What Parameter Modes do Programming Languages Use?

- C: Value mode except for arrays which pass a pointer to the start of the array.
- C++: Allows reference as well as value modes. E.g.,
  `int f(int a, int& b)`
- C#: Allows result (out) as well as reference and value modes. E.g.,
  `int g(int a, out int b)`
- Java: Scalar types (**`int`**, **`float`**, **`char`**, etc.) are passed by value; objects are passed by reference (references to objects are passed by value).
- 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* terminates 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, ...

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.

If we are going to do type checking in a program, we must decide whether two types, T1 and T2 are equivalent; that is, whether they be used interchangeably.

There are two major approaches to type equivalence:

Name Equivalence:

Two types are equivalent if and only if they refer to exactly the same type declaration.

For example,

```
type PackerSalaries = int[100];
type AssemblySizes = int[100];
PackerSalaries  salary;
AssemblySizes  size;
```

Is

```
sal = size;
```

allowed?

Using name equivalence, *no*. That is, **salary** $\not\equiv_N$ **size** since these two variables have different type declarations (that happen to be identical in structure).

Formally, we define $\equiv_N$ (name type equivalence) as:

(a) $T \equiv_N T$

(b) Given the declaration

```
Type T1 = T2;
```

$T1 \equiv_N T2$

We treat anonymous types (types not given a name) as an abbreviation for an implicit declaration of a new and unique type name.

Thus

```
int  A[10];
```

is an abbreviation for

```
Type T_new = int[10];
T_new  A;
```

Structural Equivalence

An alternative notion of type equivalence is structural equivalence (denoted $\equiv_S$).

Roughly, two types are structurally equivalent if the two types have the same definition, independent of where the definitions are located. That is, the two types have the same definitional structure.

Formally,

(a) $T \equiv_S T$

(b) Given the declaration

```
Type T = Q;
```

$T \equiv_S Q$

(c) If T and Q are defined using the same type constructor and corresponding parameters in the

two definitions are equal or structurally equivalent

then $T \equiv_S Q$

Returning to our previous example,

```
type PackerSalaries = int[100];
type AssemblySizes = int[100];
PackerSalaries  salary;
AssemblySizes  size;
```

salary $\equiv_S$ **size** since both are arrays and **100=100** and **int** $\equiv_S$ **int**.

Which notion of Equivalence do Programming Languages Use?

C and C++ use structural equivalence except for structs and classes (where name equivalence is used). For arrays, size is ignored.

Java uses structural equivalence for scalars. For arrays, it requires name equivalence for the component type, ignoring size. For classes it uses name equivalence except that a subtype may be used where a parent type is expected. Thus given

```
void subr(Object O) { ... };
```

the call

```
subr(new Integer(100));
```

is OK since **Integer** is a subclass of **Object**.

Automatic Type Conversions

C, C++ and Java also allow various kinds of automatic type conversions.

In C, C++ and Java, a **float** will be automatically created from an **int**:

```
float f = 10; // No type error
```

Also, an integer type (**char**, **short**, **int**, **long**) will be *widened*:

```
int i = 'x';
```

In C and C++ (but not Java), an integer value can also be *narrowed*, possibly with the loss of significant bits:

```
char c = 1000000;
```