Overloading

Outline

- Function overloading
- Operator overloading

Function overloading

- Overloaded functions have
  - Same name
  - Different sets of parameters
- Overloaded functions are distinguished by their signatures
- Compiler selects proper function to execute based on number, types and order of arguments in the function call
- Commonly used to create several functions of the same name that perform similar tasks, but on different data types
- Creating overloaded functions with identical parameter lists and different return types is a compilation error

Example

- Declaration& Definition

```
void AddAndDisplay(int x, int y) {
    cout<<" Integer result: "<<(x+y);
}

void AddAndDisplay(double x, double y) {
    cout<< " Double result: "<<(x+y);
}

void AddAndDisplay(float x, float y) {
    cout<< " Float result: "<<(x+y);
}
```

- Usage

AddAndDisplay(1,1); √ : call AddAndDisplay(int, int)

AddAndDisplay(1.0,1.0); √ : call AddAndDisplay(double, double)

AddAndDisplay(1,1.0); ✗ : call AddAndDisplay(int, double)

√ : AddAndDisplay((double)1, 1.0)
AddAndDisplay(1, (int)1.0)

Operator Overloading

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
  - Operator sensitive to context
- Almost all operators in C++ can be overloaded except:
  - `.` `::` `?:` `sizeof`
- Can only redefine existing operators, but CANNOT define new operators.
- CANNOT change the properties of an operator
  - *Number of arguments* an operator takes. (So you are not allowed to re-define the plus operator to take 3 arguments instead of 2.)
  - *Associativity*. E.g.: `a+b+c` is always identical to `(a+b)+c`.
  - *Precedence*. E.g.: `a+b*c` is treated as `a+(b*c)`.

Operator Overloading

- For a **user-defined class type,** every operator defined must have at least one argument.
- For a **global function,** `operator+` has two arguments. When it is called in an expression such as `a+b`, this is equivalent to writing `operator+(a, b)`.

Example: Non-member operator function

```
class Vector
{
    double vx, vy;
public:
    Vector(double x, double y) : vx(x), vy(y) { }
    double x() const { return vx; }
    double y() const { return vy; }
};
```

- To add 2 vectors, traditionally we would do like this:

```
Vector& add (const Vector& a, const Vector& b)
{
     return Vector( a.x() + b.x(), a.y() + b.y() );
}
```

```
d = add(a, add(b, c));    // d = a + b + c
```

Example: Non-member Operator Function (cont.)

- Using operator overloading, we can do like this:

```
Vector& operator+ (const Vector& a, const Vector& b)
{
     return Vector( a.x() + b.x(), a.y() + b.y() );
}
```

```
Vector a(1, 3), b(-5, 7), c(22, 2), d;
d = a + b + c;       // d = operator+(operator+(a, b), c);
```

Example: Member operator function

```
class Vector {
    double vx, vy;
public:
    Vector(double x, double y) : vx(x), vy(y) { }
    double x() const { return vx; }
    double y() const { return vy; }
    Vector& operator+ (const Vector& b) const
        { return Vector( x + b.x(), y + b.y() ); }
};
```

- Whenever the compiler sees an expression of the form `a+b`, it converts this to the two possible representations:
`operator+(a, b)` OR `a.operator+(b)`
and verifies whether one of those two operator functions are defined.
- **Note: It is an ERROR to define both.**

Example: operator function (Summary)

- Approach 1: Non-member operator

```
Vector& add (const Vector& a, const Vector& b){
  return Vector( a.x() + b.x(), a.y() + b.y() );
}
```

- Approach 2: Non-member operator overloading

```
Vector& operator+ (const Vector& a,const Vector& b){
     return Vector( a.x() + b.x(), a.y() + b.y() );
}
```

- Approach 3: Member operator overloading

```
Vector& operator+ (const Vector& b) const    {
     return Vector( x + b.x(), y + b.y() );
}
```

- Usage:

```
d = add(a, add(b, c));    //Approach 1
d = a + b + c;            //Approach 2&3
  representation
     operator+(a, b)    //Approach 2
     a.operator+(b)     //Approach 3
```

Member or Non-member Function?

- Does the following work by defining the `operator*` as a member function of Vector?

```
Vector c = 2 * a;  // c == (2, 0)
```

- NO! 2 is an object of type int, and we cannot define a new member function for this type.
- So our only choice is to define the multiplication operator as a global non-member function:

```
Vector operator* (double s, const Vector& a) {
     return Vector(s * a.x(), s * a.y()); }
```

```
Vector c = a * 2;
a.operator(2);
  Sol 1. Vector operator* (const Vector& a, double s)
  Sol 2. Vector Vector( int i );
```

Member or Non-member function?

- The Operators: `=`, `[]`, `()` are required by C++ to be defined as class member functions.
- If the left operand of an operator must be an object of the class, it c*an* be a member function.
- If the left operand of an operator may be an object of other classes, it *must* be a non-member function. E.g.: `operator<<`.

Example: Operator Function for Printing

```
ostream& operator<<(ostream& os, const Vector& a){
    os << '(' << a.x() << ',' << a.y() << ')';
    return os;
}

Vector a(1,0);
cout << "a = " << a << endl;      // a = (1,0)
```

- We CANNOT define `operator<<` as a member function.
- MUST be a *friend* function if it access private data members.
  - Declare in class

```
friend ostream& operator<<(ostream& os, const Vector& a) {
     os << '(' << a.vx << ',' << a.vy << ')';
     return os;
}
```

```
class Vector {
    double vx, vy;
Public:
    double x() const { return vx; }
    double y() const { return vy; }
    //…
```

Prefix and Postfix Operators

```
class Vector {
    double vx, vy;
public:
    Vector() : vx(0.0), vy(0.0) { }
    Vector(double x, double y) : vx(x), vy(y) { }
    Vector operator++()          //++a
        { ++vx; ++vy; return *this; }
    Vector operator++(int)      //a++
        { Vector temp(vx, vy); vx++; vy++; return temp; }
};

int main() {
    Vector a(1.2, 3.4), c, d;
    c = ++a;  // c = 2.2 + 4.4 j
    d = a++;  // d = 1.2 + 3.4 j
}
```