

Parameter Passing

- In / In-Out / Out logical modes
- Call by value
- Call by reference
- Call by value-return
- Call by name
- Example

Intro

- Logically parameter-passing has following possible semantics:
 - IN: pass info from caller to callee
 - OUT: callee writes a value in the caller
 - IN/OUT: caller tells callee value of var, which may be updated by callee
- However, different mechanisms of implementing IN/OUT can have subtle semantic differences.
- Some languages support many param passing modes (C++: by value, by reference, by const-reference; array-mode), some few (Java: by value).

Call by value

- IN mode:
 - can't have an effect on caller vars
 - *unless* value passed is a pointer (C) or reference (Java)
- used by many languages, incl.: C/C++, Java
- How it works:
 - Formal parameter is just like a local name: storage of formals in activation record of called proc.
 - Caller evaluates the actual params and sticks their R-values in the storage for the formals
- Note: this is the mechanism we've been describing when discussing code gen of function calls

Call by reference

- IN/OUT mode
- General idea: the formal and actual refer to the same memory location
- example languages: C++, Pascal
- How it works:
 - if an actual param is a name, or an expression having an L-value, the L-value (i.e., address) is passed.
 - if it doesn't have an L-value (e.g., $i+3$), then eval the expression in a new temp loc, and pass that address
 - in called proc., code generated for a reference to the formal dereferences the pointer passed
- I.e., the compiler does for us what programmers are forced to do themselves in C to get IN/OUT by “pass by pointer”.

Call by value-return

- IN/OUT mode
- General idea: copy the value in at call, then copy back at return
- example languages: some implementations of Fortran (other impls. use call by reference)
- How it works:
 - caller evals actual params. R-values are passed (as in pass by value). L-values are also computed.
 - on return from called routine current R-values of formals are copied back into L-values of actuals.

Comparison of call-by-reference and value-return

- value-return has higher cost at call/return time, but lower cost per variable reference (no-deref. necessary)
- could use value-return in situation where called proc uses a different address space (e.g., remote procedure call (RPC)).
- danger: in some cases get different results from value-return and reference. we'll see an example of this soon
 - means mechanism needs to be specified by language designer: older Ada and Fortran's gave implementers the choice.

Call by name

- IN/OUT mode
- General idea: similar to macro-expansion
- example languages: Algol-60 (first “structured” language), some functional languages (Miranda, Haskell)
- How it works:
 - proc. call done as macro expansion:
actual parameters substituted for formals in all places formals occur in proc. (i.e. variable ref sites)
 - local names are kept distinct (in case of conflicts)
 - actuals surrounded by parens to preserve precedence of operations
 - the chunk of code to eval the actual in the calling environment is called a *thunk* .

Call by name (cont.)

- Why would we want to do this?
 - *lazy evaluation*:
 - only eval params when needed. will save time if parameter is not needed at all.
 - function call is much less expensive.
 - also called *late binding* of parameters
- Disadvantages: if some ops have side effects, or with aliasing, sometimes difficult to anticipate results (more error-prone programs?). We'll see an example.

Use of call-by-name in func. languages

- in purely functional languages expressions have no side effects. I.e., get same result if you eval it one time or ten times. (Thus, call by name is for IN only)
- lazy eval. means may eval expr. zero times
- *memoization* means we'll eval expr. at most one time.
memoization idea:
 - 1st time you eval param (i.e., at first reference site), save it's value.
 - next time, use the already computed value
 - (i.e., if 10 references, only compute the value the first time).
 - E.g., suppose the value being computed is the result of fib(10), you don't have to do the repeated recursive calls more than once.

Example

We'll look at results using each parameter passing mode:

```
int i;
int A[3];

void Q(int B) {
    A[1] = 3;
    i = 2;
    write(B);
    B = 5;
}

int main() {
    i = 1;
    A[1] = 2;
    A[2] = 4;
    Q(A[i]);
}
```