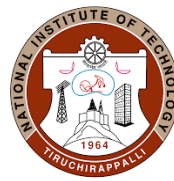
```
int main(void)
{
    char s1[80], s2[80];
    int (*p) (const char *, const char *);
    p = strcmp;
    printf("Enter two strings\n");
    gets(s1);
    gets(s2);
    check(s1, s2, p);
    return 0;
}
```

```
void check(char *a, char *b, int
    (*cmp)(const char *, const
    char *))
{
    printf("Testing for Equality\n");
    if(!(*cmp(a, b))
        printf("Equal");
    else
        printf("Not Equal");
}
```




Parameters that are Subprograms

- Languages that allow nested subprograms -> What referencing environment for executing the passed subprogram should be used
 - Environment of the call statement that enacts the passed subprogram -> Shallow Binding
 - Environment of the definition of the passed subprogram -> Deep Binding
 - Environment of the call statement that passed the subprogram as an actual parameter -> Ad hoc Binding

Parameters that are Subprograms



- Consider the execution of sub2 when it is called in sub4
 - Shallow Binding -> Referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4 (*Output is 4*)
 - Deep Binding -> Referencing environment of sub2's execution is that of sub1, so the reference to x in sub2 is bound to the local x in sub1 (*Output is 1*)
 - Ad hoc Binding, the binding is to the local x in sub3 (*Output is 3*)

```
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();  
};
```

- Environment of the call statement that enacts the passed subprogram -> Shallow Binding
- Environment of the definition of the passed subprogram -> Deep Binding
- Environment of the call statement that passed the subprogram as an actual parameter -> Ad hoc Binding

Parameters that are Subprograms

```
function sub1() {  
  var x;  
  function sub2() {  
    alert(x); // Creates a dialog box with the value of x  
  };  
  sub4(sub2); // Assume this statement is  
               inserted at this location  
  function sub3() {
```

```
    var x;  
    x = 3;  
      
  };  
  function sub4(subx) {  
    var x;  
    x = 4;  
    subx();  
  };  
  x = 1;  
  sub3();  
};
```

Note: Here, both Deep Binding and Ad hoc Binding will output value 1. Because, sub2 is declared inside sub1. And, sub2 is passed as argument to sub4 by sub1 itself.

- Environment of the call statement that enacts the passed subprogram -> Shallow Binding
- Environment of the definition of the passed subprogram -> Deep Binding
- Environment of the call statement that passed the subprogram as an actual parameter -> Ad hoc Binding

Parameters that are Subprograms



- In some cases, the subprogram that declares a subprogram also passes that subprogram as a parameter
- In those cases, deep binding and ad hoc binding are the same
- Ad hoc binding 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 subprogram
- Shallow binding is not appropriate for static-scoped languages with nested subprograms
- Eg: Suppose the procedure Sender passes the procedure Sent as a parameter to the procedure Receiver
- Problem is that Receiver may not be in the static environment of Sent, thereby making it very unnatural for Sent to have access to Receiver's variables
- On the other hand, it is perfectly normal in such a language for any subprogram, including one sent as a parameter, to have its referencing environment determined by the lexical position of its definition
- It is therefore more logical for these languages to use deep binding
- Some dynamic-scoped languages use shallow binding

Parameters that are Subprograms


Assume:


sub3 -> Sender


sub2 -> Sent

sub4 -> Receiver

```
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();  
};
```

- Shallow binding is not appropriate for static-scoped languages with nested subprograms
- Eg: Suppose the procedure Sender (sub3) passes the procedure Sent (sub2) as a parameter to the procedure Receiver (sub4)
- Problem is that Receiver (sub4) may not be in the static environment of Sent (sub2), thereby making it very unnatural for Sent (sub2) to have access to Receiver's (sub4's) variables
- On the other hand, it is perfectly normal in such a language for any subprogram, including one sent as a parameter, to have its referencing environment determined by the lexical position of its definition
- It is therefore more logical for these languages to use deep binding
- Some dynamic-scoped languages use shallow binding

Calling Subprograms Indirectly



- Occurs when the specific subprogram to be called is not known until run time
- 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
- **Applications:** Event handling in graphical user interfaces; Callbacks
-> A subprogram is called and instructed to notify the caller when the called subprogram has completed its work
- C and C++ allow a program to define a pointer to a function, through which the function can be called
- In C++, pointers to functions are typed according to the return type and parameter types of the function, so that such a pointer can point only at functions with one particular protocol

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

- In C and C++, a function name without following parentheses, like an array name without following brackets, is the address of the function (or array)

Calling Subprograms Indirectly



- Both of the following are legal ways of giving an initial value or assigning a value to a 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
pfun2 = myfun2; // Assigning a function's address to a
               // pointer
```

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

- The first of these explicitly dereferences the pointer pfun2, which is legal, but unnecessary
- The function pointers of C and C++ can be sent as parameters and returned from functions, although functions cannot be used directly in either of those roles

Miscellaneous

```
int main(void)
{
    char s1[80], s2[80];
    int (*p) (const char *, const char *);
    p = strcmp;
    printf("Enter two strings\n");
    gets(s1);
    gets(s2);
    check(s1, s2, p);
    return 0;
}
```

```
void check(char *a, char *b, int
    (*cmp)(const char *, const
    char *))
{
    printf("Testing for Equality\n");
    if(!(*cmp(a, b))
        printf("Equal");
    else
        printf("Not Equal");
}
```

Overloaded Subprograms

- Overloaded operator is one that has multiple meanings
- Meaning of a particular instance of an overloaded operator is determined by the types of its operands. **Eg:** $a * b$
- Overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment

int sum(int a, int b);

int sum(int a, int b, int c);

float sum(float a, float b);

- Every version of an overloaded subprogram must have a unique protocol
 - 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
- 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)
- Not necessary that overloaded subprograms usually implement the same process

Overloaded Subprograms

- Because C++, Java, and C# allow mixed-mode expressions, the return type is irrelevant to disambiguation of overloaded functions (or methods)
- The context of the call does not allow the determination of the return type
- **Eg:** If a C++ program has two functions named fun and both take an int parameter but one returns an int and one returns a float, the program would not compile
 - Compiler could not determine which version of fun should be used

int sum(int a, int b);

float sum(int a, int b); // Not valid

- Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls
- Call is ambiguous and will cause a compilation error

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

fun();

Design Issues for Functions

- Are side effects allowed? Eg: `int Sum = a + fun(a)`
- What types of values can be returned?
- How many values can be returned?

Functional Side Effects

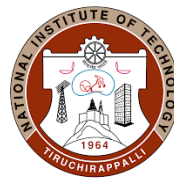
- Because of the problems of side effects of functions that are called in expressions, parameters to functions should always be in-mode parameters (Ada)
 - Effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals
- Most other imperative 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

Design Issues for Functions

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
- 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
 - Same is true for the functions in most functional languages

Design Issues for Functions



Number of Returned Values

- In most languages, only a single value can be returned from a function
- However, that is not always the case
- 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:

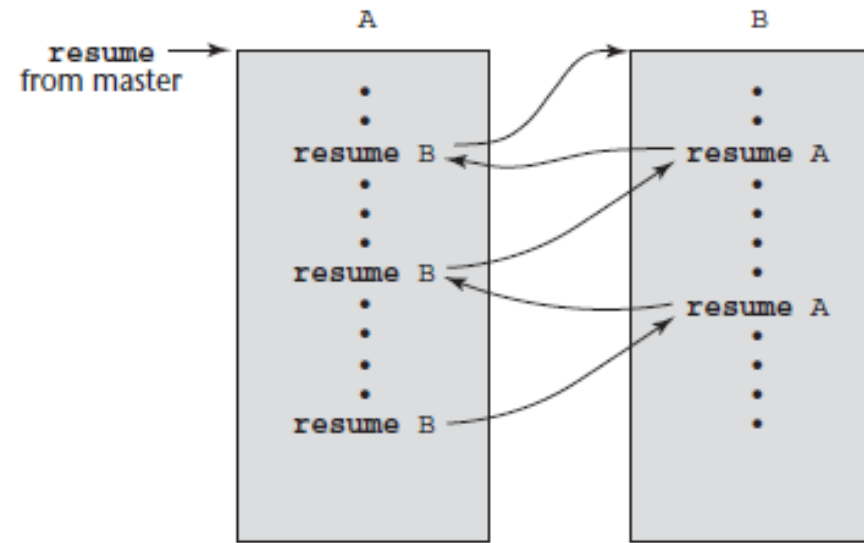
return 3, sum, index

Design Issues for Functions

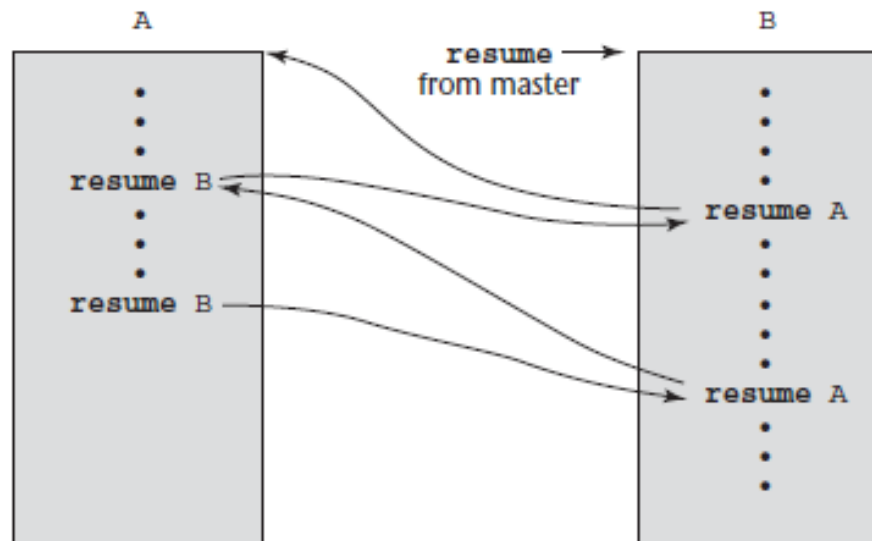
- The form of the statement that calls the function determines the number of values that are received by the caller
 - If the function is called as a procedure, that is, as a statement, all return values are ignored
 - If the function returned three values and all are to be kept by the caller, the function would be called as in the following example:

a, b, c = fun()

Coroutines



(a)



(b)

Coroutines

- 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
- 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

Coroutines



```
sub co1() {  
    ...  
    resume co2();  
    ...  
    resume 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
- This sort of interleaved execution sequence is related to the way multiprogramming operating systems work
- Although there may be only one processor, all of the executing programs in such a system appear to run concurrently while sharing the processor
- In the case of coroutines, this is sometimes called quasi-concurrency

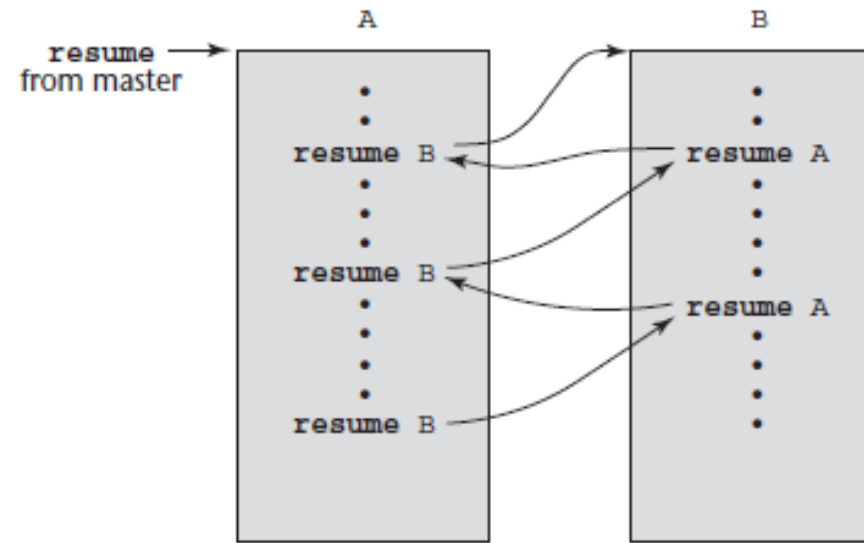
Coroutines

- 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 in fact it can be 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

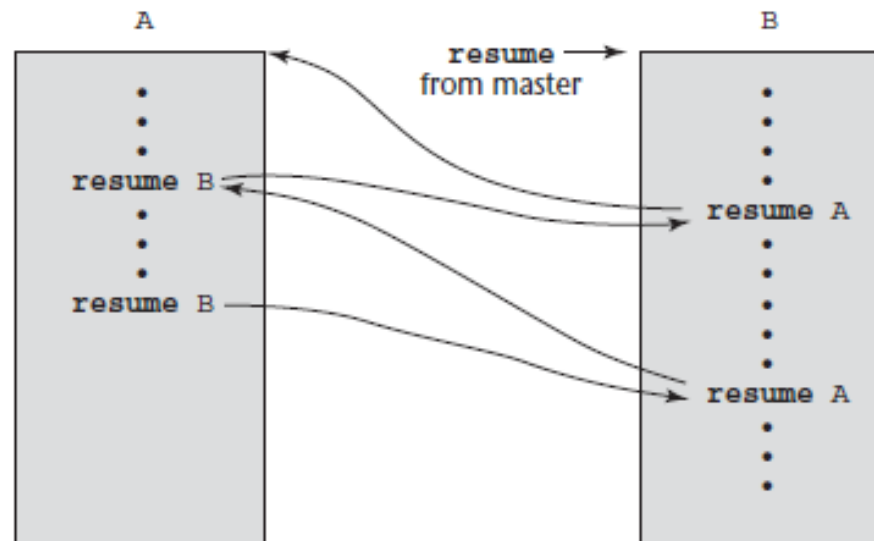
Coroutines

- 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
- Rather than have the patterns, a coroutine often has a loop containing a resume
- Among contemporary languages, only Lua fully supports coroutines

Coroutines

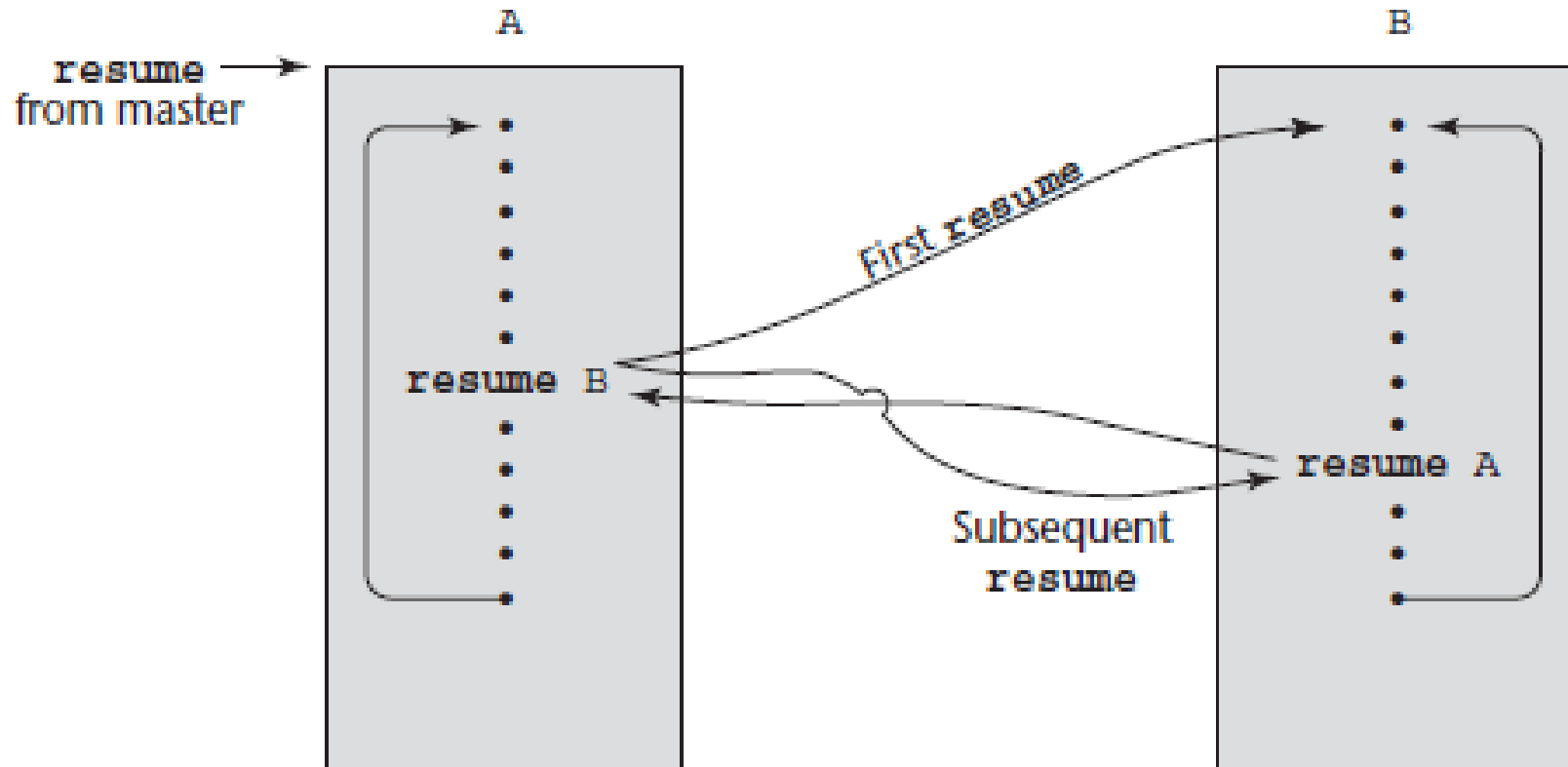


(a)



(b)

Coroutines



Rather than have the patterns, a coroutine often has a loop containing a resume