

(sometimes called a *thunk*) that is evaluated to obtain the value or address of the corresponding actual parameter. Each reference to the formal parameter causes the thunk to be reevaluated.

- Readonly (sometimes called Const): Only reads of the formal parameter are allowed. Either a copy of the actual parameter's value, or its address, may be used.

## What Parameter Modes do Programming Languages Use?

- C and C++: Value mode except for arrays where a pointer to the start of the array is passed.
- Java: Scalar types (`int`, `float`, `char`, etc.) are passed by value; objects are passed by reference.
- Fortran: Reference (even for constants!)
- Ada: Value/result, reference, and readonly are used.

## Example

```
void p(value int a,  
      reference int b,  
      name int c) {  
    a=1; b=2; print(c)  
}  
int i=3, j=3, k[10][10];  
p(i,j,k[i][j]);
```

What element of `k` is printed?

- The assignment to `a` does not affect `i`, since `a` is a value parameter.
- The assignment to `b` *does* affect `j`, since `b` is a reference parameter.
- `c` is a name parameter, so it is evaluated whenever it is used. In the print statement `k[i][j]` is printed. At that point `i=3` and `j=2`, so `k[3][2]` is printed.

## Why are there so Many Different Parameter Modes?

Parameter modes reflect different views on how parameters are to be accessed, as well as different degrees of efficiency in accessing and using parameters.

- Call by value *protects* the actual parameter value. No matter what the subprogram does, the parameter *can't* be changed.
- Call by reference allows *immediate* updates to the actual parameter.
- Call by readonly protects the actual parameter and emphasizes the "constant" nature of the formal parameter.

- Call by value/result allows actual parameters to change, but treats a call as a single step (assign parameter values, execute the subprogram's body, update parameter values).
- Call by name delays evaluation of an actual parameter until it is actually needed (which may be never).

## Call by Name

Call by name is a special kind of parameter passing mode. It allows some calls to complete that otherwise would fail. Consider

`f(i, j/0)`

Normally, when `j/0` is evaluated, a divide fault would terminate execution. If `j/0` is passed by name, the division is delayed until the parameter is needed, which may be never.

Call by name also allows programmers to create some interesting solutions to hard programming problems.

Consider the conditional expression found in C, C++, and Java:

`(cond?value1:value2)`

What if we want to implement this as a function call:

```
condExpr(cond,value1,value2) {
    if cond
        return value1;
    else return value2;
}
```

With most parameter passing modes this implementation *won't work!* (Why?)

But if `value1` and `value2` are passed by name, the implementation is correct.

## Call by Name and Lazy Evaluation

Call by name has much of the flavor of *lazy evaluation*. With lazy evaluation, you don't compute a value but rather a *suspension*—a function that will provide a value when called.

This can be useful when we need to control how much of a computation is actually performed.

Consider an infinite list of integers. Mathematically it is represented as

1, 2, 3, ...

But how do we compute a data structure that represents an infinite list?

The obvious computation

```
infList(int start) {  
    return list(start,  
                infList(start+1));  
}
```

*doesn't* work. (Why?)

A less obvious implementation, using suspensions, *does* work:

```
infList(int start) {  
    return list(start,  
                function() {  
                    return infList(start+1);  
                });  
}
```

Now, whenever we are given an infinite list, we get two things: the first integer in the list and a suspension function. When called, this function will give you the rest of the infinite list (again, one more

value and another suspension function).

The whole list is there, but only as much as you care to access is actually computed.

## Eager Parameter Evaluation

Sometimes we want parameters evaluated *eagerly*—as soon as they are known.

Consider a sorting routine that breaks an array in half, sorts each half, and then merges together the two sorted halves (this is a *merge sort*).

In outline form it is:

```
sort(inputArray) {  
    ...  
    merge(sort(leftHalf(inputArray),  
              sort(rightHalf(inputArray)));  
}
```

This definition lends itself nicely to parallel evaluation: The two halves of an input array can be sorted in parallel. Each of these two halves can again be split in two, allowing parallel sorting of four quarter-sized arrays,

then leading to 8 sorts of 1/8 sized arrays, etc.

*But,*  
to make this all work, the two parameters to merge must be evaluated *eagerly*, rather than in sequence.

## Type Equivalence

Programming languages use types to describe the values a data object may hold and the operations that may be performed.

By checking the types of values, potential errors in expressions, assignments and calls may be automatically detected. For example, type checking tells us that

```
123 + "123"
```

is illegal because addition is not defined for an integer, string combination.

Type checking is usually done at compile-time; this is *static typing*.

Type-checking may also be done at run-time; this is *dynamic typing*.

A program is *type-safe* if it is impossible to apply an operation to a value of the wrong type. In a type-safe language, plus is never told to add an integer to a string, because its definition does not allow that combination of operands. In type-safe programs an operator can still see an illegal value (e.g., a division by zero), but it can't see operands of the wrong type.

A *strongly-typed* programming language forbids the execution of type-unsafe programs.

*Weakly-typed* programming languages allow the execution of potentially type-unsafe programs.

The question reduces to whether the programming language allows

programmers to “break” the type rules, either knowingly or unknowingly.

Java is strongly typed; type errors preclude execution. C and C++ are weakly typed; you can break the rules if you wish. For example:

```
int i;  int* p;  
p = (int *) i * i;
```

Now `p` may be used as an integer pointer though multiplication need not produce valid integer pointers.