Module 3

Subroutines and Control Abstraction

- ❑Subroutines and Control Abstraction: -
- ❑Static and Dynamic Links,
- ❑Calling Sequences,
- ❑ Parameter Passing,
- ❑Generic Subroutines and Modules,
- ❑Exception Handling,
- ❑Co-routines

Abstraction Introduction

- ❑ Abstraction is the act of representing essential features without including the background details or explanations.
- ❑ Abstraction: - *abstraction* as a process by which the programmer can **associate a name with a potentially complicated program** fragment, which can then be thought of in terms of its purpose or function, rather than in terms of its implementation.
- ❑ Two types of abstraction
- ✓ *Control abstraction*→purpose of abstraction is to perform a well-defined operation,
- ✓ *data abstraction*→ purpose of the abstraction is to represent some information

subroutines Introduction

- Subroutines are the principal mechanism for control abstraction in most programming languages
- Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design.
- The reuse results in several different kinds of savings, including memory space and coding time.

subroutines Introduction

- A subroutine performs its operation on behalf of a caller , who waits for the subroutine to finish before continuing execution.
- Most subroutines are parameterized: the caller passes arguments that influence the subroutine’s behaviour, or provide it with data on which to operate.
- **General Subroutines Characteristics**

  a. A subroutines has a single entry point.

  b. The caller is suspended during execution of the called subroutines, which implies that there is only one subroutines in execution at any given time.

  c. Control always returns to the caller when the called subroutines' execution terminates

subroutines

- A **subprogram call is an explicit request** that the called subprogram be executed.
- A subprogram is said to be active if, after having been called, it has begun execution but has not yet completed that execution.
- The two fundamental types of the subroutines are:
  - o Procedures-A subroutine that does not return a value
  - o Functions-A subroutine that returns a value
- Most subroutines are parameterized & Arguments in the time of call are also called **actual parameters**. They are mapped to the subroutine's **formal parameters** at the time a call occurs.

subroutines

- Most languages require subroutines to be declared before they are used, though a few (including Fortran, C, and Lisp) do not.
- Declarations allow the compiler to verify that every call to a subroutine is consistent with the declaration;
  for example, that it passes the right number and types of arguments.
- subroutines header is the first line of the definition, serves several definitions:
  - the following syntactic unit is a subroutines definition
    - The header provides a name for the subroutines.
    - May optionally specify a list of parameters

Subroutine header examples

- Fortran

  Subroutine Adder(parameters)

- Ada

  procedure Adder(parameters)

- C

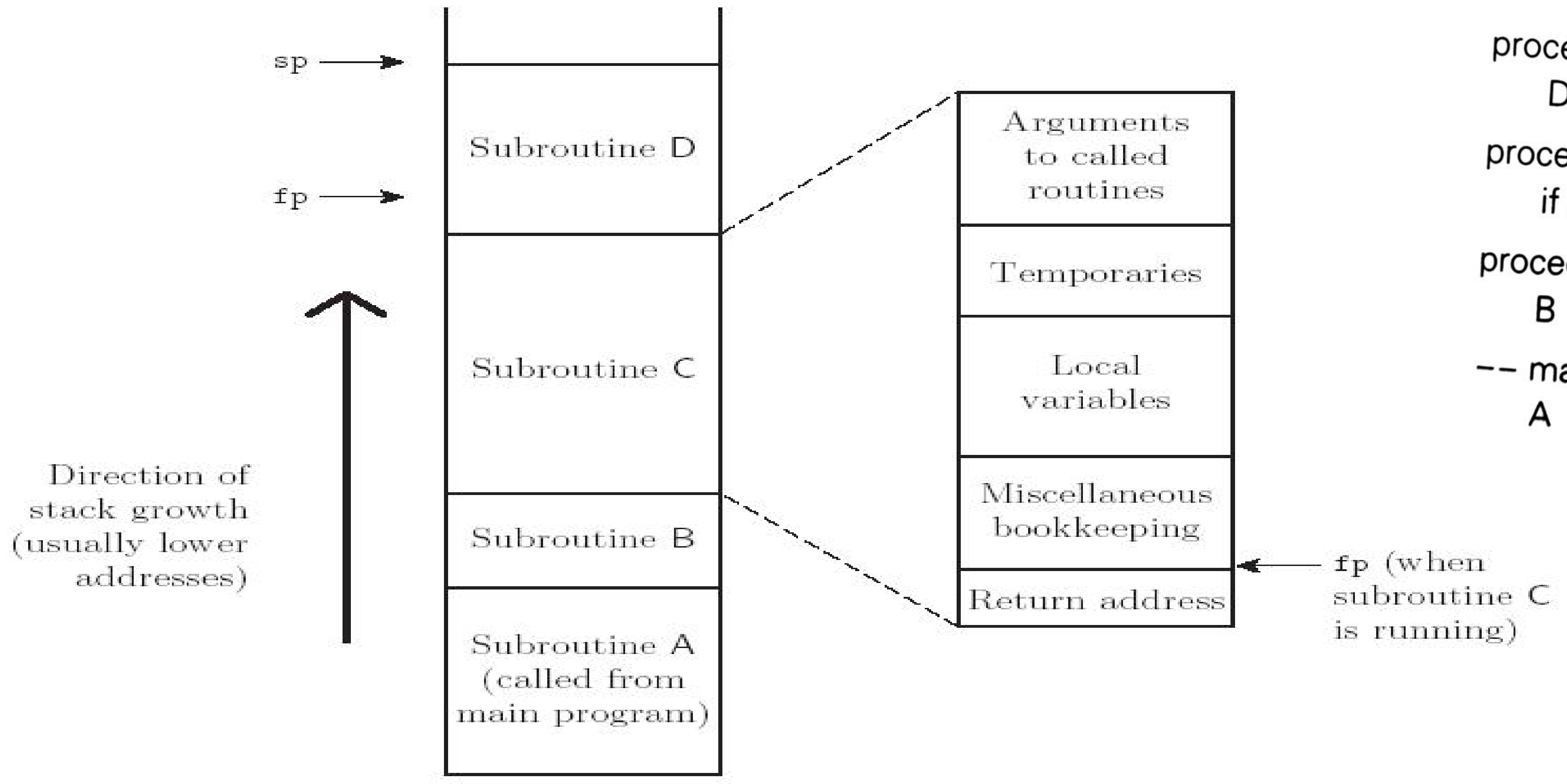
  void Adder(parameters)

- Function declarations are common in C and C++ programs, where they are called prototypes.
- Java and C# do not need declarations of their methods, because there is no requirement that methods be defined before they are called in those languages.

1. Stack layout for storage

- The storage consumed by parameters and local variables can in most languages be allocated on a stack
- The allocation of space on a subroutine call - stack
- Each routine, as it is called, is given a **new stack frame**,or activation record, at the top of the stack.
- This frame may contain arguments and/or return values, bookkeeping information (including the return address and saved registers), local variables, and/or temporaries.
- When a subroutine returns, its frame is popped from the stack.

Stack layout for storage


sp
fp
Direction of stack growth (usually lower addresses)
Subroutine D
Subroutine C
Subroutine B
Subroutine A (called from main program)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)

```
procedure C
    D;  E
procedure B
    if … then B else C
procedure A
    B
-- main program
    A
```

Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (sp) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (fp) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Stack layout for storage

- Stack recap: Each routine, as called, gets a new frame put on the top of the stack
  - Contains arguments, return values, book-keeping info, local variables, and temporaries
- With stack allocation, the subroutine frame for the current frame is on top of the stack
  - sp: top of the stack
  - fp: an address within the top frame
- To access non-local variables, static link (for static scoping) or dynamic link (dynamic scoping) are maintained in the frame

Stack layout for storage

- Static and dynamic links maintained on stack:
  - **Dynamic links** allow one to walk back the frame pointer linearly down the call stack.
  - **Static links** allow one to walk back the frame pointer from a lexical viewpoint.

Static & Dynamic Links

- ❑ Each stack frame contains a reference to the frame of the lexically surrounding subroutine. This reference is called the static link.
- ❑ By analogy, the saved value of the frame pointer, which will be restored on subroutine return, is called the dynamic link.
- ❑ The static and dynamic links may or may not be the same, depending on whether the current routine was called by its lexically surrounding routine, or by some other routine nested in that surrounding routine.

Static & Dynamic Links

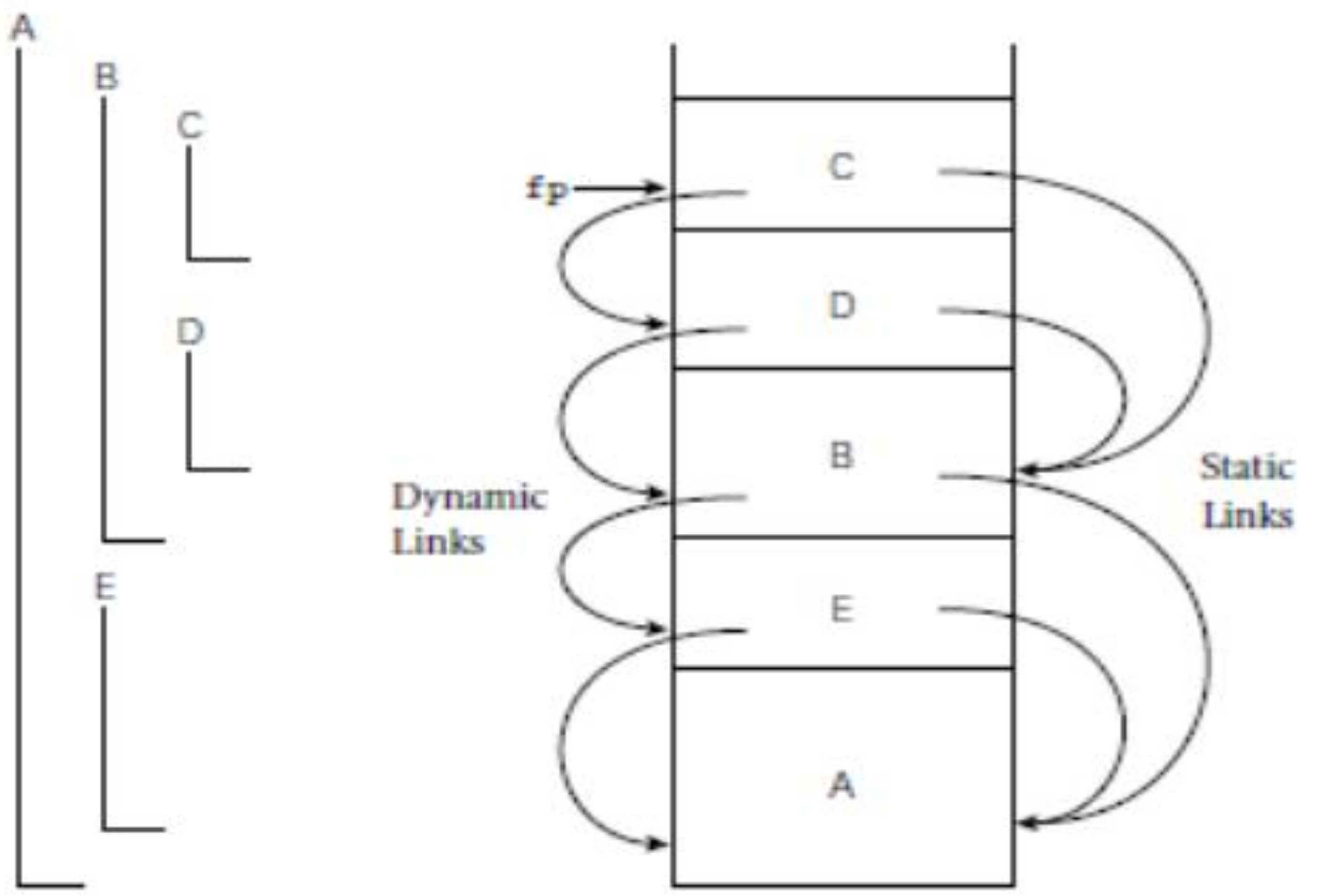
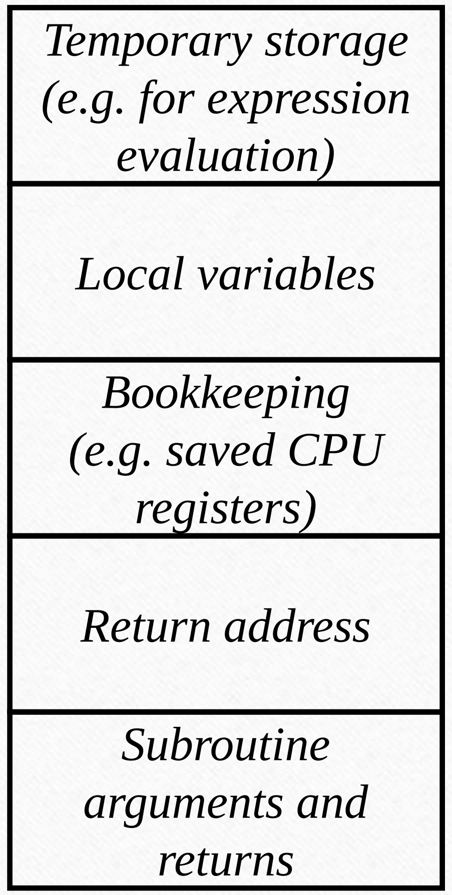


Figure 8.1 Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

Static & Dynamic Links

- If subroutine D is called directly from B, then clearly B's frame will already be on the stack .
- It is not visible in A or E, because it is nested inside of B.
- A moment's thought makes clear that it is only when control enters B (placing B's frame on the stack) that D comes into view.
- It can therefore be called by C, or by any other routine (not shown) that is nested inside C or D, but only because these are also within B.

Subroutine frame (Activation record)


Temporary storage (e.g. for expression evaluation)
Local variables
Bookkeeping (e.g. saved CPU registers)
Return address
Subroutine arguments and returns

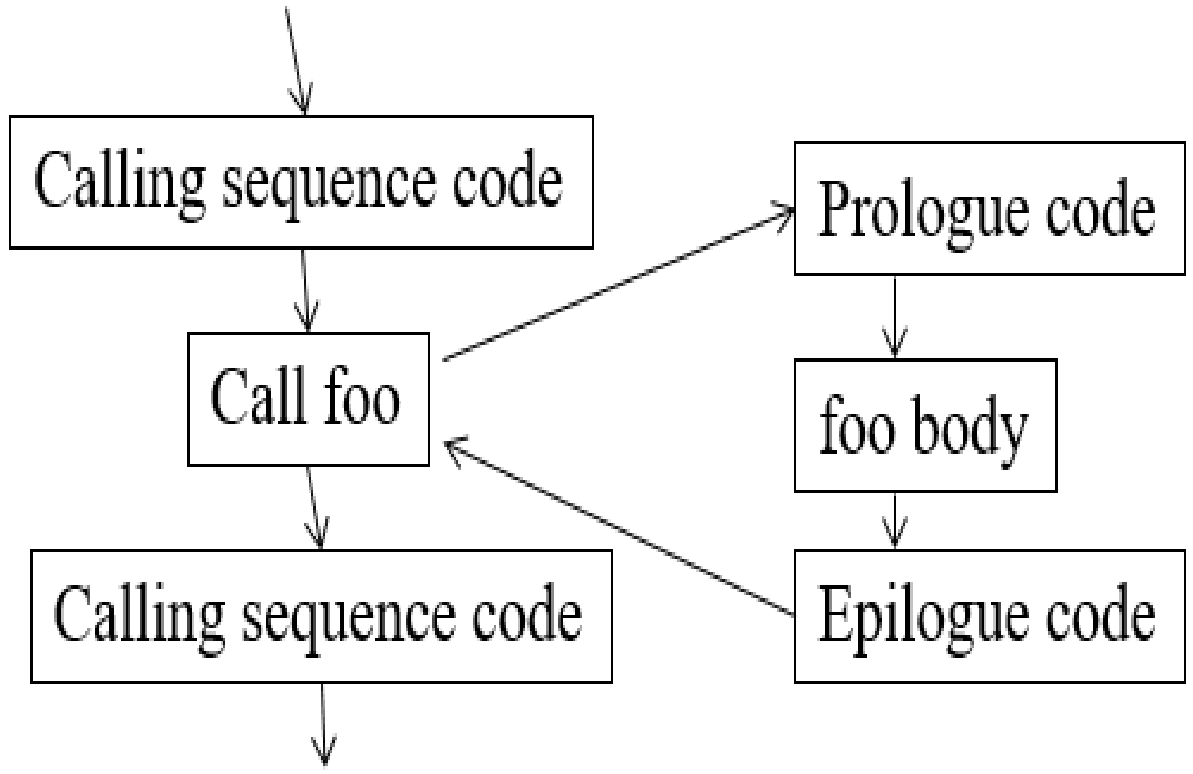
- Activation record (subroutine frame) is used **to store all related information for the execution of a subroutine**
- Before a subroutine is executed, the frame must be set up and some fields in the frame must be initialized
    - Formal arguments must be replaced with actual arguments
- This is done in the calling sequence, a sequence of instructions before and after a subroutine, to set-up the frame.

2.Calling Sequence

- ❑Calling Sequence is maintaining the subroutine call stack
- ❑Calling sequences in general have three components
  - The code executed by the caller immediately before and after a subroutine call.
  - Prologue: code executed at the beginning of the subroutine
  - Epilogue: code executed at the end of the subroutine

Example:

....
foo(100+I, 20+j)
...



2.Calling Sequence

- The term "**calling sequence" is used to refer to the combined operations of the caller, the prologue , and the epilogue.**
- **Maintenance of the subroutine call stack is the responsibility of the calling sequence**
- **calling sequence refers to th**e code executed by the caller immediately before and after a subroutine call —and of the prologue (code executed at the beginning) and epilogue(code executed at the end) of the subroutine itself.

Calling Sequence contd..

- The calling sequence is the code a caller executes to set up a new subroutine
- Responsibilities:
- On the way in: Pass parameters, save return address, change program counter, change stack pointer, save registers that might be overwritten, changing frame pointer to new frame, and initializing code for new objects in the new frame
- On the way out: passing return parameters/values, executing finalization code for local objects, deallocating the stack frame, restoring registers, and restoring the PC

Calling Sequence contd..

- Some of these tasks (e.g., passing parameters) must be performed by the caller, because they differ from call to call.
- Most of the tasks, can be performed either by the caller or the callee.
- space is saved if work is mostly done by callee:
- Anything done in callee appears only once in the target program
- Anything done by caller has to appear before and after every call in the final compiled code

A Typical Calling Sequence

- The stack pointer (sp) points to the first unused location on the stack (or the last used location, depending on the compiler and machine).
- The frame pointer (fp) points to a location near the bottom of the frame.
- Space for all arguments is reserved in the stack, even if the compiler passes some of them in registers (the callee will need a place to save them if it calls a nested routine).

Typical Calling Sequence

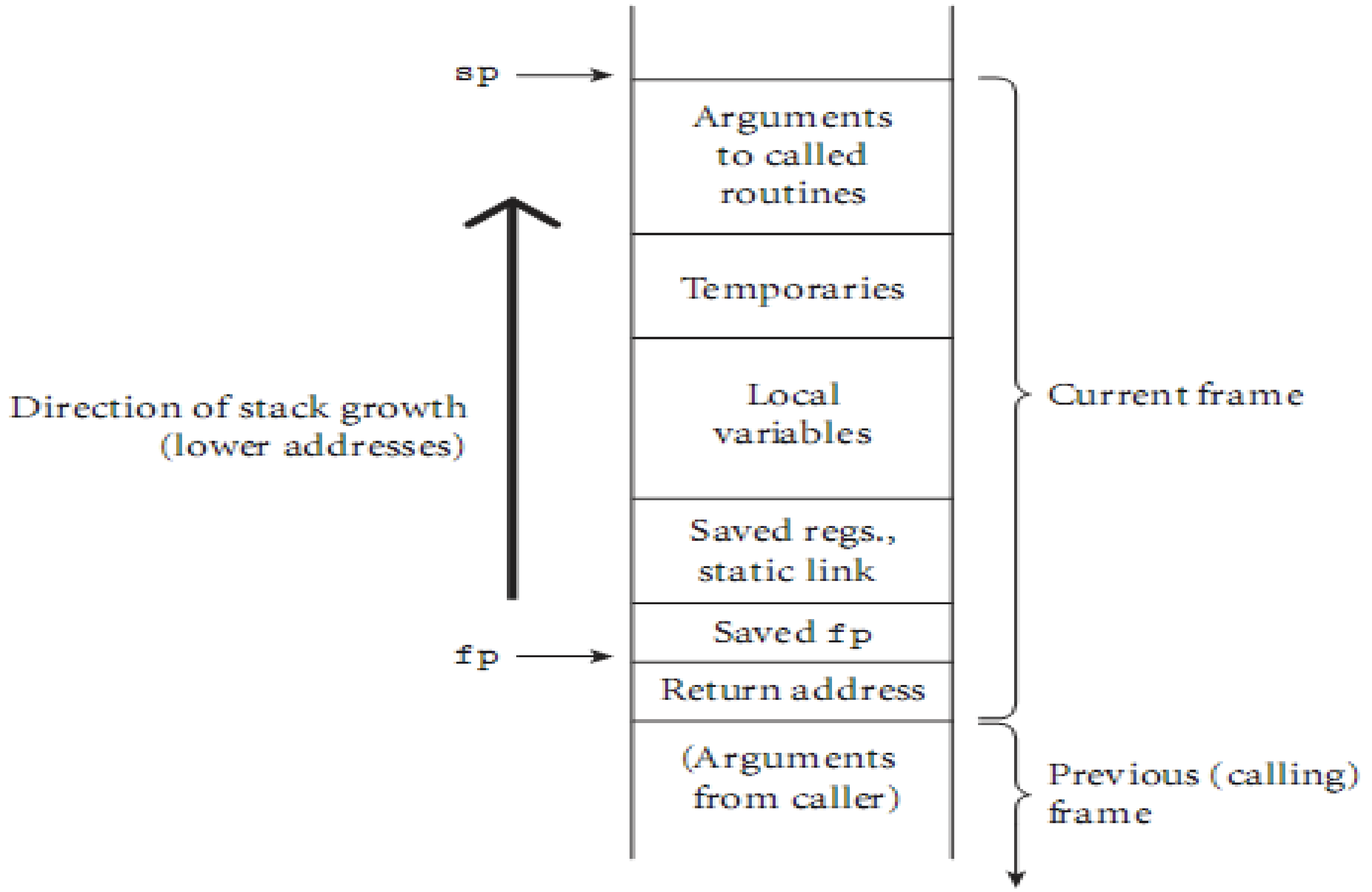


Figure 8.2 **A typical stack frame.** Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

What needs to be done in the calling sequence?

- ❑ Before the subroutine code can be executed

  (caller code before the routing + prologue)

  - ❑ set up the subroutine frame
  - ❑ Compute the parameters and pass the parameters
  - ❑ Saving the return address
  - ❑ Save registers
  - ❑ Changing sp, fp (to add a frame for the subroutine)
  - ❑ Changing pc (start running the subroutine code)
  - ❑ Execute initialization code when needed

What needs to be done in the calling sequence?

- After the subroutine code is executed (caller code after the routine + epilogue) – remove the subroutine frame
  - Passing return result or function value
  - Finalization code for local objects
  - Deallocating the stack frame (restoring fp and sp to their previous value)
  - Restoring saved registers and PC
- Some of the operations must be performed by the caller, others can either be done by the caller or callee.

To maintain this stack layout, the calling sequence might operate as follows.

- The caller

1. saves any caller-saves registers whose values will be needed after the call
2. computes the values of arguments and moves them into the stack or registers
3. computes the static link, and passes it as an extra, hidden argument
4. uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the return address on the stack or in a register

To maintain this stack layout, the calling sequence might operate as follows.

- In its prologue, the callee

    1. allocates a frame by subtracting an appropriate constant from the sp

    2. saves the old frame pointer into the stack, and assigns it an appropriate new value

    3. saves any callee-saves registers that may be overwritten by the current routine

To maintain this stack layout, the calling sequence might operate as follows.

- After the subroutine has completed, the epilogue
  1. moves the return value (if any) into a register or a reserved location in the stack
  2. restores callee-saves registers if needed
  3. restores the fp and the sp
  4. jumps back to the return address

To maintain this stack layout, the calling sequence might operate as follows.

- Finally, the caller

    1. moves the return value to wherever it is needed

    2. restores caller-saves registers if needed

- Many parts of the calling sequence, prologue, and epilogue can be omitted in common cases.

Inline expansion

- In some languages, programmers can actually flag routines that should be expanded inline –stack overhead is avoided.
- Example (in C++ or C99):

```
inline int max(int a,int b)
{ return a > b ? a : b}
```

- Ada does something similar, but keyword is:

```
pragma inline(max);
```

PARAMETER PASSING

Parameter Passing

- Most subroutines are parameterized:
- Parameter names that appear in the declaration of a subroutine are known as formal parameters.
- Variables and expressions that are passed to a subroutine in a particular call are known as actual parameters.
- actual parameters are also known as arguments.
- Parameter passing mechanisms have three basic implementations
  - –*value*
  - –*reference* (aliasing)
  - –*closure/ name*

Parameter Passing Modes

- Some languages—including C, Fortran, ML, and Lisp—define a single set of rules that apply to all parameters.
- Other languages, including Pascal , Modula, and Ada, provide two or more sets of rules, corresponding to different parameter-passing modes.
- The two most common parameter-passing modes
- ✓ Call-by-value &
- ✓ Call by-reference,

Parameter Passing Modes

- Suppose x is a global variable in a language, and we wish to pass x as a parameter to subroutine p: **p(x)**;
- we may provide **p with a copy of x's value**, or we may provide it **with x's address.**
- **With value parameters**, each actual parameter is assigned into the corresponding formal parameter when a subroutine is called; from then on, the two are independent.
- **With reference parameters** ,If the actual parameter is visible within the subroutine under its original name, then the two names are aliases for the same object, and changes made through one will be visible through the other

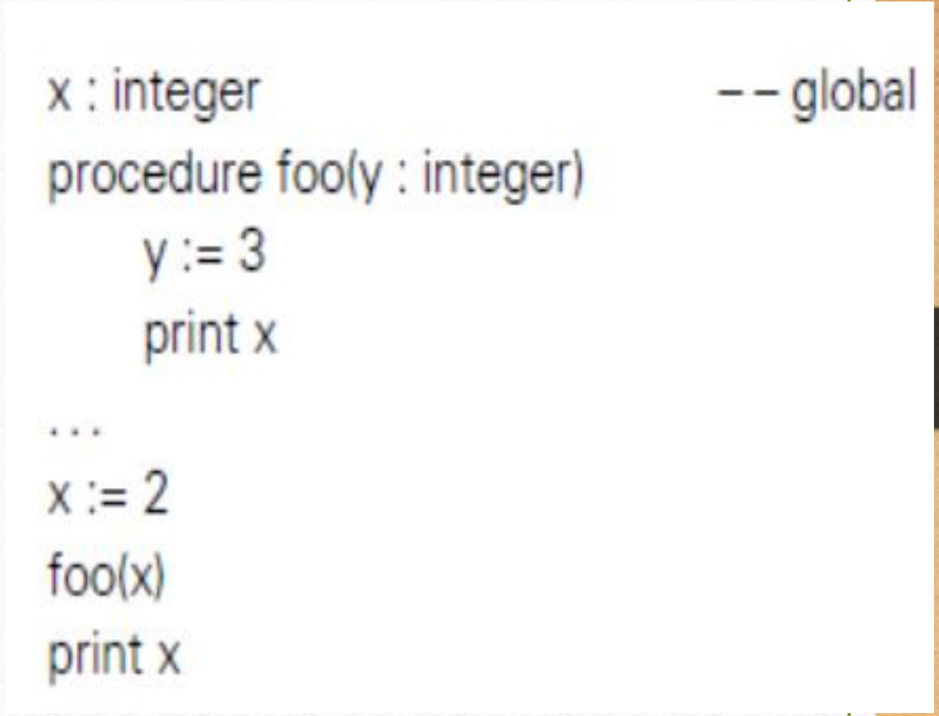
Techniques used for argument passing modes:

- call by value: copy going into the procedure
- call by result: copy going out of the procedure
- call by value result: copy going in, and again going out
- call by reference: pass a pointer to the actual parameter
- call by name: works like call by reference for simple values.

for expression it will re-evaluate the actual parameter on every use.

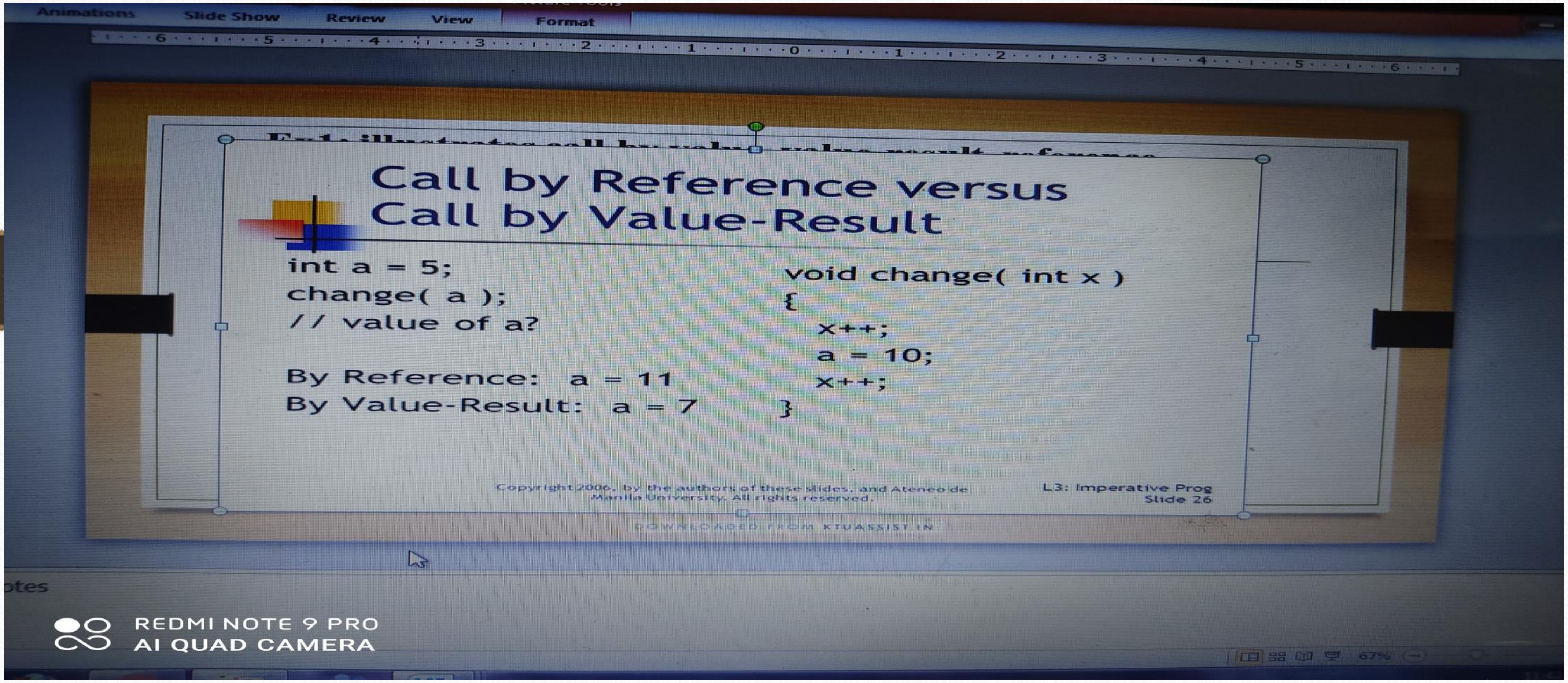
Parameter Passing Modes

- If **y is passed to foo by value**, then the assignment inside foo has no visible effect—y is private to the subroutine—and **the program prints 2 twice**.
- If **y is passed to foo by reference**, then the assignment inside foo changes x—y is just a local name for x—and **the program prints 3 twice.**
- Call by value/result copy Call-by-value/result x into y at the beginning of foo, and y into x at the end of foo. so the program would print 2 and then 3.



```
x : integer                          -- global
procedure foo(y : integer)
    y := 3
    print x
...
x := 2
foo(x)
print x
```

Ex1: illustrates call by value, value-result, reference



Call by Reference versus Call by Value-Result

```
int a = 5;                    void change( int x )
change( a );                  {
// value of a?                    x++;
                                  a = 10;
By Reference:   a = 11            x++;
By Value-Result:   a = 7      }
```

Copyright 2006, by the authors of these slides, and Ateneo de Manila University. All rights reserved.

L3: Imperative Prog Slide 26

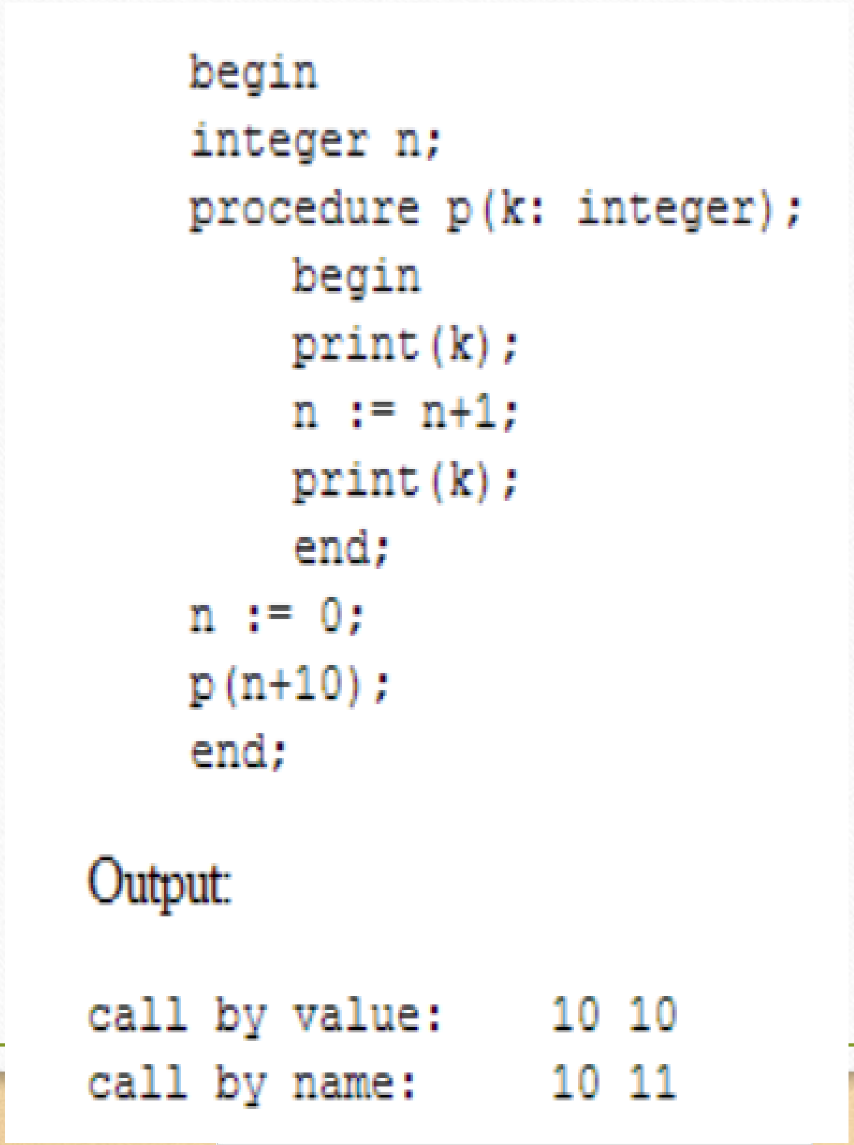
Call-by-Name

- By textual substitution
- Call by name work as call by reference when actual parameter is scalar, but be different when actual parameter is expression or array then **actual parameter is re-evaluated on each access**
- **Call by name parameter passing reevaluates actual parameter expression each time the formal parameter is read**
- Resulting semantics:

  - If actual is a scalar variable , it is pass-by-reference

  - If actual is a expression or array, then **actual parameter is re-evaluated on each access**

Example 2 : Call by value and call by name



```
begin
integer n;
procedure p(k: integer);
    begin
    print(k);
    n := n+1;
    print(k);
    end;
n := 0;
p(n+10);
end;
```

Output:

```
call by value:    10 10
call by name:     10 11
```

Parameter Passing Modes-Call-by-value/result

- Like call-by-value, **call-by-value/result copies the actual parameter into the formal parameter at the beginning of subroutine execution**.
- Unlike call-by-value, it also copies the formal parameter back into the actual parameter when the subroutine returns.
- value/result would copy Call-by-value/result x into y at the beginning of foo, and y into x at the end of foo.
- assignment of 3 into y would not affect x until after the inner print statement, so the program would print 2 and then 3.

Parameter Passing Modes-Call-by-value/

- In **Pascal**, parameters are passed by value by default;
- They are passed by reference if preceded by the keyword var in their subroutine header's formal parameter list.
- Parameters in C are always passed by value, though the effect for arrays is unusual : because of the interoperability of arrays and pointers in C
- To allow a called routine to modify a variable other than an array in the caller's scope, the C programmer must pass the address of the variable explicitly:

```
void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }

swap(&v1, &v2);
```

Parameter Passing Modes. Call-by-reference

- **Fortran** passes all parameters by reference
- If a built-up expression appears in an argument list, the compiler creates a temporary variable to hold the value, and passes this variable by reference.
- A Fortran subroutine that needs to modify the values of its formal parameters without modifying its actual parameters must copy the values into local variables, and modify those instead.

Parameter Passing Modes-Call-by-Sharing

- **Java** uses call-by-value for variables of built-in type (all of which are values), and call-by-sharing for variables of user-defined class types (all of which are references)
- A Java subroutine cannot change the value of an actual parameter of built-in type.
- When desired, parameters **in C#** can be passed by reference instead, by labelling both a formal parameter and each corresponding argument with the ref or out keyword.
- In contrast to Java, therefore, a C# subroutine can change the value of an actual parameter of built-in type, if the parameter is passed **ref or out.**

Parameter Passing Modes-Call-by-Sharing

- Language like Clu uses the mode call-by-sharing.The actual and formal parameters refer to the same object.
- It is different from call-by-value because, although we do copy the actual parameter into the formal parameter, both of them are references;
- if we modify the object to which the formal parameter refers, the program will be able to see those changes through the actual parameter after the subroutine returns.
- Call-by-sharing is also different from call-by-reference, because although the called routine can change the value of the object to which the actual parameter refers, it cannot change the identity of that object.

Purpose of Call-by-Reference

- In a language that provides both value and reference parameters (e.g., Pascal or Modula), there are **two principal reasons** why the programmer might choose **Call-by-Reference**.

- ✓ First, if the called routine is supposed to change the value of an actual parameter (argument), then the programmer must pass the parameter by reference. Conversely, to ensure that the called routine cannot modify the argument, the programmer can pass the parameter by value.

- ✓ Second, the implementation of value parameters requires copying actuals to formals, a potentially time-consuming operation when arguments are large. Reference parameters can be implemented simply by passing an address..

Read-Only Parameters

- To combine the efficiency of reference parameters and the safety of value parameters,Modula-3 provides a READONLY parameter mode.
- Any formal parameter whose declaration is preceded by READONLY cannot be changed by the called routine: the compiler prevents the programmer from using that formal parameter on the left-hand side of any assignment statement, reading it from a file, or passing it by reference to any other subroutine.
- Small READONLY parameters are generally implemented by passing a value;
- larger READONLY parameters are implemented by passing an address

Read-Only Parameters

- The equivalent of READONLY parameters is also available in C, which allows any variable or parameter declaration to be preceded by the keyword const. Const variables are "elaboration-time constants,"
- Const parameters in C parameters are particularly useful when passing addresses:

```
void append_to_log(const huge_record* r) { ...

...

append_to_log(&my_record);
```

- Here the keyword const applies to the record to which r points; the callee will be unable to change the record's contents.

Parameter Modes in Ada

- Ada provides three parameter-passing modes, called in, out, and in out.
- In parameters pass information from the caller to the callee; they can be read by the callee but not written.
- Out parameters pass information from the callee to the caller. In Ada 83 they can be written by the callee but not read; in Ada 95 they can be both read and written, but they begin their life uninitialized.
- In out parameters pass information in both directions; they can be both read and written.
- Changes to out or in out parameters always change the actual parameter

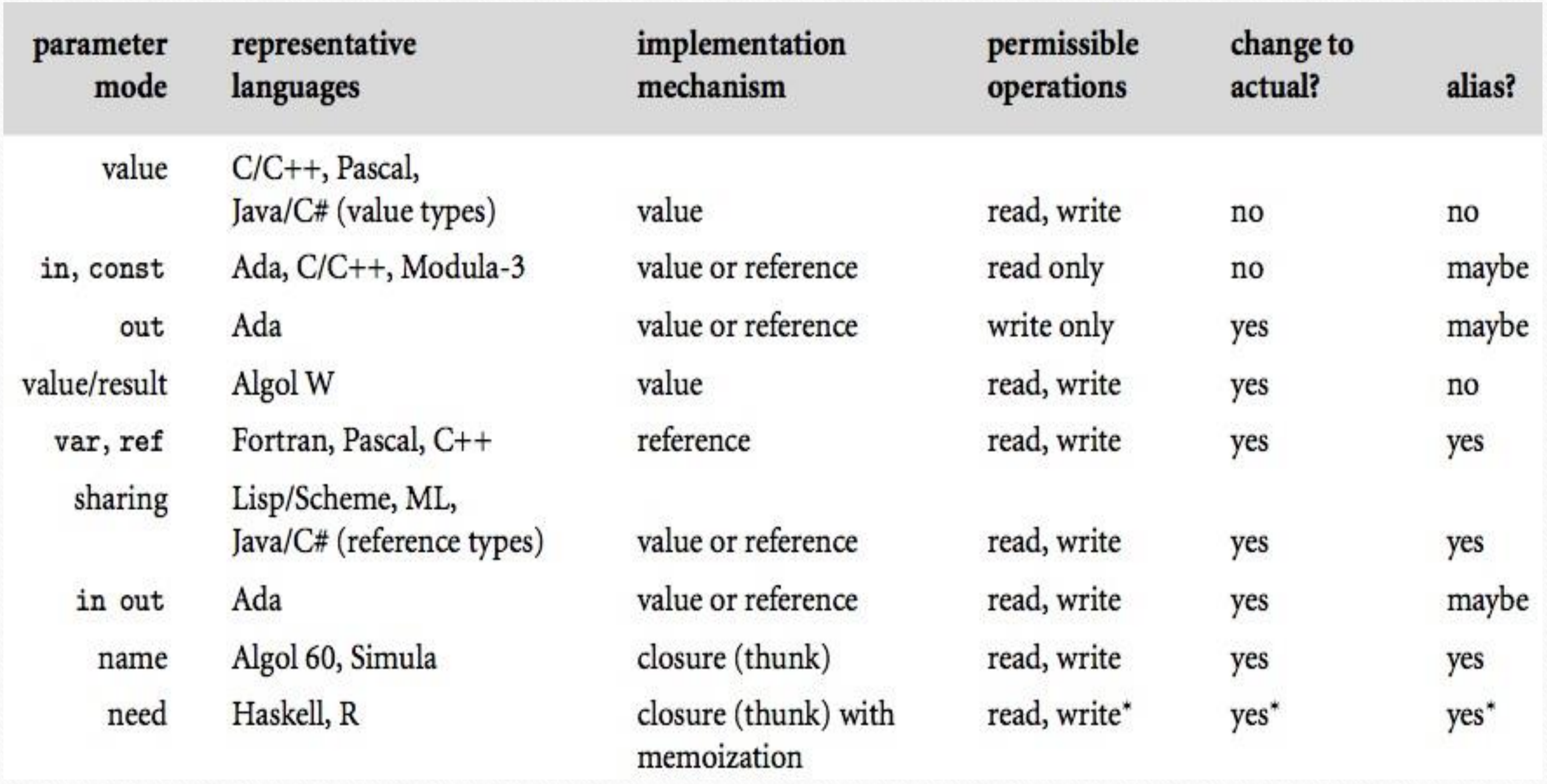
Closures as Parameters

- A closure needs to include both a code address and a referencing environment because, in a language with nested subroutines,
- Ada 83 did not permit subroutines to be passed as parameters
- Fortran has always allowed subroutines to be passed as parameters, but only allowed them to nest beginning in Fortran 90
- Subroutines are routinely passed as parameters (and returned as results) in functional languages
- C and C++have no need of subroutine closures , because their subroutines do not nest

Techniques used for argument passing:

- Fortran uses call by reference ,early FORTRANs.passing a constant
- Algol 60 has call by name, call by value. call-by-value/result
- Ada uses different designations: IN, OUT, IN OUT:
  - ✓ For scalar data types (such as integers), IN is the same as call by value, OUT is the same as call by result, and IN OUT is the same as call by value result
  - ✓ For compound data types (such as arrays), these can be implemented as above, or using call by reference.
- Lisp and Smalltalk use call-by-value with pointer semantics.
- Java uses call-by-value -- with just copying for primitive types, and pointer semantics for objects.

Parameter Modes in Ada



parameter mode	representative languages	implementation mechanism	permissible operations	change to actual?	alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
in, const	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
out	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
var, ref	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
in out	Ada	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write*	yes*	yes*

Generic Subroutines and Modules

Generic Subroutines and Modules

- Structured programming is a way to split your code in blocks.
- Subroutines is a sequence of program instructions that performs a specific task,**packaged as a unit**.
- **Modules help the programmers to split their codes into small modules**
- It is used to group the routines and data structures.
- With large programs containing many methods - subroutines will be needed to **perform similar operations on many different *types*.**

Subroutines-Add example

```
String add (String a, String b)
  return a + b;
-----------------------------------------------------------
int add ( int a, int b )
  return a + b;
-----------------------------------------------------------
  double add ( double a, double b)
  return a + b;
```

Subroutines and Modules

- Subroutine should be written once and be capable of accepting any arguments . This idea is known as generic programming.
- A generic or polymorphic subroutine is one that takes parameters of different types on different activations
- In Java version 5, generic programming has been added.
- In C++, generics are known as templates.
- Other languages that feature generics are Ada, Clu, Eiffel,Modula-3 & C#.

Subroutines and Modules

- Generic modules/classes: useful for creating “containers”
- Examples of containers include stack , queue, heap, set, dictionary ,lists , arrays, trees, or hash tables.
- A generic or polymorphic subroutine is one that takes parameters of different types on different activations
- Generic subroutines are needed in generic classes.
- They allow a method to be parameterised by a single type.

Subroutines and Modules

- Hence our earlier example becomes vastly simplified:

```
public T <T> add ( T a, T b)
return a + b;
```

- we now have a generic type **T** passed as a parameter to the add method.
- Hence we can now call the code:

```
int c = add( 5 , 7 );
double d = add ( 4.5, 6.9 );
String concat = add( “gen” , “erics” );
```

Implementing Generics

- In C++, they are a static mechanism.
- All of the work required to use multiple instances of the generic code is handled by the compiler.
- The compiler creates a *separate copy* of the code for every instance.
- In Java, *all* instances of the generic code will share that same code at run-time.
- If we call some generic parameter, **T** in Java, this behaves identically to java.lang.Object except that we do not have to use casts.

Generics in C++

- A generic add method in C++:

```
template<typename T>  T add (T a, T b)
{
    return a+b;
}
```

- To call the method

```
add(5,6);
add(5.6 , 7.8);
```

Java Generics

- Recall the C++ example:

```
template<typename T>  T add (T a, T b)
  { return a+b;
  }
```

- This allows any number (int, float, short, double etc) to be added together.
- How can we achieve this is Java?

```
public <T> T add ( T a, T b )
{     return a+b;
}
```

Java Generics

- Lets perform the type erasure:

```
public Object add ( Object a, Object b )
{
  return a+b;
}
```

- Does this compile?
- No!, because the + operator is not applicable to Object.
- We must modify the code to make it typesafe so it can compile:

Java Generics

```
public <T extends Number> T add ( T a, T b )
  {
    return a+b;
  }
```

- We have to impose a *higher bound* on the type passed to the generic method.
- In this case we are saying that the type passed in will be a *subclass* of `java.lang.Number`
- Will the code compile now?

Java Generics

```
public <T extends Number> T add ( T a, T b )
  {     if ( T instanceof Integer )
        return a.intValue() + b.intValue();
     if ( T instanceof Double )
        return a.doubleValue() + b.doubleValue();
     return null;
  }
public static void main( String args [] )
  {int a = add (5 , 6);
     double b = add ( 7.9, 11.3 );
```

Generics in C++, Java, and C#

- C++(templates)is the most ambitious of the three.
- Java 5 and C# 2.0 provide generics purely for the sake of polymorphism.
- Java's design was heavily influenced by the desire for backward compatibility,.
- The C# designers , though building on an existing language, did not feel as constrained.
- They had been planning for generics from the outset, and were able to engineer substantial new support into the .NET virtual machine.

Exception Handling

Exception Handling

- An exception is a special unexpected error condition at run time
- Built-in exceptions may be detected automatically by the language implementation
- Exceptions can be explicitly raised
- Exceptions are handled by exception handlers to recover from error conditions.
- Exception handlers are user-defined program fragments that are executed when an exception is raised

Exception Handling

- What is an exception?

A hardware-detected run-time error or unusual condition detected by software

- Examples
- ✓ arithmetic overflow
- ✓ end-of-file on input
- ✓ wrong type for input data
- ✓ user-defined conditions, not necessarily errors

Exception Handling

- An exception can be defined as an unexpected—or at least unusual—condition that arises during program execution, and that cannot easily be handled in the local context.
- It may be detected automatically by the language implementation, or the program may raise it explicitly
- Exception-handling mechanisms address these issues by moving error-checking code “out of line,” allowing the normal case to be specified simply, and arranging for control to branch to a handler when appropriate.

Exception Handling

- What is an exception handler?

  code executed when exception occurs may need a different handler for each type of exception

- Purpose of an exception handler:

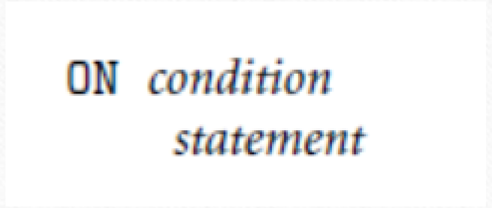
  1.Recover from an exception to safely continue execution

  2.If full recovery is not possible, print error message(s)

  3.If the exception cannot be handled locally, clean up local resources and re-raise the exception to propagate it to another handler

exception handlers can be attached to a collection of program statements

Exception Handling



```
ON  condition
        statement
```

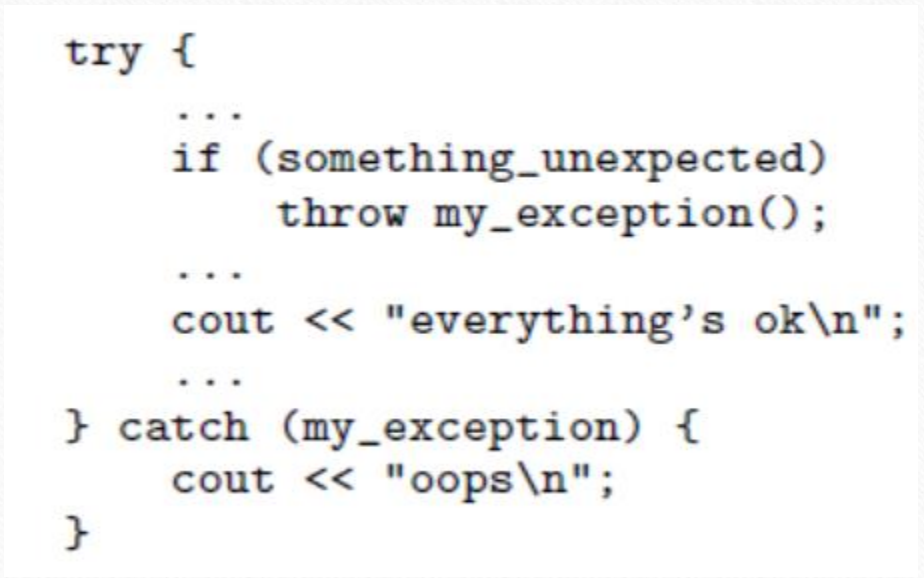
- Exception handling in PL/I, which includes an executable state-ON conditions
- The nested statement (often a GOTO or a BEGIN...END block) is a handler.
- It is not executed when the ON statement is encountered, but is "remembered" for future reference. It will be executed later if exception condition (e.g., OVERFLOW) arises.
- Because the ON statement is executable, the binding of handlers to exceptions depends on the flow of control at run time

Exception Handling

- languages like Clu, Ada, Modula-3, Python, PHP , Ruby, C++, Java, C#, and ML, all provide exception-handling facilities in which handlers are lexically bound to blocks of code,
- In C++ or Lisp, all are programmer defined
- In PHP, set_error_handler can be used to turn semantic errors into ordinary exceptions
- In Ada, exception is a built-in type, and so you can easily make your own: declare empty_queue : exception;
- Most languages allow a throw or raise in an if to raise an exception

Exception Handling in C++

- If something_unexpected occurs, this code will throw an exception, and the catch block will execute in place of the remainder of the try block



```
try {
    ...
    if (something_unexpected)
        throw my_exception();
    ...
    cout << "everything's ok\n";
    ...
} catch (my_exception) {
    cout << "oops\n";
}
```

- if an exception is not handled within the current subroutine,then the subroutine returns abruptly and the exception is raised at the point of call:
- If the exception is not handled in the calling routine, it continues to propagate back up the dynamic chain.
- If it is not handled in the program's main routine, then a predefined outermost handler is invoked, and usually terminates the program.

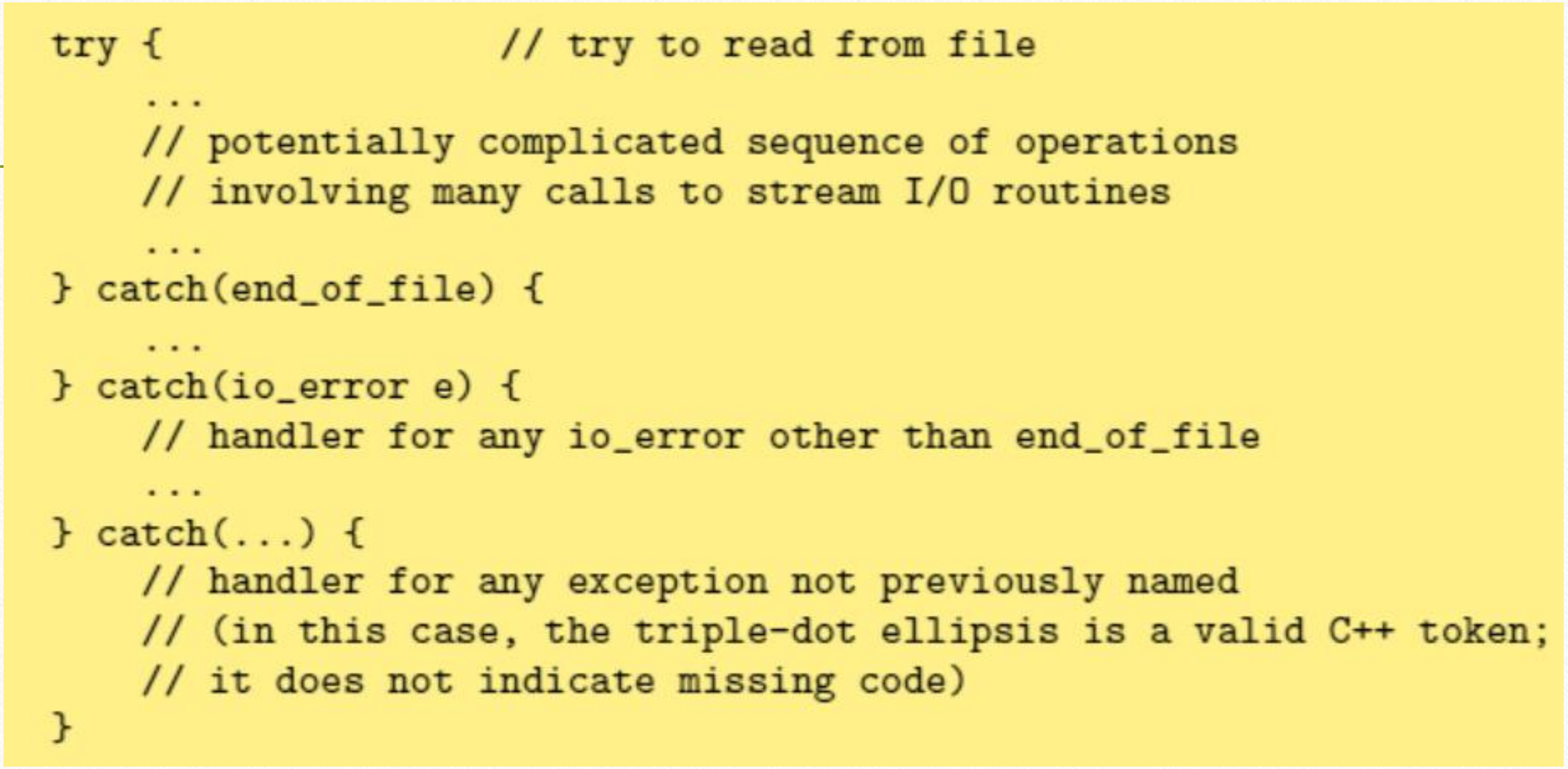
Exception Handling in C++

```
try {
...foo();
...
cout << "everything’s ok\n";
...
} catch (my_exception) {
cout << "oops\n";
}
void foo() {
...
if  (something_unexpected)
throw my_exception();
...}
```

Exception Handling -Exception Propagation

- In most languages, a block of code can have a list of exception handlers
- When an exception arises, the handlers are examined in order;
- control is transferred to the first one that matches the exception.
- In C++, a handler matches if it names a class from which the exception is derived, or if it is a catch-all (...).

Exception Handling -Exception Propagation



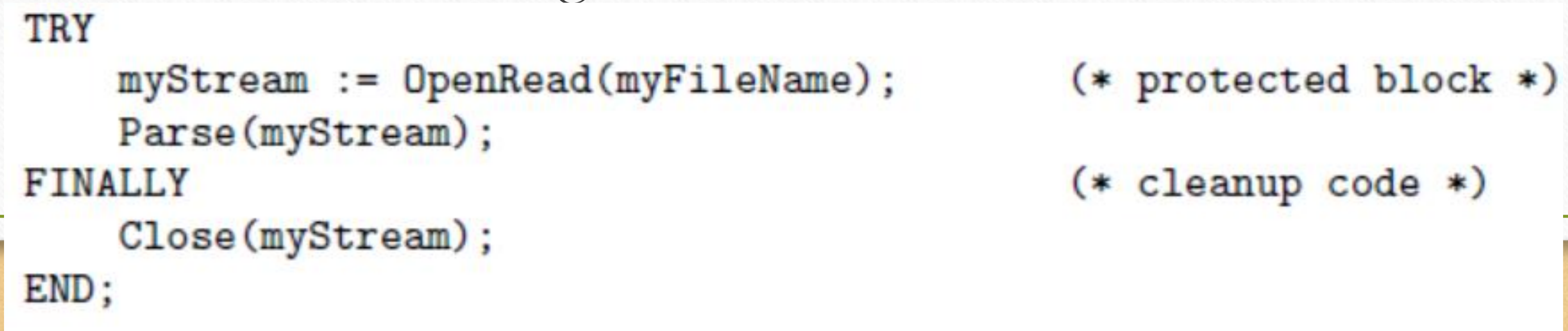
```
try {                   // try to read from file
    ...
    // potentially complicated sequence of operations
    // involving many calls to stream I/O routines
    ...
} catch(end_of_file) {
    ...
} catch(io_error e) {
    // handler for any io_error other than end_of_file
    ...
} catch(...) {
    // handler for any exception not previously named
    // (in this case, the triple-dot ellipsis is a valid C++ token;
    // it does not indicate missing code)
}
```

Exception Handling -Exception Propagation

- In the example here, let us assume that end_of_file is a subclass of io_error.
- Then an end_of_file exception, if it arises, will be handled by the first of the three catch clauses. All other I/O errors will be caught by the second; all non-I/O errors will be caught by the third.
- If the last clause were missing, non-I/O errors would continue to propagate up the dynamic chain.

Exception Handling -Cleanup Operations

- In C++, an exception that leaves a scope, to call destructor functions for any object s declared within that scope.
- Destructors are often used to deallocate heap space and other resources (e.g., open files).
- in Common Lisp by an unwind-protect expression, and
- in Modula-3, Python, Java, and C# by means of try. . . finally
- Code in Modula-3 might look like this:



```
TRY
    myStream := OpenRead(myFileName);        (* protected block *)
    Parse(myStream);
FINALLY                                       (* cleanup code *)
    Close(myStream);
END;
```

Exception Handling -Implementation of Exceptions

- If a protected block of code has handlers for several different exceptions, they are implemented as a single handler containing a multi arm if statement:

```
if exception matches end of file
. . .
elsif exception matches io error
. . .
else
. . . –– "catch-all" handler
```

Exception Handling without Exceptions

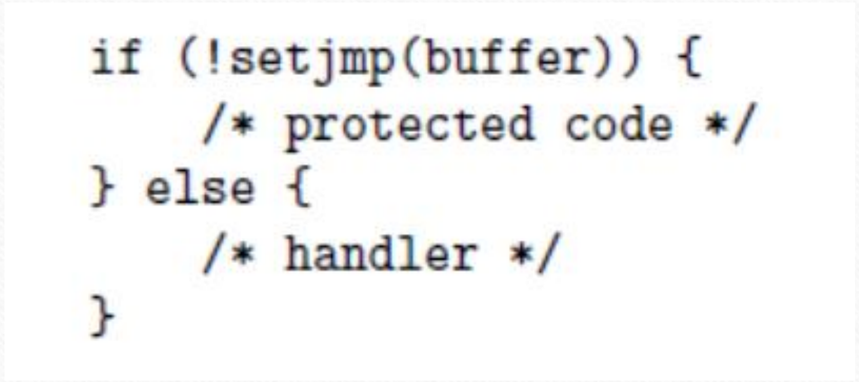
- exceptions can sometimes be simulated in a language that does not provide them as a built-in.
- most versions of C provide a pair of library routines entitled **setjmp and longjmp.**
- Setjmp takes as argument a buffer into which to capture a representation of the program’s current state.
- This buffer can later be passed to longjmp to restore the captured state.
- Setjmp has an integer return type: zero indicates “normal” return; nonzero indicates “return” from a longjmp

setjmp and longjmp.

- **setjmp(jmp_buf buf)** : uses buf to remember current position and returns 0.
- **longjmp(jmp_buf buf, i)** : Go back to place buf is pointing to and return i
- **setjmp** can be used like **try** (in languages like C++ and Java).
- The call to **longjmp** can be used like **throw** (Note that longjmp() transfers control to the point set by setjmp()).

Exception Handling without Exceptions

- When initially called, setjmp returns a 0, and control enters the protected code.

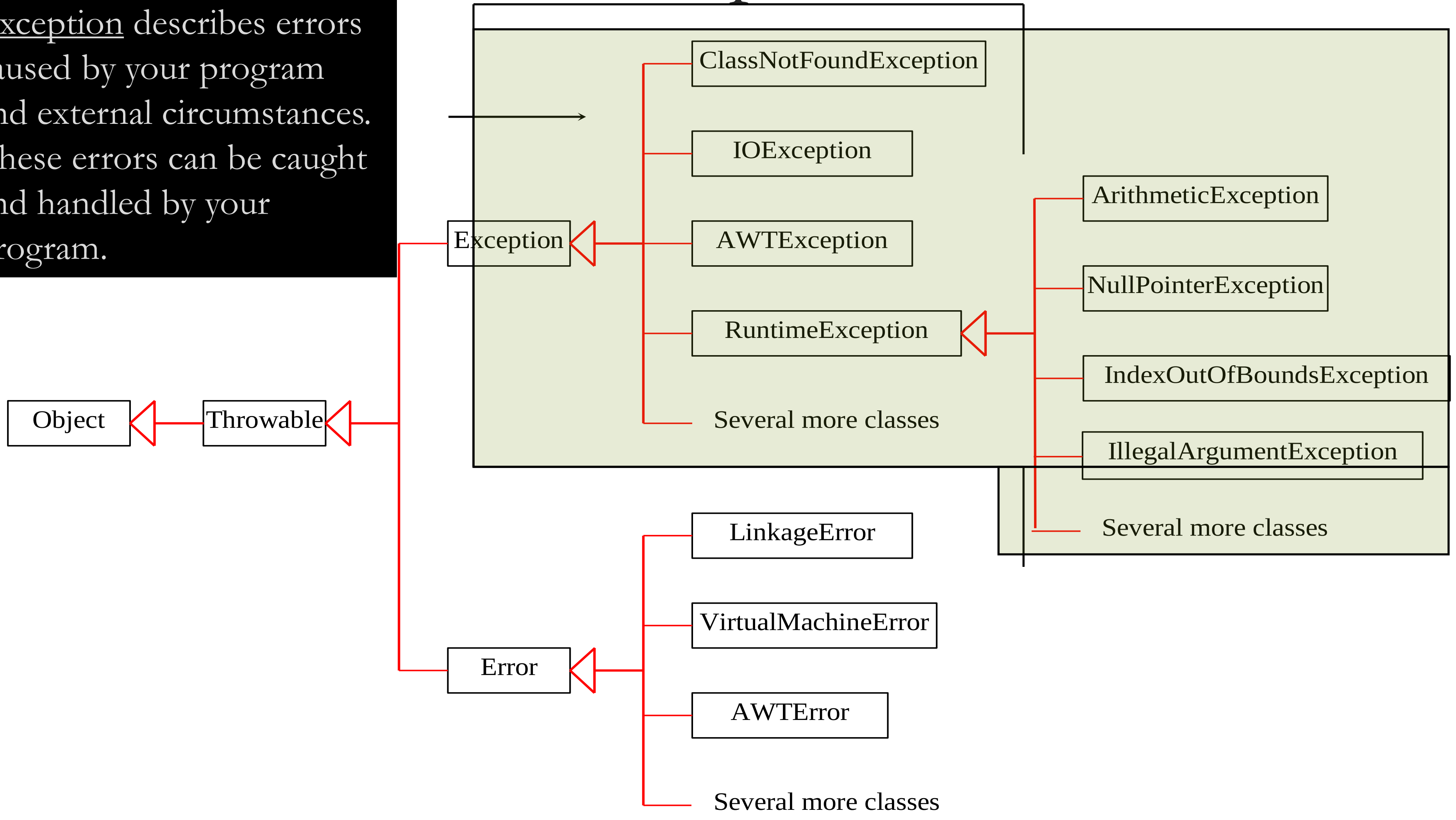


```
if (!setjmp(buffer)) {
      /* protected code */
} else {
      /* handler */
}
```

- If longjmp(buffer, v) is called anywhere within the protected code, or in subroutines called by that code, then setjmp will appear to return again, this time with a return value of v, causing control to enter the handler.
- *Setjmp and longjmp are usually implemented by saving the current machine registers in the setjmp buffer, and by restoring them in longjmp*

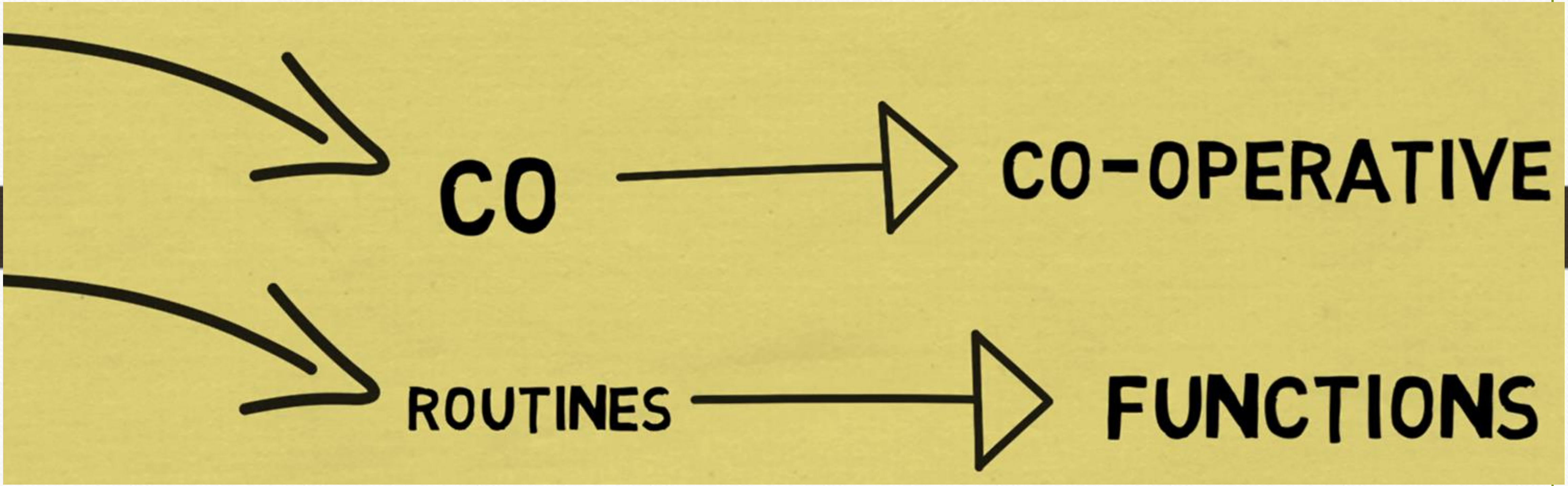
Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.

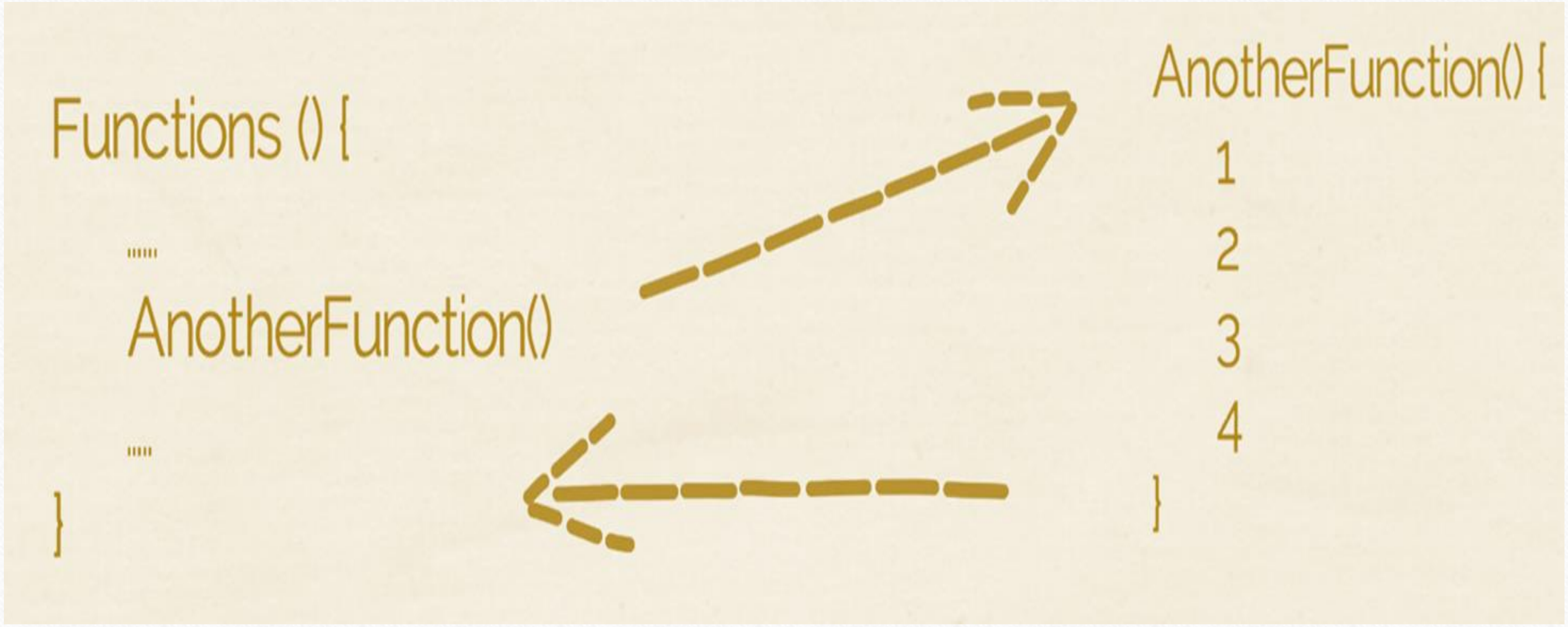


COROUTINES

COROUTINES



Normal ROUTINES



```
function () {

   AnotherFunction()

   Okay..I'll do 'a'

  Hey Why don't you continue

   ok.. I will do b....
   c.....
  You now finish rest of your job

   d....
e....
}
```

```
AnotherFunction() {
   1......
    // I am done now.. will do rest later

   Ok..
   2......
   3......
  You know what.. Its your turn...

   OK....
   5....
   6...
      } // and return
```

Even Better.. What if we could also pass parameters around...??

```
Function () {

   AnotherFunction()

   Thank you for give me "10"

   Now you continue with value "20"

   b....
   c....
   }
```

```
AnotherFunction () {

   1.....
    You continue with a value "10"

    Thankyou for giving  "20"
    2....

    3...
    }
```

Coroutines

- A coroutines is a subprogram that has multiple entries and controls them itself. Also called symmetric control
- A coroutine call is named a resume
- coroutines repeatedly resume each other, possibly forever
- Coroutines provide concurrent execution of program units (the coroutines)
- Their execution is interleaved, but not overlapped

Coroutines

- The first resume of a coroutine is to its beginning , but subsequent calls enter at the point just after the last executed statement in the coroutine
- A coroutine changes the continuation every time it runs.
- When we transfer from one coroutine to another, our old program counter is saved:the coroutine we are leaving is updated to reflect it.
- if we perform a transfer into the same coroutine multiple times, each jump will take up where the previous one left off.

Coroutines

- ❑ Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name
- ❑ Coroutines can be used to implement
  - ✓ iterators
  - ✓ threads
- ❑ Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack

Coroutines-Stack Allocation

- Because they are concurrent (i.e., simultaneously started but not completed),coroutines cannot share a single stack: their subroutine calls and returns, taken as a whole, do not occur in last-in-first-out order.
- The simplest solution is to give each coroutine a fixed amount of statically allocated stack space.
- This approach is adopted in Modula-2, which requires the programmer to specify the size and location of the stack when initializing a coroutine.

Coroutines-*cactus* Stack

- If coroutines can be created at arbitrary levels , then two or more coroutines may be declared in the same nonglobal scope, and must thus share access to objects in that scope.
- To implement this sharing, the run-time system must employ a so-called cactus stack.
- Each branch off the stack contains the frames of a separate coroutine.

Coroutines-*cactus* Stack

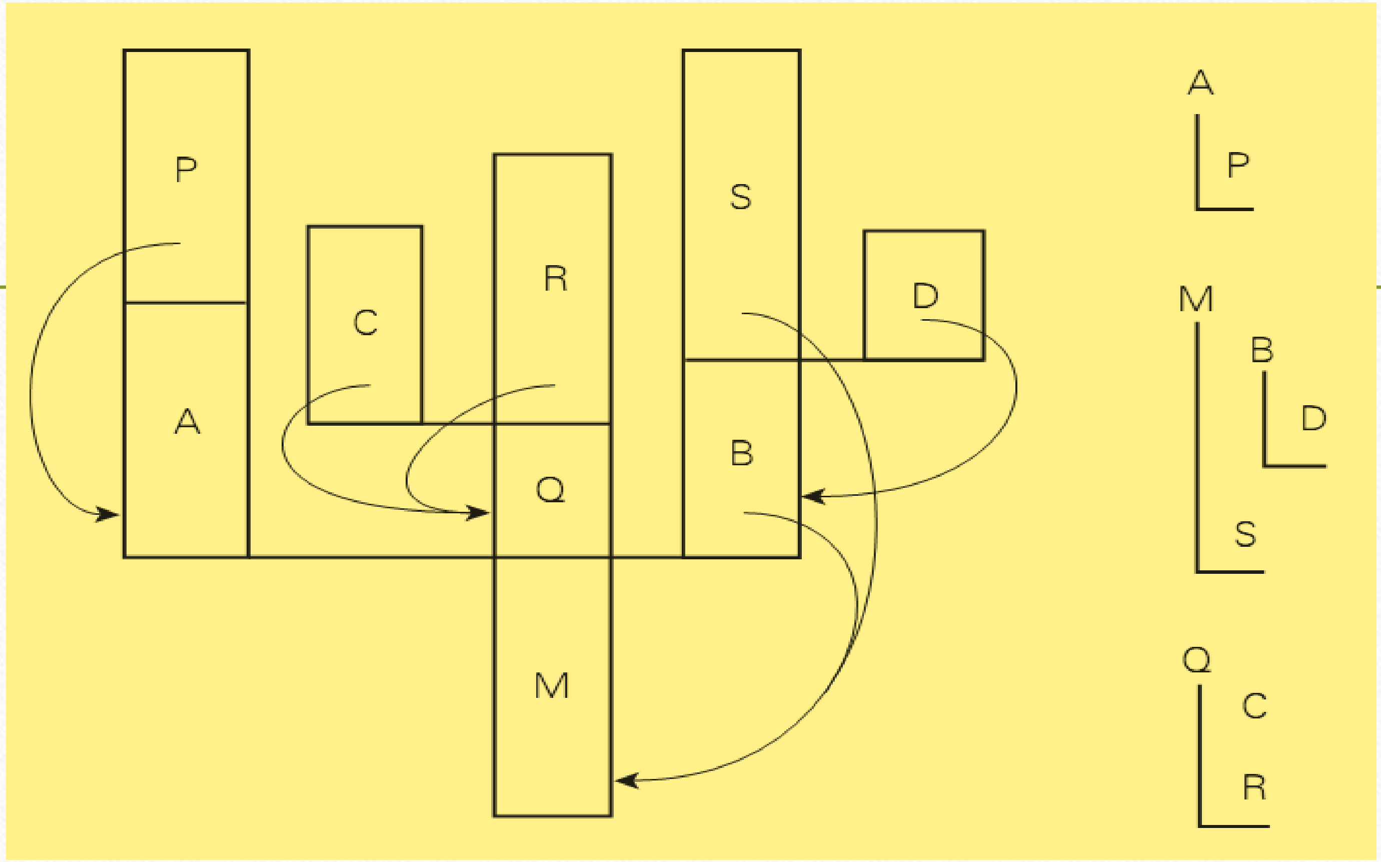


Figure A cactus stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)

Coroutines-*cactus* Stack

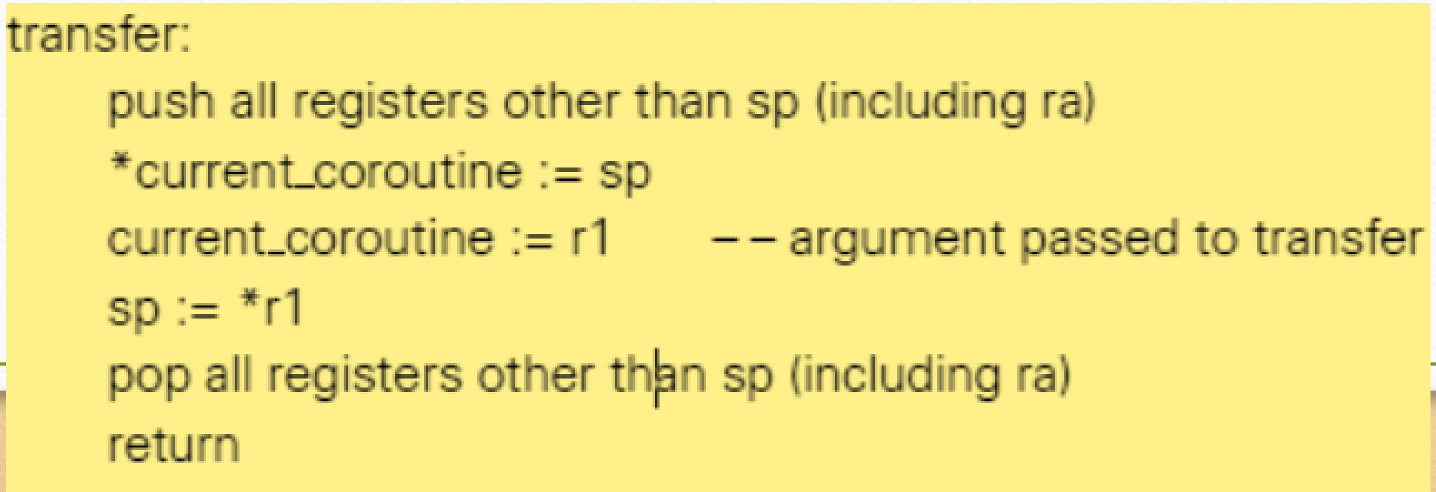
- The dynamic chain of a given coroutine ends in the block in which the coroutine began execution.
- The static chain of the coroutine, extends down into the remainder of the cactus, through any of surrounding blocks
- Returning from the main block of a coroutine will generally terminate the program as a whole.
- When a given **coroutine is no longer needed**, the **Modula-2** programmer can **simply reuse its stack space**.
- **In Simula, the space will be reclaimed via garbage collection**

Coroutines-*Transfer*

- To transfer from one coroutine to another, the run-time system must change the program counter (PC), the stack, and the contents of the processor's registers.
- These changes are encapsulated in the transfer operation: one coroutine calls transfer; a different one returns.
- Because the change happens inside transfer , changing the PC from one coroutine to another simply amounts to remembering the right return address:

Switching coroutines

- to change the stack ,change the stack pointer register,and to avoid using the frame pointer inside of transfer itself.
- At the beginning of transfer we **push the return address and all of the other callee saves registers** onto the current stack.
- We then change the sp, pop the (new) return address (ra) and other registers off the new stack, and return:



```
transfer:
        push all registers other than sp (including ra)
        *current_coroutine := sp
        current_coroutine := r1        -- argument passed to transfer
        sp := *r1
        pop all registers other than sp (including ra)
        return
```