When we activate a procedure, the formal parameters get bound to locations containing values.

- How is this done and which locations are used?
- Do we pass addresses or contents of variables from the caller?
- How do we pass actual values that aren’t variables?
- What does it mean to pass a large value like an array?

Two main approaches:

- call-by-value
- call-by-reference
CALL-BY-VALUE

- Each actual argument is evaluated to a value before call.
- On entry, value is bound to formal parameter just like a local variable.
- Updating formal parameter doesn’t affect actuals in calling procedure.

```c
double hyp(double a, double b) {
    a = a * a;
    b = b * b;
    return sqrt(a+b);
}
```

- Simple; easy to understand!
- Implement by binding the formal parameters to freshly-allocated locations, and and copying the actual values into these locations (just like assignment).
PROBLEMS WITH CALL-BY-VALUE (1)

- Can be inefficient for large directly-represented values:

Example (C): Calls to `dotp` copy 20 doubles

```c
typedef struct {double a1,a2,...,a10;} vector;

double dotp(vector v, vector w) {
    return v.a1 * w.a1 + v.a2 * w.a2 + ...
        + v.a10 * w.a10;
}

vector v1,v2;
double d = dotp(v1,v2);
```
PROBLEMS WITH CALL-BY-VALUE (2)

• Cannot affect calling environment directly. (Of course, perhaps this is a **good** thing!)

Example: calls to `swap` have no effect:

```c
void swap(int i, int j) {
    int t;
    t = i; i = j; j = t;
}
...swap(a[p], a[q]);
```

• Can at best **return** only one result (as a value), though this might be a record.
CALL-BY-REFERENCE

• Pass the existing **location** of each actual parameter.
• On entry, the formal parameter is bound to this location, which must be
dereferenced to get value, but can also be **updated**.
• If actual argument doesn’t have a location (e.g., “2 + 3”), either:
  - Evaluate it into a temporary location and pass address of temporary, or
  - Treat as an error.
• Now **swap**, etc., work fine!
• Accesses are slower.
• Lots of opportunity for **aliasing** problems, e.g.,
  
  \[
  \text{PROCEDURE matmult}(a,b,c: \text{ MATRIX})
  \]
  
  \[
  \ldots \text{ (* sets } c := a \ast b \ast)\]

  \[
  \text{matmult}(a,b,a) \text{ (* oops! *)}\]

• **Call-by-value-result** (a.k.a. **copy-restore**) addresses this problem, but
  has other drawbacks.
HYBRID METHODS; RECORDS AND ARRAYS

How might we combine the simplicity of call-by-value with the efficiency of call-by-reference, especially for large directly-represented values?

- In Pascal, Ada, and similar languages, where records and arrays are both represented directly, the programmer can specify (in the procedure header) for each parameter whether to use call-by-value or call-by-reference.

- In ANSI C/C++, record (struct or class) values are represented directly, but arrays are represented indirectly. C always uses call-by-value, but programmers can take the address of a variable explicitly, and pass that to obtain cbr-like behavior:

  ```c
  swap(int *a, int *b) {
      int t;
      t = *a; *a = *b; *b = t;
  }
  swap (&a[p],&a[q]);
  ```

Of course, it is the programmer’s responsibility to make sure that the address remains valid (especially when it is returned from a function).
• C++ supports both cbr parameters and explicit pointers:

```c
swap(int &a, int *b) {
    int t;
    t = a; a = *b; *b = t;
}
...
swap(a[p], &a[q]);
```

Mixing explicit and implicit pointers can be very confusing!

• In Java and ML, values of both records (objects) and arrays are represented indirectly. These languages have only call-by-value, but this doesn’t actually cause copying, even for record or array values.

• Approach is made more feasible because programmer doesn’t have to worry about lifetime of heap data, due to automatic garbage collection.

• Clever compilers can decide whether smallish objects should be heap-allocated or manipulated directly, while continuing to give the semantic effect of indirect representation.
One simple way to give semantics to procedure calls is to say they should behave as if the procedure body was **textually substituted** for the call, with the actual parameters substituted for the formal ones.

- This is very similar to **macro-expansion**, which really does this substitution (statically). E.g (in C):
  ```c
  #define swap(x,y) {int t; t = x; x = y; y = t;}
  ...
  swap(a[p],a[q]);
  ```

- It even makes sense for recursive procedures (though of course it cannot be **implemented** by static substitution in this case).

- **BUT** blind substitution is dangerous because of possible “variable capture,” e.g.,
  ```c
  swap(a[t],a[q])
  ```

expands to
  ```c
  {int t; t = a[t]; a[t] = a[q]; a[q] = t;}
  ```
**CALL-BY-NAME**

- Here $t$ is “captured” by the declaration in the macro, and is undefined at its first use.
- Note that name of local variable is not important: it could be renamed:
  ```
  {int u; u = a[t]; a[t] = a[q]; a[q] = u;}
  ```
- **Call-by-name** (first proposed in Algol60) can be thought of as “substitution with renaming where necessary.”
- In practice, call-by-name is implemented by binding any free variables in arguments at the point of call (rather than the point of use).
- This makes CBN much less efficient to implement than CBV or CBR.
Call-by-name is flexible, but potentially very confusing in the presence of side-effects.

```plaintext
real procedure SIGMA(x, i, n);
    real x; integer i, n;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    SIGMA := s;  (sets return value)
end

SIGMA(a(i),i,10);   (computes $\sum_{i=1}^{10} a_i$)
SIGMA(a(i)*b(i),i,10); (computes $\sum_{i=1}^{10} a_i b_i$)
```
If language has no mutable variables (as in “pure” functional languages), call-by-name gives a substitution gives a beautifully simple semantics for procedure calls.

Arguments are evaluated only if needed.

```plaintext
foo x y = if x > 0 then x else y

foo 1 (factorial 1000000)
```

As a further refinement, pure functional languages typically use call-by-need (or lazy) evaluation, in which arguments are evaluated at most once.

```plaintext
foo x y = if x > 0 then x else y * y

foo (-1) (factorial 1000000)
```