

## Reading Assignment

- Sethi: Chapters 4-5
- Scott: Chapters 3

## 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, since 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 correct answer can be tricky, though solution algorithms do exist.

## Operator Overloading

Some languages, like 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.