- Subroutine, subprogram, procedure, function, coroutine -- many related terms
- Formal parameters Vs Actual parameters (aka arguments)
- Primary control abstraction, subroutine performs computation for the "caller"
- Most subroutines, parameter and local variables are stored on a stack
- Review of generic stack layout
  - Often starts at large addresses, grows towards smaller addresses
  - Stack pointer (top of stack), Frame pointer (current subprogram call frame)
  - Access to variables typically +/- from Frame pointer
  - Depending on implementation and language: has one of static links, displays, dynamic links
  - Space in frame for saving registers, restored on function return
  - Typical stack frame (depending on architecture)
    - arguments from caller, return address, frame pointer, saved registers and static link,
    - local variables, dynamic locals, temporaries, arguments to called routines.

Call sequence:

- Caller:
  - saves any registers/temps that may be overwritten and still needed
  - computes value, address, or access method for formal parameters
    - push on stack or put in registers
  - push static link, or store to a register, push "hidden" args.
  - calls the subroutine using a machine "call" instruction
- Callee:
  - Runs a "preamble" or "prologue" -- code to run before user code
    - Allocates the frame, saves old frame pointer, sets frame pointer to current frame
    - Saves any other things/registers that may be needed after return (e.g static link)
    - (Not listed in book) Does parameter work done by callee, e.g. copy a call by value array
    - (Not listed in book) Dynamic stack storage allocation, e.g. int ary[N] where N is a parameter
  - At return, runs a "postamble" or "epilogue"
    - Moves return value (if any) to proper place
    - restores saved things/registers
    - restores fp and sp
    - returns to the return address
- Caller:
  - moves return value if needed, restores saved values
- Optimizations: leaf routines may not need full frame, ret addr in a register, leave it there, ...

Subroutines (page 2)

- Display: We talked about a display before, book talks about it here, review it.
- Berkeley RISC machine: register windows, e.g. stack frame in registers
- In-line expansion:
  - subroutine does not generate code
  - at point of call, code is generated with subroutine as the "template"
  - can request it in some languages
  - getters/setters in object oriented languages can be done in-line, reduces call overhead
  - can act like a macro, semantics may dictate how parameters are evaluated
  - can inline a tail-recursive function -- others may be problematic
- Parameter passing:
  - Calling methods: How to pass the parameter ...
  - Calling syntax:
    - most languages use prefix: name(parameters)
    - ML can be infix: arg1 fname arg2 (see book for details)
  - Call by value
    - P(expr) -- evaluate expr in caller, send a single value to callee
    - P(x) with x as a global won’t produce an alias
    - defn: void P(int val)
      - val acts like an initialized local variable
        - val := 3; // does not effect x
      - val acts like a constant value (void P(const int val) in C, other languages support this)
        - val := 3; // illegal

Parameters (page 2)

- Call by reference
  - P(expr) -- only if expr can compute an l-value, pass the address
  - P(x) with x as a global often creates an alias
  - defn: void P(int &val) // C++
    - val := 3; // changes actual parameter ... e.g. x
  - Why a reference parameter?
    - change actual parameter (e.g. a return value!)
    - avoid copying a huge structure (how about a "readonly" parameter)
- Call by sharing
  - Languages where a variable is a reference, you share the reference (Smalltalk, Lisp, ML, Ruby...)
  - Reference is copied to the callee
  - May not be able to set the actual parameter
- Value/result
  - P(expr) -- only if expr can compute an l-value, pass the address (and the value)
  - caller or callee computes the value, stores value in local variable
  - code uses local variable, no changes to actual parameter
  - epilogue/postamble: use passed l-value to store local variable in actual parameter
- Languages define standard way of passing parameters
  - Pascal (and similar) -- default is by value, keyword (var) causes by reference
  - C: everything is by value! But an array name produces a pointer to the array, pass the pointer
    - Simulate by reference by passing pointers by value ... programmer has to do it.

Parameters (page 3)

- Ada parameter modes:
  - in parameters: read by callee, not written
  - out parameters: can not be read by callee, written to send data to callee
  - in out parameters: value/result parameters
  - Spec: for scalar types, copy values
    - for constructed types ... can use value or reference
    - how to tell the difference between reference and value/result ?
- Aliases in C++
  - int i; int &j = i; i = 2; j = 3; cout << i; -- prints?
- Returning a reference in C++
  - cout << a << b << c; // output in C++ .... but << is an overloaded operator!
  - ((cout.operator<<(a)).operator<<(b)).operator<<(c); // qualified version
  - Allows a return of a dangling reference, similar to returning pointers
- R-value reference in C++11 -- read about them
- Closures as parameters (aka procedure/function parameters)
  - routine in functional languages
  - C: typically just a pointer
  - Swift/ObjectiveC: delegate -- myWindow.delegate = MyDelegate() {...}
    - Used extensively in iOS applications, part of a "Delegate Protocol"
  - C# -- delegate may have a list of closures, each one called at invocation time
- Call by Name -- Algol 60 feature
  - Pass two closures: (thunks) ... r-value thunk and an l-value thunk.

Parameters (page 4)

- Default parameters
  - Just constants?
  - Evaluated in which reference environment?
- Syntax -- allowing to name parameters in call: Put(37, Base=>8) -- ada
  - self documenting?
  - Swift does this
- Variable number of arguments
  - C -- stdarg.h
  - supported in Java: static void printLines(String foo, String... lines) ...
    - lines.length (looks like an array)
- Function returns
  - variety of syntaxes
    - Algol 60, Fortran and Pascal: func_name := expr;
    - many others: return expr;
    - go: func A_max(A []inti) (rtn int) { .... rtn = A[0].... return }
  - return of tuple (ML, Chapel, ...)
  - return of a structure (C, [deep copy] vs Java return of class [shallow copy])

Exceptions

- Errors in subprograms -- how to deal with them
  - - specific return value
  - - return a status and a value
  - - pass in a closure to call in the error case
  - None are very good ... C does setjmp/longjmp (not really one of these)
- Enter the exception
  - PL/I had an exception structure ... "on condition statement"
    - statement was executed and then program terminated or return to statement after one causing exception
    - caused confusion, error prone
  - C++ and others
    - try { ...} catch (error e) {...}
    - raise or throw the exception
    - if not caught, function returns and throws the error in the caller
  - Result
    - fix the error, retry
    - cleanup and rethrow
    - terminate the program
  - Lisp/Ruby have two concepts:
    - normal multilevel return
    - raising an exception, not a normal multilevel return
  - Varitey of issues ... how to define, parameters to exceptions?, how to clean up, propagation
  - Python’s finally clause. runs when control escapes from the try block by what ever method

Exceptions (page 2)

- Exceptions in languages without exception processing
  - C -- often uses goto error label. (inside a function)
  - C -- setjump/longjump
  - Scheme, Ruby, can be done with a continuation ...

Coroutines -- support for limited threading

- switch back and forth between two (or more) functions
- needs activation records to be in the heap rather than on the stack
  - often appears as a separate stack for each coroutine (see example 9.51 cactus stack)
- coroutines are still strictly sequential operations.
- typically there would be a "transfer" call to cause the other routine to run
- initial threads were done using coroutines
- full threading with concurrent execution added later
- (not in book) Path Pascal added "processes" and full process synchronization in the language

Events

- Events -- another abstraction with subroutine calls
  - C/UNIX -- signals -- OS generated events
    - signal handler -- causes what appears to be a function call "out of order"
    - Something called a "signal trampline" -- OS builds a call frame on the stack ...
  - Threading and signals -- single thread responds to signals
  - Mobile/GUI based systems -- something similar
    - An event thread (GUI thread)
    - Events cause a "delegate" to be called

    goButton.Clicked += delegate(object sender, EventsArgs a) { ... code ... } // C#
  - Primary abstraction .. call a method/function when event happens
  - Events -- often OS generated (e.g. windowing system)
    - Possible for one program to send an event to another program
    - Unix can have one process send a signal to another process
- Read chapter 9.7 Asynchronous Programming -- taking event programming to another level