Subroutines

- 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 in the end
  - 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
  - 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 — this is precisely what is meant by the term "calling sequence"
- The *prologue* and *epilogue* of the subroutine — the code that executes at the beginning and end of the subroutine, respectively — not strictly part of the calling sequence, but the term is sometimes extended to include these too

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, frequently implemented directly in the processor (stack pointers, jump/return instructions); thus their specifics can be 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

Parameter Passing

- Not completely essential — you can always use globals — but these days it's pretty much taken for granted
  - Increases the level of abstraction
- *Formal* parameters stand for the parameters as they are declared by the subroutine
- *Actual* parameters represent the specific values and/or expressions sent to a subroutine on a particular call
  - "Arguments" is sometimes used as a synonym for actual parameters
- Subroutines with parameters are typically expressed in prefix notation, with some infix exceptions

Parameter Modes

- Call by value
  - Arguments are copied into the subroutine, and thus do not affect the original value
  - Watch out for large objects — sometimes will end up doing call by reference due purely to size issues
- Call by reference
  - Arguments are stored as addresses to their original values; thus everything that the subroutine does to them "sticks" after the subroutine returns to the caller
  - To provide "call by value" semantics when copying is not practical, read-only or const tags are sometimes provided
- Languages may provide just one or both modes
  - C, Fortran, ML, Lisp: either all-value or all-reference (or the sharing variety)
  - Pascal, Modula, Ada: provides either
  - Java: mode is based on "primitive" vs. "object" parameters

Parameter Mode Issues

Main issue with call-by-value: what to do with composite types (records, arrays, etc.)

- Composite types can be quite large, requiring a lot of stack space when passed by value
- Older languages had no way around this — in Pascal, arguments were sometimes passed by reference to save on memory even if the semantics was really call-by-value
  - Can lead to bugs, since your code no longer accurately reflects your intent

To have your cake and eat it too: we want the efficiency of call-by-reference with the safety of call-by-value

- Add a new keyword that specifies an argument as read-only (Modula-3: READONLY; ANSI C: const)
- This changes the core issue: are we concerned about how a parameter is passed, or just whether or not it can be changed?

Parameter Mode Variations

- Pascal: explicit specification of value vs. reference via the *var* keyword
- C: always pass by value; pass by reference is "simulated" by passing pointers — a bit of a cheat, because then the subroutine's argument is not the base type, but a *pointer* to the type
- Fortran: always pass by reference, creating temporary variables for parameter expressions

- Java: mode is based on parameter type — primitives pass by value and objects pass by sharing (not quite pass by reference, since you can't reassign an object parameter)
- Ada: express parameter modes in terms of readability or writability: *in*, *out*, and *in out*
- C++: adds true references to C via the & symbol

Closures as Parameters

- Recall that a *closure* is a reference to a subroutine, along with its referencing environment (remember static vs. deep binding?)
- Many languages give subroutines their own types — thus closures-as-parameters are completely orthogonal (ML, Modula-2, Modula-3)
- C/C++ use pointers, which can be interpreted as types, but they don't mask that they are still pointers
- Java uses *reflection*: can't pass a subroutine directly; you need to get a Method object for that subroutine

Call-by-Name Parameters

- Introduced in Algol 68: mimics macro-like behavior in parameters
  - Re-evaluates a parameter's expression every time that parameter is accessed
- Implemented as a hidden mini-subroutine that evaluates the expression — called a *thunk*
- Clever use: *Jensen's device* — call-by-name parameters that are related to each other, such that one parameter influences the evaluation of another parameter
  - See Scott page 452

More Parameter Tricks

- *Label parameters* — ack! Allows *goto* labels to be passed to subroutines…superceded by exceptions
- *Conformant arrays*: For flexibility, many languages do not specify array sizes in their parameters, to allow processing of multiple array shapes of the same type
- *Default/optional parameters*: Some languages allow the specification of default parameters, so that the caller can skip some of them

- *Named parameters*: Instead of specifying parameters by position, specify them explicitly by name
- *Variable argument lengths*: Allow functions to accept a flexible number of arguments (classic example: C's *printf*; also, Perl's arguments-as-array approach)

Return Values

- Initially, only scalars can be returned; these days, it can be pretty much anything, including a closure
- Variations in syntax: depends on whether a statement is also an expression in the language
  - Return value can be "the last expression evaluated" — ML, Perl
  - Return value must be explicit, e.g., through a *return* statement — C, Java, many more
  - Older variation: assign the return value to the subroutine name within the subroutine body (Pascal, Fortran)
  - Rarer variation: predefined variable in the function that holds the return value (SR, Eiffel) — saves overhead of having to allocate yet another local variable to hold the result

Generic Subroutines

- Some subroutines perform valid sequences of operations regardless of the types of values involved (collection management, print statements)
  - *Overloading*: allow more than one subroutine of the same name; limited use, requires different implementations for each overloaded version
  - *Polymorphism*: allow subroutines to act upon unspecified types (*'a* in ML, *void* * in C/C++, *Object* in Java); incurs run-time overhead
- Generic subroutines allow subroutines to handle different types using the same source code: C++ (templates), Ada, Java > 1.5
  - Same source code, but different compiled code, results in benefits of polymorphism without the overhead

Exceptions

- Exceptions are unexpected or unusual conditions that may arise during execution
  - I/O errors, division by zero, formatting/parsing errors — any errors that are possible, but cannot be detected at compile time
- Older ways to handle these cases:
  - Return an invalid value (–1?)
  - Set or return a status value (success/failure)
  - Pass a closure as an error handler
  - Each of these mechanisms is sufficient in certain contexts, but none are completely general
- The general solution: exceptions

- Approaches to exceptions:
  - Initial version by PL/I: "*on condition*" statement — doesn't execute the statement, but "remembers" to execute it when the given condition becomes true (e.g., OVERFLOW)
  - Newer approach: "lexically bound" exception handling — *try*/*catch*-style: Clu, Ada, Modula-3, C++, Java, ML
- Exceptions can *propagate* — if a handler is not specified within a subroutine, then the entire subroutine terminates, "passing on" the exception
- Exceptions can be first class values — data types in their own right, and manipulated as such
- Variations:
  - Parameterized exceptions: allow further specification or context of an error condition
  - Exceptions may be "thrown" or "raised" at will
  - Java has "checked" vs. "unchecked" exceptions — checked exceptions are statically enforced, but unchecked will only be caught dynamically