Module 3

Subroutines and Control Abstraction: - Static and Dynamic Links, Calling Sequences, Parameter Passing, Generic Subroutines and Modules, Exception Handling, Coroutines.

Stack Layout: 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, local variables, and/or temporaries.
When a subroutine returns, its frame is popped from the stack.
The *stack pointer* register contains the address of either the last used location at the top of the stack, or the first unused location, depending on convention.
The *frame pointer* register contains an address within the frame.

Static and Dynamic links: In a language with nested subroutines and static scoping, objects that lie in surrounding subroutines, and that are thus neither local nor global, can be found by maintaining a static chain.
Each stack frame contains a reference to the frame of the lexically surrounding subroutine.
This reference is called the static link.

- 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.

We can be sure that the surrounding routine is active.
There is no other way that the current routine could have been visible, allowing it to be called.

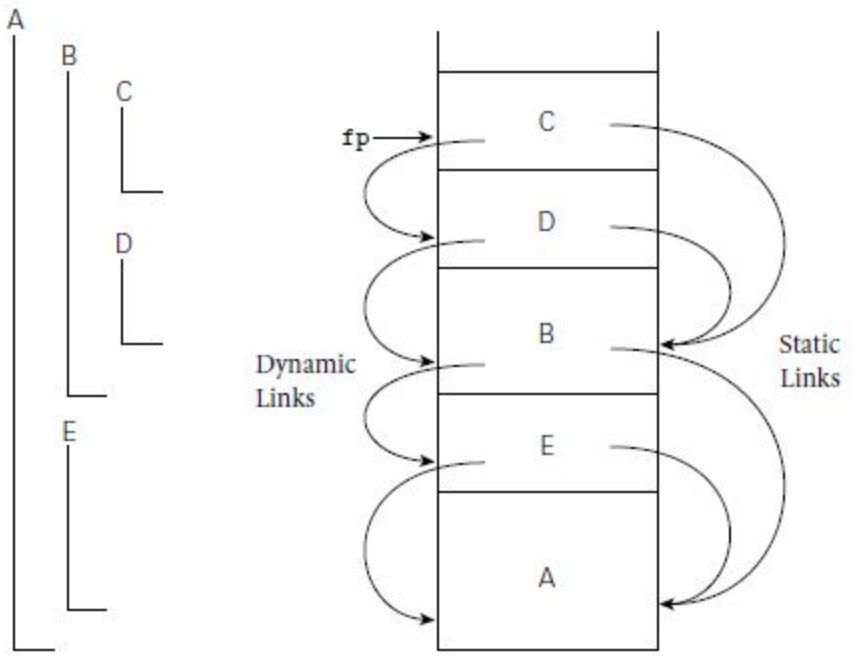


Fig: Example of subroutine nesting

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.
If subroutine D is called directly from B, then clearly B's frame will already be on the stack.
D is nested inside of B, when control enters B that D comes into view.
It can therefore be called by C, or by any other routine that is nested inside C or D, but only because these are also within B.
Calling Sequences: Maintenance of the subroutine call stack is the responsibility of the calling sequence.
Sometimes the term calling sequence is used to refer to the combined operations of the caller, the prologue, and the epilogue.

- Tasks that must be accomplished on the way into a subroutine include passing parameters, saving the return address, changing the program counter, saving registers that contain important values and that may be overwritten by the callee etc.
- Tasks that must be accomplished on the way out include passing return parameters or function values, executing finalization code for any local objects that require it, deallocating the stack frame etc.
- Some of these tasks must be performed by the caller, because they differ from call to call.

Saving and Restoring Registers The trickiest division-of-labor issue pertains to saving registers. The ideal approach is to save precisely those registers that are both in use in the caller and needed for other purposes in the callee.
Because of separate compilation, it is difficult to determine this intersecting set.
A simpler solution is for the caller to save all registers that are in use, or for the callee to save all registers that it will overwrite
Registers not reserved for special purposes are divided into two sets of approximately equal size.
Maintaining the Static Chain: In languages with nested subroutines, at least part of the work required to maintain the static chain must be performed by the caller, rather than the callee. The standard approach is for the caller to compute the callee's static link and to pass it as an extra, hidden parameter.

- The stack pointer (sp) points to the first unused location on the stack.
- 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.
- Fig: shows 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.

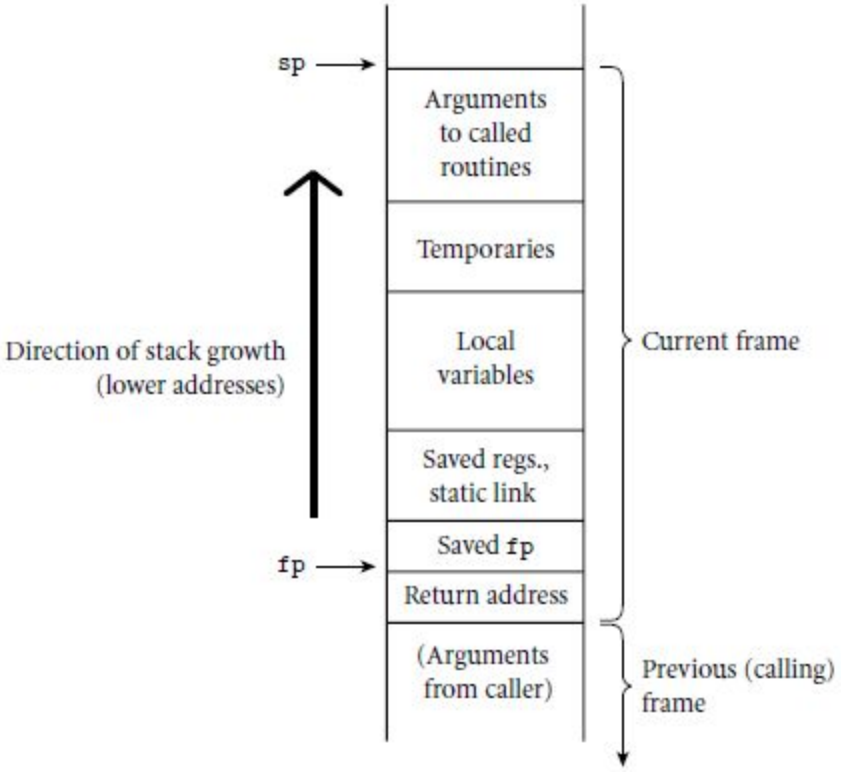
- Local variables and temporaries are accessed at negative offsets from the fp.
- Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

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

A Typical Calling Sequence:



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

After the subroutine has completed, the epilogue,

1. Moves the return value 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

Special-Case Optimizations: Many parts of the calling sequence, prologue, and epilogue can be omitted in common cases.
If the hardware passes the return address in a register, then a *leaf routine* can simply leave it there; it does not need to save it in the stack.
Likewise it need not save the static link or any caller-saves registers.
Displays: One disadvantage of static chains is that access to an object in a scope K levels out requires that the static chain be dereferenced K times.

If a local object can be loaded into a register with a single memory access, an object K levels out will require K+1 memory access.

Register Windows:- As an alternative to saving and restoring registers on subroutine calls and returns, Berkeley RISC machines introduced a hardware mechanism known as register windows.

The basic idea is to map the ISA's limited set of register names onto some subset of a much larger collection of physical registers, and to change the mapping when making subroutine calls. Old and new mapping overlap a bit, allowing arguments to be passed in the intersection.

In-Line Expansion:- Many language implementations allow certain subroutines to be expanded in –line at the point of call.

A coy of the "called" routine becomes a part of the "caller"; no actual subroutine call occurs.

In –line expansion avoids a variety of overheads, including space allocation, branch delays from the call and return etc.

In C++ and C99, the keyword 'inline' can be prefixed to a function declaration:

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

In Ada, the programmer can request in –line expansion with a significant comment, or pragma:

```
function max (a, b : integer) return integer is

begin

  If a>b then return a; else return b; end if;

end max;

pragma inline(max);
```

Consider a binary tree whose leaves contain character strings.
A routine to return the fringe of this tree might look like this:

```
string fringe(bin_tree *t) {
// assume both children are nil or neither is
if (t->left == 0) return t->val;
return fringe(t->left) + fringe(t->right);
}
```

A compiler can expand this code in-line if it makes each nested invocation a true subroutine call.

Parameter Passing: A parameter is a special kind of variable, used in a subroutine to refer to one of the pieces of data provided as input to the subroutine.

These pieces of data are called arguments.

- An ordered list of parameters is usually included in the definition of a subroutine, so that, each time the subroutine is called, its arguments for that call can be assigned to the corresponding parameters.
- 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.
- Lisp places the function name inside the parentheses, as in (max a b).

The following program in C defines a function that is named "sales_tax" and has one parameter named "price".

The type of price is "double" .

The function's return type is also a double.

```
double sales_tax(double price)

{
   return 0.05 * price;
}
```

After the function has been defined, it can be invoked as follows:

```
sales_tax(10.00);
```

The function has been invoked with the number 10.00.

When this happens, 10.00 will be assigned to price, and the function begins calculating its result.

Most languages use a prefix notation for calls to user-defined subroutines, with the subroutine name followed by a parenthesized argument list.

Lisp places the function name inside the parentheses, as in (max a b).
ML allows the programmer to specify that certain names represent infix operators, which appear between a pair of arguments:

```
infixr   8 tothe;          (* exponentiation *)
fun x tothe   0 = 1.0
| x tothe  n = x * (x   tothe(n-1));         (* assume n >= 0 *)
```

The infixr declaration indicates that tothe will be a right-associative binary infix operator, at precedence level 8.

Parameter 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 set of rules, corresponding to different parameter passing modes.

Suppose for the moment that 'X' is a global variable in a language with a value model of variables, and that we wish to pass 'X' as a parameter to subroutine 'P':

P(X);

We have two principal alternatives: we may provide 'P' with a copy of X's value, or we may provide it with X's address.

- The two most common parameter- passing modes, called call-by-value and call-by-reference, are designed to reflect these implementations.

[call-by-value- a parameter acts within the subroutine as a local (isolated) copy of the argument.

call-by-reference- the argument supplied by the caller can be affected by actions within the called subroutine]

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, each formal parameter introduces, within the body of subroutine, a new name for the corresponding actual parameter.

Variations on Value and Reference Parameters**:-** If the purpose of call-by-reference is to allow the called routine to modify the actual parameter, we can achieve a similar effect using 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.

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

Here value/result would copy x into y at the beginning of foo, and y into x at the end of foo.
foo accesses x directly in –between, and the program's visible behavior would be different than it was with call-by-reference.
The assignment of 3 into y would not affect x until after the inner print statement, so the program would print 2 and then 3.

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);
Fortran passes all parameters by reference, but does not require that every actual parameter be an l-value.
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.

Call-by-Sharing:- Call –by-value and call-by-reference make the most sense in a language with a value model of variables: they determine whether we copy the variable or pass an alias for it. It is most natural simply to pass the reference itself, and let the actual and formal parameters refer to the same object.
Clu calls this mode call-by-sharing.

- It is different from call-by-value because, although we do copy the actual parameter into the formal parameter, both of them are references
- Call-by-sharing is also different from call-by-reference.
- 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.

The Purpose of Call-by-Reference: In a language that provides both value and reference parameters, there are two principal reasons why the programmer might choose one over the other.
First, if the called routine is supposed to change the value of an actual parameter, then the programmer must pass the parameter by reference.
Second the implementation of value parameters requires copying actual to formals, a potentially time-consuming operation when arguments are large.
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.
Small READONLY parameters are generally implemented by passing a value; larger READONLY parameters are implemented by passing an address.
The equivalent of READONLY parameters is also available in C, which allows any variable or parameter declaration to be preceded by the keyword const.
void append_to_log(const huge_record * r) {.....
.....

append _to_log (&my_record);

Reference parameters in C++:- Programmers who switch to C after some experience with Pascal, Modula, or Ada are often frustrated by C's lack of reference parameters.
One can always arrange to modify an object by passing its address, but then the formal parameter is a pointer, and must be explicitly dereferenced whenever it is used.
C++ addresses this problem by introducing an explicit notion of a *reference.*
void swap (int &a, int &b) { int t = a; a=b; b=t; }
Closures as Parameters: A closure may be passed as a parameter for any of several reasons.
The most obvious of these arises when the parameter is declared to be a subroutine.
A closure needs to include both a code address and a referencing environment because, in a language with nested subroutines, we need to make sure that the environment available is the same that would have been available if it had been called directly.
Call-by-Name: Explicit subroutine parameters are not the only language feature that requires a closure to be passed as a parameter.
A language implementation must pass a closure whenever the eventual use of the parameter requires the restoration of a previous referencing environment.
Examples are the *call-byname* parameters of Algol 60 and Simula, the *call-by-need* parameters of Miranda, Haskell, and R etc..
Special-Purpose Parameters
Conformant Arrays: A formal array parameter whose shape is finalized at run time, is called a *conformant*, or *open*, array parameter.
Default (Optional) Parameters :Default parameter is one that need not necessarily be provided by the caller; if it is missing, then a pre established default value will be used instead.
One common use of default parameters is in I/O library routines.
The built-in type *attribute* width determines the maximum number of columns required to print an integer in decimal on the current machine
Named Parameters :Named parameters (*keyword* parameters) are particularly useful in conjunction with default parameters.
Positional notation allows us to write put(37, 4) to print "37" in a four-column field, but it does not allow us to print in octal in a field of default width.
Any call that specifies a base must also specify a width, explicitly, because the width parameter precedes the base in put's parameter list.
Variable Numbers of Arguments : Lisp, Python, and C and its descendants are unusual in that they allow the user to define subroutines that take a variable number of arguments.
The ellipsis (...) in the function header is a part of the language syntax.
It indicates that there are additional parameters following the format, but that their types and numbers are unspecified.
In C, args is defined as an object of a special type used to enumerate the elided parameters.
Function Returns: The syntax by which a function indicates the value to be returned varies greatly.
In languages like Lisp, ML, and Algol 68, the value of a function is simply the value of its body, which is itself an expression.
An explicit return statement is like:
return *expression*

In addition to specifying a value, return causes the immediate termination of the subroutine. A function that has figured out what to return but doesn't want to return yet can always assign the return value into a temporary variable, and then return it later:

```
rtn := expression
...
return rtn
```

In Python, for example, we might write:

```
def foo():
return 2, 3
...
i, j = foo()
```

Parameters and Arguments:-These two terms are sometimes loosely used interchangeably; in particular, "argument" is sometimes used in place of "parameter".

Parameters appear in procedure definitions; arguments appear in procedure calls.

Arguments are more properly thought of as the actual values or references assigned to the parameter variables when the subroutine is called at run-time.

Any values or references passed into the subroutine are the arguments, and the place in the code where these values or references are given is the *parameter list*.

In C, when dealing with threads it's common to pass in an argument of type void* and cast it to an expected type:

```
void ThreadFunction( void* pThreadArgument )
{
   // Naming the first parameter 'pThreadArgument' is correct, rather than 'pThreadParameter'.
   // At run time the value we use is an argument.
 Reserve the
   // term parameter when discussing subroutine definitions.
}
```

The term *formal parameter* refers to the variable as found in the function definition (*parameter*), while *actual parameter* refers to the actual value passed (*argument*).

```
int sumValue;
int value1 = 40;
int value2 = 2;
sumValue = sum(value1, value2);
```

The variables *value1* and *value2* are initialized with values.

value1 and *value2* are both arguments to the *sum* function in this context.

At runtime, the values assigned to these variables are passed to the function *sum* as arguments.

Generic Subroutines and Modules: Subroutines provide a natural way to perform an operation for a variety of different object values.

An operating system tends to make heavy use of queries, to hold processes, memory descriptors, file buffers, device control blocks, and a host of other objects.

The standard mechanisms for declaring enqueue and dequeue subroutines in most languages require that the type of the items be declared statically.
Generic modules or classes are particularly valuable for creating containers- data abstractions that hold a collection of objects, but whose operations are generally oblivious to the type of those objects.

- Generic subroutines are needed in generic modules, and may also be useful in their own right.
- Generics can be implemented several ways.
- In most implementations of Ada and C++ they are a purely static mechanism: all the work required to create and use multiple instances of the generic code takes place at compile time.
- If several queues are instantiated with the same set of arguments, then the compiler may share the code of the enqueue and dequeue routines among them.

C++ calls its generics *templates*.
Checks for overflow and underflow have been omitted.
Implementation Options: Generics can be implemented several ways.
In most implementations of Ada and C++ they are a purely static mechanism.
All the work required to create and use multiple instances of the generic code takes place at compile time.
The compiler creates a *separate copy* of the code for every instance.
If several queues are instantiated with the same set of arguments, then the compiler may share the code of the enqueue and dequeue routines among them.
Generic Parameter Constraints:-Because a generic is an abstraction, it is important that its interface provide all the information that must be known by a user of the abstraction.
Several languages, including Clu, Ada, Java,and C#, attempt to enforce this rule by constraining generic parameters.

type item is private;

A private type in Ada is one for which the only permissible operations are assignment, testing for equality and inequality, and accessing a few standard attributes.

In simple cases, it may be possible to specify a type pattern such as

type item is (<>);

Here the parantheses indicates that items is a discrete type, and will thus support such operations as comparison for ordering (<, >, etc.) and the attributes first and last.

In more complex cases, the Ada programmer can specify the operations of a generic type parameter by means of a trailing '*with*' clause

Without the '*with*' clause, procedure sort would be unable to compare elements of A for ordering, because type T is private.

Generic sorting routine in Java**:-**

```
public static < T extends Comparable <T>> void sort ( T A [ ])

    {.....

     if ( A [i].compareTO (A [j]) >=0) .....

      .....

     }

      .....

      Integer [ ] my Array = new Integer [50];

      .....

      sort (myArray);
```

In the C++ the code for sort makes use of the < operator.
For ints and doubles, this operator will do what one would expect.
For character strings, it will compare pointers, to see which referenced character has a lower address.

Implicit Instantiation:-Because a class is a type, one must generally create an instance of a generic class before the generic can be used.

The declaration provides a natural place to provide generic arguments:

```
queue<int, 50> *my_queue = new queue<int, 50>(); // C++ _
```

Some languages for eg:Ada, also require generic subroutines to be instantiated explicitly before they can be used:

```
procedure int_sort is new sort(integer, int_array, "<");
...

int_sort(my_array);
```

Generic class instance in C++:

queue <int, 50> *my_queue= new queue <int, 50 >( );

Other languages treat generic subroutines as a form of overloading.

int ints [10];
double reals [50];
string strings [30];

We can perform the following calls without instantiating anything explicitly:

Sort (ints, 10);
Sort (reals, 50);
Sort (strings, 30);

- In each case, the compilers will implicitly instantiate an appropriate version of the sort routine.
- The rules for implicit instantiation in C++ are more restrictive than the rules for resolving overloaded subroutines in general.

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.

The programmer has basically three options:

1. "Invent" a value that can be used by the caller when a real value could not be returned.
2. Return an explicit "status" value to the caller, who must inspect it after every call.
3. Rely on the caller to pass a closure for an error-handling routine that the normal routine can call when it runs into trouble.
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 was pioneered by PL/I, which includes an executable statement of the form

ON *condition*
statement

The nested statement (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.

In C++ we might write:

try {
...

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

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.

Common Lisp is also unusual in providing four different versions of its exception-handling mechanism.
Two of these provide the usual "exceptional return" semantics; the others are designed to repair the problem and restart evaluation of some dynamically enclosing expression.

Two perform their work in the referencing environment where the handler is declared; the others perform their work in the environment where the exception first arises.

Defining Exceptions: In Ada, some of the predefined exceptions can be *suppressed* by means of a pragma.

In Ada, exception is a built-in type; an exception is simply an object of this type:

```
declare empty_queue : exception;
```

Most languages use a throw or raise statement, embedded in an if statement, to raise an exception at run time.
In most languages, a block of code can have a *list* of exception handlers.
In C++:

```
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)
```

}
When an exception arises, the handlers are examined in order; control is transferred to the first one that *matches* the exception.
Handlers on Expressions: In an expression-oriented language such as ML or Common Lisp, an exception handler is attached to an expression, rather than to a statement.
A handler attached to an expression must provide a value for the expression.
In ML, a handler looks like this:

```
val foo = (f(a) * b) handle Overflow => max_int;
```

Cleanup Operations: The exception-handling mechanism must "unwind" the run-time stack by reclaiming the stack frames of any subroutines from which the exception escapes.
Code in Modula-3 might look like this:

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

A FINALLY clause will be executed whenever control escapes from the protected block.
The escape may be due to normal completion, an exit from a loop, a return from the current subroutine etc..

Implementation of Exceptions: The handler begins by checking to see if it matches the exception that occurred; if not, it simply reraises it:

```
if exception matches duplicate in set
. . .
else
reraise exception
```

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

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

Exception Handling without Exceptions: Scheme, provides a general-purpose function called call-with-current-continuation, sometimes abbreviated call/cc.

This function takes a single argument *f* , which is itself a function.

It calls *f* , passing as argument a continuation *c* (a closure) that captures the current program counter and referencing environment.

Most versions of C provide a pair of library routines entitled setjmp and longjmp.
Setjmp has an integer return type: zero indicates “normal” return; nonzero indicates “return” from a longjmp.
The usual programming idiom looks like this:

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

When initially called, setjmp returns a 0, and control enters the protected code.
Exception handling in C++: To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*.

An exception is thrown by using the throw keyword from inside the try block.

Exception handlers are declared with the keyword catch, which must be placed immediately after the try block.

The code under exception handling is enclosed in a try block.

In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter which is passed as an argument to the exception handler.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is.

This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last.

It is also possible to nest try-catch blocks within more external try blocks.

Exception handling in Java: To understand how exception handling works in Java, we need to understand the three categories of exceptions:

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer.

- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer.

- Runtime exceptions are ignored at the time of compilation.

- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.

- Errors are typically ignored in the code because we can rarely do anything about an error.

Catching Exceptions: A method catches an exception using a combination of the **try** and **catch** keywords.

A try/catch block is placed around the code that might generate an exception.

Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  //Protected code
}catch(ExceptionName e1)
{
  //Catch block

}
```

A catch statement involves declaring the type of exception we are trying to catch.

If an exception occurs in protected code, the catch block that follows the try is checked.

If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Multiple catch Blocks: A try block can be followed by multiple catch blocks.

The syntax for multiple catch blocks looks like the following:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
```

```
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block

}
```

The previous statements demonstrate three catch blocks, but we can have any number of them after a single try.

If an exception occurs in the protected code, the exception is thrown to the first catch block in the list.

If the data type of the exception thrown matches ExceptionType1, it gets caught there.

If not, the exception passes down to the second catch statement.

This continues until the exception either is caught or falls through all catches.

Coroutines: A coroutine is represented by a closure, into which we can jump by means of a nonlocal goto, in this case a special operation known as transfer.
The principal difference between the abstractions is that a continuation is a constant—it does not change once created—while a coroutine changes every time it runs.
When we goto a continuation, our old program counter is lost.
When we transfer from one coroutine to another, our old program counter is saved.
If we perform a goto into the same continuation multiple times, each jump will start at precisely the same location.
Imagine that we are writing a “screen-saver” program, which paints a mostly black picture on the screen of an inactive workstation, and which keeps the picture moving, to avoid phosphor or liquid-crystal “burn-in.”
We could write our program as follows:

loop
– – update picture on screen
– – perform next sanity check
To break it into pieces that can be interleaved with the screen updates, the programmer must follow each check with code that saves the state of the nested computation, and must precede the following check with code that restores that state.
A much more attractive approach is to cast the operations as coroutines:

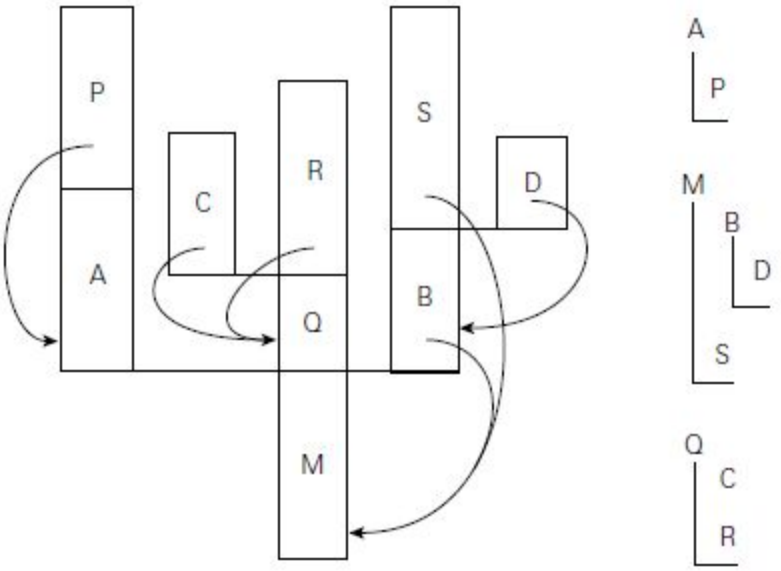
```
us, cfs : coroutine

coroutine check_file_system
    -- initialize
    detach
    for all files
        ...
        transfer(us)
        ...
            transfer(us)
        ...
        transfer(us)
        ...
```

```
coroutine update_screen
    -- initialize
    detach
    loop
        ...
        transfer(cfs)
        ...

begin       -- main
    us := new update_screen
    cfs := new check_file_system
    transfer(us)
```

The syntax here is based loosely on that of Simula.
When first created, a coroutine performs any necessary initialization operations, and then detaches itself from the main program.
A coroutine can also call subroutines, just as the main program can, and calls to transfer may appear inside these routines.
Stack Allocation: Most operating systems make it easy to allocate one stack, and to increase its portion of the virtual address space as necessary during execution.
An intermediate option is to allocate the stack in large, fixed-size "chunks."
At each call, the subroutine calling sequence checks to see whether there is sufficient space in the current chunk to hold the frame of the called routine.
At each subroutine return, the epilogue code checks to see whether the current frame is the last one in its chunk.
If so, the chunk is returned to a "free chunk" pool.
Fig. shows 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.
To implement sharing, the run-time system must employ a so-called *cactus stack*.
Each branch off the stack contains the frames of a separate coroutine.
The *static* chain of the coroutine, however, extends down into the remainder of the cactus, through any lexically surrounding blocks.
"Returning" from the main block of a coroutine will generally terminate the program as a whole.



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.
If transfer saves its return address in the stack, then the PC will change automatically as a side effect of changing stacks.
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.
The data structure that represents a coroutine or thread is called a *context block*.
In Simula, after the coroutine completes any application-specific initialization, it performs a detach operation.
Detach sets up the coroutine stack to look like the frame of transfer, with a return address that points to the following statement.
Implementation of Iterators: Iterators are almost trivial: one coroutine is used to represent the main program; a second is used to represent the iterator.
Additional coroutines may be needed if iterators nest.
Discrete Event Simulation
A *discrete event* simulation is one in which the model is naturally expressed in terms of events that happen at specific times.
Discrete event simulation is usually not appropriate for the continuous processes unless these processes are captured at the level of individual particles.