Issues in Overloading

Though many languages allow overloading, few allow overloaded methods to differ only on their result types. (Neither C++ nor Java allow this kind of overloading, though Ada does). For example,

```
class MyClass {
  int f() { ... }
  float f() { ... }
}
```

is illegal. This is unfortunate; methods with the same name and parameters, but different result types, could be used to automatically convert result values to the type demanded by the context of call.

Why is this form of overloading usually disallowed?

It's because overload resolution (deciding which definition to use) becomes much harder. Consider

```
class MyClass {
  int   f(int i, int j) { ... }
  float f(float i, float j) { ... }
  float f(int i, int j) { ... }
}
```

in

```
 int a = f( f(1,2), f(3,4) );
```

which definitions of `f` do we use in each of the three calls? Getting the correctly answer can be tricky, though solution algorithms do exist.

Operator Overloading

Some languages, like C++ and C#, allow operators to be overloaded. You may add new definitions to existing operators, and use them on your own types. For example,

```
 class MyClass {
    int i;
  public:
    int operator+(int j) {
            return i+j; }
}
MyClass c;
int i = c+10;
int j = c.operator+(10);
int k = 10+c; // Illegal!
```

The expression **`10+c`** is illegal because there is no definition of **`+`** for the types **`int`** and **`MyClass&`**. We can create one by using C++'s *friend* mechanism to insert a definition into **`MyClass`** that will have access to **`MyClass`**'s private data:

```
class MyClass {
    int i;
   public:
    int operator+(int j) {
           return i+j; }
    friend int operator+
       (int j, MyClass& v){
           return j+v.i; }
}
MyClass c;
int k = 10+c; // Now OK!
```

C++ limits operator overloading to existing predefined operators. A few languages, like Algol 68 (a successor to Algol 60, developed in 1968), allow programmers to define brand new operators.

In addition to defining the operator itself, it is also necessary to specify the operator's precedence (which operator is to be applied first) and its associativity (does the operator associate from left to right, or right to left, or not at all). Given this extra detail, it is possible to specify something like

```
op +++ prec = 8;
int op +++(int& i, int& j) {
  return (i++)+(j++); }
```

(Why is **`int&`** used as the parameter type rather than **`int`**?)

Parameter Binding

Almost all programming languages have some notion of *binding* an *actual parameter* (provided at the point of call) to a *formal parameter* (used in the body of a subprogram).

There are many different, and inequivalent, methods of parameter binding. Exactly which is used depends upon the programming language in question.

Parameter Binding Modes include:

- Value: The formal parameter represents a local variable initialized to the value of the corresponding actual parameter.

- Result: The formal parameter represents a local variable. Its final value, at the point of return, is copied into the corresponding actual parameter.
- Value/Result: A combination of the value and results modes. The formal parameter is a local variable initialized to the value of the corresponding actual parameter. The formal's final value, at the point of return, is copied into the corresponding actual parameter.
- Reference: The formal parameter is a pointer to the corresponding actual parameter. All references to the formal parameter indirectly access the corresponding actual parameter through the pointer.

- Name: The formal parameter represents a block of code (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: 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.