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 that are sent to a subroutine on a particular call
  – Scott likes to use *arguments* 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

• Call by sharing
  – Used by languages where variables are already references to objects
  – Parameters are references to objects, but assignments to those parameters don’t change them at the level of the caller — e.g. Java uses call-by-value for primitives (int, char) and uses call-by-sharing for Objects
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 when 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 as well, although they don’t mask the notion that they are pointers

• Java uses reflection: you can’t pass a subroutine directly; you need to retrieve a Method object that represents that subroutine, then invoke it

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)

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

- Many approaches
  - 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: templates in C++, Ada
  - 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 are sufficient in certain contexts, but none are completely general

- The general solution: exceptions
Approaches to Exceptions

• Initial version by PL/I: allows an “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-like constructs: Clu, Ada, Modula-3, C++, Java, ML

• Exceptions can propagate — if a handler is not specified within a subroutine, then the entire subroutine terminates, with a marker for the exception that occurred

• Exceptions can be first class values — they are data types in their own right, and can be manipulated as such

Variations on Exceptions

• 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