Spandan G
Asst prof|dept, CSE
Gitam university

Module - III

chapter-1

Subroutines and control abstraction

Abstraction - a process by which the programmer can associate a name with a program fragment which can then be thought of in terms of its purpose or function, rather than in terms of it's implementation.

Control abstraction - purpose is to perform a well-defined operation.

Data abstraction - purpose is to represent information

- Subroutines are the principal mechanism for control abstraction.

1. Review of stack layout

- Each subroutine is placed in a stack
- stack frame (or activation record)
- stack pointer
- Frame pointer.
- static chain

In a language with nested subroutines & static scoping (pascal, ada, ML, common Lisp, scheme), objects that lie in the surrounding subroutines & that are ~~neither~~ neither local or global, can be found by maintaining a static chain

- Static link

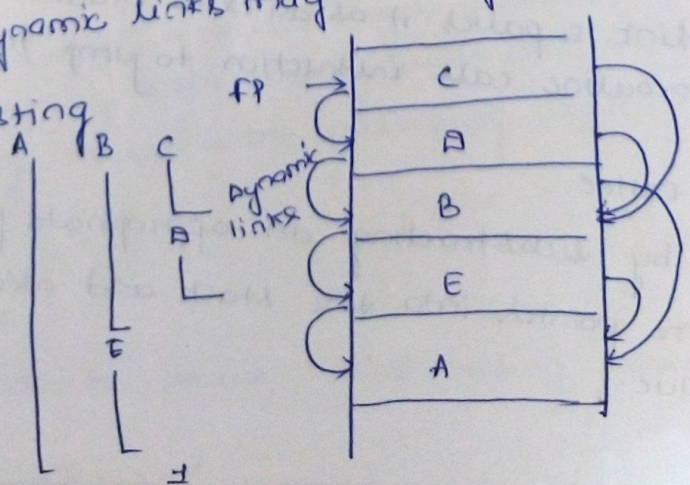
The frame of the lexically surrounding subroutine. This reference is called the static link.

- Dynamic 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)

Eg: subroutine nesting



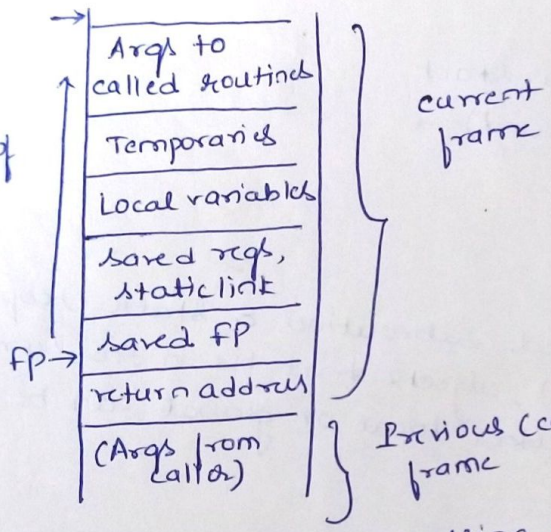
2. Calling Sequences:

→ Responsibility of the calling sequence is to maintain the subroutine call stack.

→ The code executed by the caller immediately before and after a sub-routine call, prologue (code executed at the beginning) and epilogue (code executed at the end) of the subroutines itself.

→ Saving & Restoring registers
- caller saves all registers that ax is use.
- calle to save all registers that it will overwrite.

A typical calling sequence



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

The caller

1. saves any caller-saves register whose values will be needed after the call.
2. computes the values of the arguements & moves them into the stack or registers.
3. Computes the static link & pases it as an extra hidden arguement.
4. uses a special subroutine call instruction to jump to the subroutine.

In its prologue, the callee

1. Allocates a frame by substracting an appropriate from the sp.
2. saves the old frame pointer into the stack and assigns it an appropriate new value.

3. saves any callee-saved registers that may be overcoritten by the current subroutine.

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 & sp.
4. jumps back to the return address.

Finally, the caller

1. Moves the return value to wherever it is needed.
2. Restores caller-saves register if needed.

(i) Displays:

→ One of the disadvantage of static chain is that, accel to an object in a scope k level out required that the static chain be dereferenced k times.

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

→ This number can be reduced to a constant by use of display.

→ Display is a small array that replaces the static chain.

→ For most programs the cost of maintaining a display in the subroutine callings tends to be higher than that of static chain.

ii) Register windows:

- it is an alternative to saving & restoring on subroutine calls & returns.
- it is a hardware mechanism known as a register windows
- The basic idea is to map the limited set of register names onto the some subset (window) of a much larger collection of physical registers & to change the mapping when making subroutine calls.
- old & new mapping overlap a bit, allowing arguments to passed in the Intersection.

iii) In-line Expansion:

- it is an alternative to stack based calling conventions.

- it means a copy of the "called" routines becomes a part of the "caller", no actual subroutine call occurs.
- it avoids many overheads like
  - Space allocation
  - branch delays from the call & return.
  - Maintaining the static chain or display
  - Saving & restoring Registers.
- In C++, the keyword inline can be prefixed to a function declaration.

3. Parameter Passing.

- Most subroutines are parameterized
- variables & expressions that are passed to a subroutine in a particular call are known as actual parameters.
- parameter names that appear in the declaration of a subroutine are known as formal parameters.
- In most languages, function name is prefixed followed by a list of arguments

  Eg; Lisp (max a b)

  Lisp places the function name inside the parantheses.
- user defined subroutines in Lisp & Smalltalk use the same style of syntax as built in operators

Pascal: if a>b then max = a else max = b

Lisp: (if (> a b) (setf max a) (setf max b))

SmallTalk (a>b) if true [max ← a] if False [max ← b]

1) Parameter modes:

Suppose x is a global variable & p(x) is a subroutine call.

Two methods

- call by value (a copy of 'x' is value is provided to p)
- call by Reference (address of x is provided)

- With call-by-value, actual & formal parameters are independent
- With Reference parameters, within the body of the subroutine, there will be a new name for corresponding actual parameter.

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

- if y is passed to foo by value then the program prints 2 twice
- if y is passed to foo by Reference then the program prints 3 twice

a) 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, a mode first introduced in algol w
- This method copies the actual parameter into the formal parameter at the beginning of the subroutine execution. it copies the formal parameter back into the actual parameter when the subroutine returns

According to this, the program above would print 2 & then 3

Pascal - call by value (default)

c - parameters are always passed by value and pointers

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

b) call-by-sharing: - In either case, instead of making a copy the variable or passing its address, the variable itself is
In case
- Languages like smalltalk, Lisp, ML, or clu in which the variables already a reference. Here pass by reference is more natural.

- clu call this mode call-by-sharing.
- it is different from call by value
[The program can see the changes made to object by formal parameter through the actual parameter after the subroutine returns]
- it is 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 the object]
- It is more natural simply to pass the reference itself & let the actual & formal parameters refer to the same object

Eg: java use call by sharing for user-defined data types

c) Read-only Parameters

- Modula-3 provide READONLY parameter mode to combine the efficiency of reference parameters & safety of value parameter.
- Any Formal parameter whose declaration is preceded by Readonly cannot be changed by the called routine.
- Small readonly parameter are generally implement by passing a value
- Larger readonly parameter —"— —"— by passing an address.

Eg: In C the equivalent of Readonly parameter is using const before formal parameter

```
void append_to_log(const huge_record *r)
{
---
}
append_to_log(&myrecord);   → callee cannot change the record's contents
```

d) Parameter mode in Ada:

- Ada provide three parameter-passing mode called in, out & in out
- In parameter - pass information from the caller to the callee, they can be read by the callee but not written
- out parameter - pass information from the callee to the caller
- Inout parameter - pass information in both direction they can be both read & written.
- changes to out or in out parameter always change the actual parameter.

e) Reference in C++:

→ C++ introduced an explicit notion of Reference

- Reference parameter are specified by preceding their name with an ampersand in the Header of the function

Eg:
```
void swap(int &a, int &b)
{
int t = a;
a = b;
b = t;
}
```

- C++ has the concept of reference variables

2) Call-by-Name

- In general, a language implementation must pass a closure consisting of the code of the parameter required the retrieval of a previous referencing environment.

Examples

- call by name parameters of algol 60 & simula.

3) Special-purpose parameters:

i) Conformant Arrays.

- A formal array parameter whose shape is finalized at run-time is called a "conformant or open array parameter"

Eg;

```
void apply_to_A (int (*f)(int), int A[], int A_size)
{
    int i;
    for (i=0; i<A_size; i++)
        A[i] = f(A[i])
}
```

A[] - open array parameter

f(n) means - f is the name of a function or f is a pointer to a subroutine

ii) Default (Optional) Parameters:

- A default parameter is one that need not necessarily be provided by the caller, if it is missing then a predestablished default value will be used instead.

Eg: In I/O library routine 'put' routine for integers

```
default_width: field = integer'width;
default_base : number_base = 10;
procedure Put (width : in field := default_width;
               base  : in number_base := default_base);
```

iii) Named parameters

- usually parameters are positional i.e. 1st actual parameter corresponds to the 1st formal parameter, 2nd actual parameter to the 2nd formal & so on...

- In some languages, this need not be the case, they allow parameters to be named called keyword parameter.

Eg:

put (base => 8, item => 37)
named

Ada

iv) variable Number of Arguements:

- Lisp, python and c allow subroutines to take a variable number of arguements.

Eg: printf & scanf functions

```
int printf (char *format, ...)
{
  ---
  ---
}
```

... - indicate that there are additional parameter.

4) Function Returns:

- In Languages like algol 60, Fotran, pascal, the Function's return value appears in the an assignment statement (LHS is the name of the function)

- Most Languages allow return statement or ~~return expression~~

return expression

- In addition to specifying a value, return causes the immediate termination of the subroutine.

Eg:

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

4. Generic SubRoutine and Modules:

- A subroutine performs an operation for a variety of different object (parameter) values

- Large programs may require to perform an operation for a variety of different object types.

Eg; OS makes heavy use of queue to Hold.

- Generic modules (class) are particulary valuable for creating containers (data abstraction) that hold a collection of objects

- generic subroutine (methods) are needed in generic module (class)

Implementation options

- In Ada, C++, it is a static mechanism
- All the work required to create & use multiple instances of the generic code takes places at compile time

Function Template

Syntax

```
template<class T>
T someFunction(T arg)
{
-------
-------
-------
}
```

class Template

Syntax

```
template<class T>
class Large(T1, T2)
& class className
{
....
....
Public T var;
T someoperation(T arg);
};
```

```
#include<iostream.h>
#include<string.h>
template<class T>
class Tempclass
{
Public T value;
Public Tempclass(T item)
    {
    value = item;
    }
T getvalue()
  {
  return value;
  }
};
```

```
int main()
{
Tempclass<string>*string = new Tempclass
                          <string>("Generics");
cout<<"output values"<<string->
                          getvalue();
Tempclass<int>*Int = new Tempclass
                          <int>(6);
cout<<"output values"<<Int->getvalue();
}
```

- The compiler created a separate copy of a code for every instance.
- In java 5, by contrast gaurantell that all instances of a given generic coill share the same code at run-time

Generic parametric constraints:

- In ada portion

  type item is private

- A private type is which the only permissional operations are assignment, testing for equality & inequality etc

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

type item is (<>);

() - item is an discrete type;

=> - supported operations

Eg:

```
generic
    type T is privat
    type Tarray is array (integer range <>) of T;
```

-Java & C# employ particular clean approach to constraints & has a exploit ability of object-oriented types to inheritance method from a parent type or Interface

eg; Public static <T extends comparable <T> > void sort (T A[])

```
{

}
Integer[] myArray = new Integer [50];
sort(myArray);
```

Generic sorting in C++

```
template <class T>
void sort (T A[], int A_size)
{
- - - - - -
- - - - - -
- - - - - -
}
```

Implicit Instantion:

-In C++, because a class is a type, one must generally created instance of a general class before the generic can be used

Tempclass <int> = new Tempclass <int> (50);

-Implicit instiation in C++, languages like C++, Java, & C# does not require explicit Instiation

```
int int8[10];
double real8[50];
sort(int8,10);
sort(real8,50);
```

5. Excepting Handling:

Exception - an unexpected or unusual condition that arises during program execution & that cannot easily be handled in the local context.

Eg; various sorts of run-time errors

In c++, A simply try block

```
try
{
---
if (Something_unexpected)
   throw my exception();
...
cout << "Everything is ok\n";
}
catch (my-exception e)   // exception Handler
{
cout << "oops\n";
}
```

Exception for user-defined subroutine

```
try
{
foo();
...
cout << "everything is ok\n";
...
}
catch (my-exception)
{
cout << "oops\n";
}
void foo()
{
if (something-unexpected)
throw my-exception();
...
}
```

Exception Handler performs three kinds of operations

i) Handler compensates for the exception.

Ex: Exception - out of memory
Handler - makes a request to the os to allocate additional space.

ii) Exception cannot be handled in a local block of code.

iii) if recovery is not possible, a handler can at least print a helpful error message before the program terminates.

1) Defining Exceptions:

→ Predefined exceptions - arithmetic overflow, division by zero, end of file on input, subscript and subrange errors, null pointer dereference etc

→ Dynamic exceptions - returns from a subroutine that has not yet designated a return value.

- In C++, common, Lisp - all Exceptions are programmer defined.

2) Exception propagation:
A block of code can have a list of exception handlers

In C++

```
try
{
  ----  // many calls to I/O routines
}
catch (end_of_file)
{
----
}
catch (io_error e)
{
    // any io-error other than eof
}
catch(...)
{
    // for any exception not previously named
}
```

. catch (...) → 'catch all'

a) Handlers on Expressions:
- In ML, common, Lisp are expression oriented languages, exception handler is attached to an expression rather than to a statement

Eg;
val foo = (f(a) * b) handle overflow ⇒ max-int;

b) 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
remove

- In C++
when an exception leaves a scope, destructor are called for any objects declared within that scope.
- Destructor are used to deallocate heap space

c) Implementation of Exceptions

→ Implementation maintains a linked_list stack of handlers.
- when control enters a protected block, the handler for that block is added to the head of the list.
- when an exception arises (either implicitly or as a result of raise statement) the language run-time system pops the innermost handler off the list & call it.
- The Handler begins by checking to see if it matches the exception that occurs.
- if exception matches end-of-file
    ...
  elsif exception matches io_error
    ...
  else
    ... "catch_all" handler.
- if a protected block of code has handlers for several different exceptions, they are implemented as a single handler containing a multi if statement.

d) Exception Handling without Exceptions:
Exception can sometimes be simulated in a language that does not provide them as built in.
Pascal - permit go to labels outside the current subroutine
Algol-60 - allows labels to be passed as parameter.

6) Co-Routines:

- A coroutine is a closure (a code address & a referencing environment) in which one can jump by means of a non-local go to, in this case a special operation known as Transfer
- The continuation is a constant & it does not change once created where as coroutines change every time it runs
- In continuation the old program counter is lost, unless one create a new continuation to hold it
- In co-routine the program counter is saved, it is updated to reflect it
- Coroutines are execution contexts that exists concurrently, but that execute one at a time & that transfer control to each other explicitly by name
- Coroutines can be used to implement iterators, & threads, certain kind of servers, discrete event simulation

Eg: screen-saver program ~~to~~ which paints a mostly black picture on the screen of an workstation & which keeps the picture moving it also performs "sanity checks" on file system in the background looking for corrupted Files

```
loop
---update picture on screen
---perform next sanity check
```

- The problem with this approach is that successively sanity checks are likely to depend on each other

```
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)
```

2) Transfer
- To transfer from one coroutine to another, the run-time sys must change the program counter (PC), the stack & contents of registers
- All these are encapsulated in transfer operations.
- The usual approach to change a stack pointer, register & to avoid using the frame pointer inside a transfer itself

transfer:
push all registers other than sp
*current coroutine = sp
current_coroutine = r1
sp = *r1
pop all registers other than sp
return

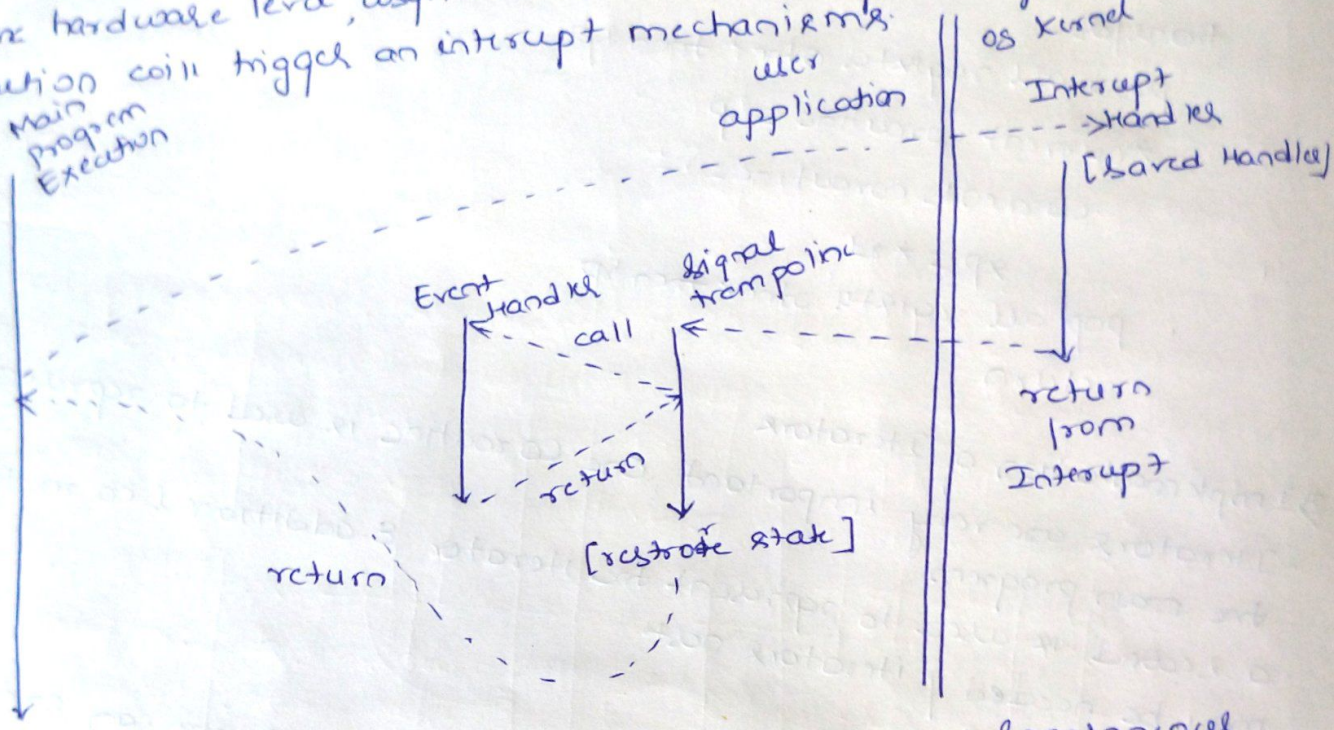
3) Implementation of Iterators
- Iterators are very important, one coroutine is used to represent the main program
- a second is used to represent the iterator & additional coroutines may be needed if iterators nest

4) Discrete event simulation
- simulation in general refers to any process in which one can create an abstract model of real world sim.
- A discrete event simulation is one which the model is naturally expressed in terms of events that happen at specific time

7. Events
- An event is something to which a running programs (a process) needs to respond but which occurs outside the program at an unpredictable time
- The most common events are inputs to the GUI systems are keystrokes mouse motion, button clicks
- They may also be network operations or other asynchronous I/O activity

a) Sequential Handlers

- Traditionally event handlers are implemented in sequential programming languages as subroutine calls, typically using a mechanism defined & implemented by the O/S system.
- To prepare to receive events through this mechanism, a program call it P-invoked a setup-handler library routine passing as argument
- At the hardware level, asynchronous device activity during the P's execution will trigger an interrupt mechanisms



- The Figure Illustrated the typical implementation of spontaneous subroutine calls as used by the unix signal mechanism
- The Langanguage run-time library contains a block of code known as the signal trampoline
- The signal trampoline create a frame itself in the stack & then the return event handler using the normal subroutine calling sequence.
- When the event Handler returns, the trampoline restores its state from the buffer written by the Kernel & jumps to the main program
- To avoid recursive events, the kernels typically disable further signals when it jumps to the signal tramapoline

b) Thread-Based Handlers:

- In modern programming languages & run-time s/m, events are often Handled by a separate thread of control

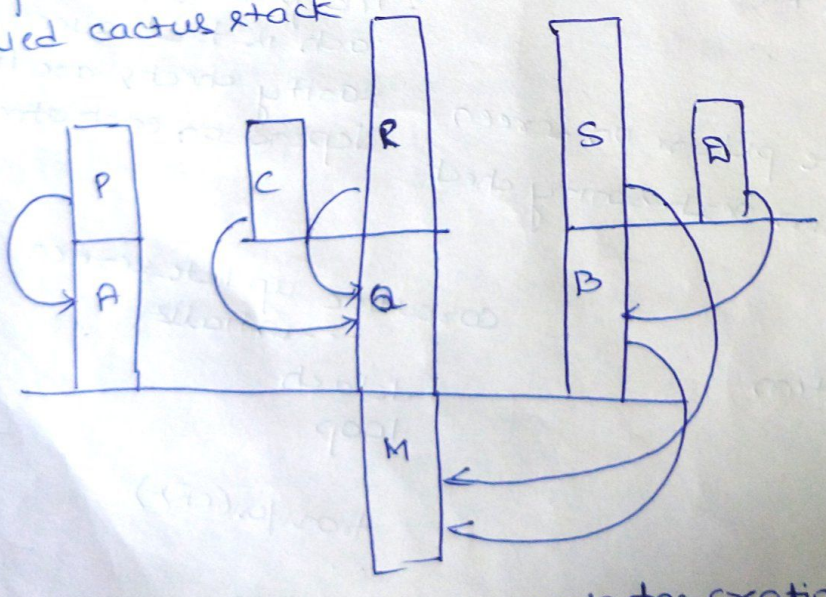
Eg; Java Awings Library, openGL, C#

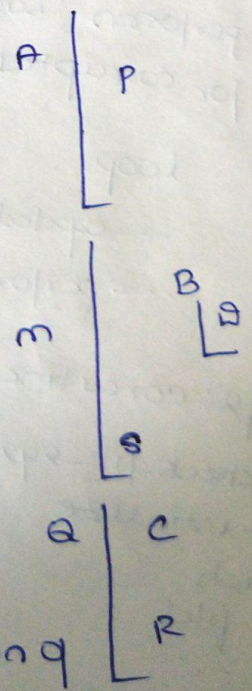
The syntax is based loosely on that of Simula

- when first created, a co-routine performs any necessary initialization operations & detaches from the main program.
- The detach operation creates a co-routine object to which control can later be transferred & returns a reference.
- The transfer operation saves the current program counter in the current coroutine object & returns the co-routine specified as a parameter

1) Stack allocation:

- The co-routines are concurrent, they cannot share a single stack
- The simple solution is to give co-routine a fixed amount of statically allocated stack-space.
- This approach is adopted in modula-2, in which the programmer needs to specify the size & location of a stack
- Some implementation catch the overflow & halt with the error message.
- If co-routine can be created at arbitary levels of lexical nesting then two or more co-routines may be declared in the nonglobal scope & must share accl to object
- To implement this sharing, the run-time sys must employ is called cactus stack





- Each branch to the side represents the creation of a co-routine (A, B, C, D)
- Static links are shown with arrows
- Dynamic links are indicated by vertical arranged
- Static nesting of block is shown at right

An Event handler in C#

```
void Paused (object sender, Event args a)
  {
  . . .          // do whatever needs doing when the pause
  . . .             button is pulled
  }

Button pB = new Button("pause");
pb.clicked += new EventHandld (Paused);
```

An Event handler in java

```
class PauseListener implements ActionListener
  {
  Public void actionPerformed (ActionEvent e)
    {
    - - - -
    - - - -
    }
  }
JButton PB = new JButtion ("pause");
pb.addAction listenr (new Pauselistened ());
```