

C++ provides true reference parameters (like Pascal & Modula-2 **var** parameters):

C++

```
void incr( int& i )
{ i = (i+1)%size; }
...
incr( x );
```

C (and C++)

```
void incr( int* i )
{ *i = (*i+1)%size; }
...
incr( &x );
```

In C++, **&** is a true *type constructor* (unlike **var**).

This enables references to be *returned*, as in

```
int& min( int& x, int& y )
{
  if ( x < y ) return x;
  return y;
}
...
int z = min( p, q );
min( a[i], a[j] ) = 7;
min( p, q )++;
```

In Pascal & Modula-2, reference (**var**) parameters

- (+) eliminate (expensive) argument copying
- (-) make arguments vulnerable to update

In C++, we can get (+) without (-) by declaring parameters **const...&**, as in

```
int max( const int x[] ) { ... }
```

This is especially useful for C++ arrays (which are—as in C—*reference* parameters).

3 April 2005

### By Value-Result

Not available in C or Pascal (“in out” in Ada).

Combination of *by-value* and *by-reference*:

At call:  $par_i := arg_i$  “copy-in”

At return:  $arg_i := par_i$  “copy out”  
(iff  $arg_i$  has an *l-value*)

Same as *by-reference*? Not quite— consider this:

```
var n;
procedure f(x); begin x := 5; n := 0 end
begin
  n := 3; f(n); print(n)
end.
```

$n$ 's final value:

if by value-result: 5

if by reference: 0

The reason for the difference is **aliasing**:

two **different names** — $n$  and  $x$ —  
refer to the **same location**

Ada's *by-copy* binding is designed to prevent aliasing. But copying can be expensive, so Ada also provides *by-reference* binding.

Minor variations (in Ada):

to omit *copy-in* at call, specify **out par**;

to omit *copy-out* at return, specify **in par**.

3 April 2005

### By Name

(last seen in ALGOL-60)

Effect is like textual substitution:

1. in a copy of the body, replace each occurrence of *par* with a copy of the *arg* expression (without evaluating it)
2. substitute the modified body for the call

Advantage: no unnecessary evaluation:

```
procedure f(x,y)
  if x=0 then return else loop ... y ... end
end f;
begin
  f(0, exp1); f(1, exp2)
end
```

Disadvantage:

an argument can be evaluated repeatedly.

Number of evaluations of argument: *evals*

call-by-**value**:  $evals = 1$

call-by-**name**:  $0 \leq evals \leq \infty$

In functional languages, call-by-name is known as the **outermost** evaluation strategy.

3 April 2005

Potential *name conflicts* arising from the use of the same name in argument expressions and in the procedure body are avoided by two rules:

1. **argument** names are bound by the environment of each procedure **call**
2. **body** names are bound by the environment of the procedure's **declaration**.

Example:

```
integer p = 10, z = 4;
procedure f( integer x, y );
  integer p = 5;
  x := x * p + y
end;
```

the statement

$f( z, \text{sqrt}( p ) );$

is equivalent to

$z := z * 5 + \text{sqrt}( 10 )$

Implementation: Each reference to a *call-by-name* parameter is implemented by

0. a *jump* to the argument (which is called a “*thunk*”)
1. execution of the *thunk* (side-effects possible)
2. a *jump back* into the procedure

3 April 2005

**Macro-expansion vs. procedure calls**

Both “employ” textual substitution for parameter-binding:

- in procedure calls it’s a theoretical model
- in macro-expansion it’s the actual implementation.

Macro definition:

```
#define name [ [ ( par { , par } ) ] body ]
```

Macro invocation:

```
name [ ( arg { , arg } ) ]
```

Example:

```
#define SQUARE(x) ((x) * (x))
```

A macro processor would translate

```
j = SQUARE(a + b);
```

into

```
j = ((a + b) * (a + b));
```

before compiling.

The expansion steps:

1. substitute text of macro call’s argument for each occurrence of *par* in body;
2. substitute resulting body for the macro call.

Example: a macro “swap”.

```
#define swap(a,b) {int z; z=a; a=b; b=z;}
```

Then

```
swap( i, A[i] );
```

becomes

```
{ int z; z=i; i=A[i]; A[i]=z; }
```

—which fails to interchange values of *i* and *A[i]*.

This is a **side-effect** problem— changing *i*’s *r*-value changes *A[i]*’s *l*-value.

Call-by-name fails in the same way.

The cure: Use call-by-**reference**.

Macros can also cause **name conflicts**, e.g.,

```
float e = 2.718281828459045;
```

```
#define p(x) (x/e)
```

```
...
```

```
int f( int e ){... p(z)...}
```

which macro-expands to

```
float e = 2.718281828459045;
```

```
...
```

```
int f( int e )
```

```
{... (z/e)...} /* divides by the wrong e */
```

The macro expansion imports *e* into the scope of a different binding of *e*—a case of **dynamic binding**.

Such naming problems could be avoided by true call-by-name parameter binding.

One attraction of macros such as `swap` is a kind of polymorphism— a single macro

```
#define swap(t, a, b) {t z = a; a = b; b = z;}
```

can take the place of multiple function definitions:

```
void intSwap(int& a, int& b)
{ int z; z=a; a=b; b=z; }
```

```
void floatSwap(float& a, float& b)
{ float z; z=a; a=b; b=z; }
```

```
void charSwap(char& a, char& b)
{ char z; z=a; a=b; b=z; }
```

In C++, we could use function-name *overloading* to cut down on the proliferation of function names—

```
void swap(int& a, int& b)
{ int z; z=a; a=b; b=z; }
```

```
void swap(float& a, float& b)
{ float z; z=a; a=b; b=z; }
```

```
void swap(char& a, char& b)
{ char z; z=a; a=b; b=z; }
```

...but we still have three functions which are essentially identical.

Better than overloading, for procedures like `swap`, are C++ function *templates*, which provide another kind of polymorphism:

```
template <typename T>
void swap( T& a, T& b )
{ T z; z=a; a=b; b=z; }
```

Given this template and

```
int m, n; float a, b;
```

some possible calls of `swap` are:

```
swap( m, n );           T := int
swap( a, b );          T := float
swap( a, n )           error: ambiguous call
```

The C++ compiler uses the *function template* to generate a *template function* for each argument type.

**Observation:** What looks on the surface like parametric polymorphism turns out to be another form of overloading, because a function template:

- generates a *different* template function for each type
- generates template functions *only* for argument types for which the functions it applies to its parameters are defined.