

## Subroutine Basics

- Subroutines constitute a form of *control abstraction* — the ability to package related or common sets of activities into a single, callable entity
- We'd like our subroutines to have:
  - self-contained environments for temporary values (e.g. locals) that go away after we're done
  - the ability to call other subroutines while maintaining the calling order, including themselves (recursion)
  - the ability to receive arguments/parameters to customize the subroutines' behavior
- Generally, this is all done by maintaining a stack of *frames* or *activation records* representing the subroutines that we have called so far

## Nested Subroutines

- Subroutines-within-subroutines: not an issue for C/C++/Java
  - Completely absent from C/C++
  - While one can embed subroutine-like code inside a Java subroutine, this code is built within Java's objects, and so is not quite the same as how subroutines can be nested into Perl, JavaScript, Pascal, Modula, and Ada
- Nested subroutines require both dynamic (runtime) and static (lexical) chains
  - Parallel *static chain* that tracks a subroutine's *lexical* nesting as opposed to the runtime calling sequence
  - Lexical nesting is stored using a *display* — a conversion of the static chain into an array
  - Display size for each subroutine can be fixed at compile time, because lexical nesting is known then

## Calling Sequences

- The *calling sequence* refers to the activity surrounding the actual subroutine invocation
  - Code performed by the subroutine's caller before and after the call
  - The *prologue* and *epilogue* of the subroutine — the code that executes at the beginning and end of the subroutine, respectively
- Typical calling sequence tasks
  - Before
    - Passing parameters
    - Saving the return address and registers
    - Setting the program counter and stack pointer
    - Performing any initialization
  - After
    - Passing return values or parameters
    - Assorted deallocations
    - Restoration of previously saved values

## Implementation Issues

- Prefer to put as much of the calling sequence as possible within the callee (subroutine) — that way the code is stored only once. Calling sequence code that is part of the caller is copied at each subroutine call
- Notice how subroutine calls are very low-level; thus they are very dependent on the architecture on which they are implemented
- Many processors have built-in conventions for subroutine calls; compilers are best off following these conventions when targeting these processors
- Many more permutations and cases to consider for nested subroutines — so it's no wonder that C's designers skipped those entirely!

## In-Line Subroutines

- Alternative to stack-based view of subroutines
- “In-line” expands (copies) the subroutine code directly at the point at which it is called
  - Avoids a lot of the overhead involved with “conventional” subroutines
  - But increases code size (the usual time vs. space tradeoff)
- In-lining can be the compiler’s choice (implicit) or requested by the user (C++, Ada) — still, the compiler gets the final say
- Different from macros — macros are straight text replacements, while in-lined subroutines maintain the exact same semantics
  - `#define MAX(a, b) ((a) > (b) ? (a) : (b))` expands `MAX(x++, y++)` into `((x++) > (y++) ? (x++) : (y++))`
- Slight trickiness with recursive subroutines: usually in-line only the first level, then use conventional calling for succeeding levels