Deep or Shallow?

- `int x = 3;`
- `int y = x; // is the copy of x deep or shallow?`
- `// it is DEEP – we copy a value, not an address`
- `y = 2; // this will not change x!`

- `int * x = new int(3);`
- `int * y = x; // is the copy of x deep or shallow?`
- `// it is SHALLOW – we copy an address`
- `*y = 2; // BOTH x and y now point to 2`
- `delete * x; // What will happen to y?`
OPERATOR OVERLOADING

- Operators such as =, +, and others can be redefined when used with objects of a class.
- You can choose what `object1 = object2;` means.
- Every operator is basically a function that you overload.
- The name of the function for the overloaded operator is `operator` followed by the operator symbol, e.g., `operator+` to overload the `+` operator, and `operator=` to overload the `=` operator.
- Prototype for the overloaded operator goes in the declaration of the class that is overloading it.
- Overloaded operator function definition goes with other member functions.
Let’s write a vector class:

```cpp
class Vector {
public:
    Vector(int);
    ~Vector();
    Vector(Vector&); // make a deep copy
    Vector& operator=(Vector&); // make a deep copy
    void setCoordinate(int, int);
    void print();

private:
    int * coords;
    int dimension;
};
```

What is this?
**Operator Overloading**

Overloaded =

```cpp
Vector& Vector::operator=(Vector& vec)
{
    delete [] coords; // deallocates previous vector

    int dimension = vec.dimension;
    coords = new int[dimension];

    for(int i = 0; i < dimension; i++)
        coords[i] = vec.coords[i];

    return *this;
}
```

What is this?
**Operator Overloading: ** *THIS*

Consider:

```cpp
#include "Vector.h"
int main()
{
    Vector vec(3); // vector of dimension 3
    setCoordinate(0, 5); // does this make sense?
    // NO! A class function can only be invoked on 
    // an object of that class:
    vec.setCoordinate(0, 5);
}
```

Every member function knows what object has invoked it

`this*` pointer will point to the object on which the function is invoked
OPERATOR OVERLOADING

A reference to *this is returned:

```
Vector& Vector::operator=(const Vector& vec)
{
    delete [] coords; // deallocates previous vector

    int dimension = vec.dimension;
    coords = new double[dimension];

    for(int i = 0; i < dimension; i++)
        coords[i] = vec.coords[i];

    return *this;
}
```
# Returning “by reference” is similar to “passing by reference”

<table>
<thead>
<tr>
<th>“Pass by value”</th>
<th>“Pass by reference”</th>
</tr>
</thead>
<tbody>
<tr>
<td>void doStuff(int x)</td>
<td>void doStuff(int &amp;x)</td>
</tr>
<tr>
<td>{</td>
<td>// Notice the “&amp;”</td>
</tr>
<tr>
<td>x++;</td>
<td>{</td>
</tr>
<tr>
<td>}</td>
<td>x++;</td>
</tr>
<tr>
<td>int y = 42;</td>
<td>int y = 42;</td>
</tr>
<tr>
<td>doStuff(y);</td>
<td>doStuff(y);</td>
</tr>
<tr>
<td>cout &lt;&lt; y &lt;&lt; endl;</td>
<td>cout &lt;&lt; y &lt;&lt; endl;</td>
</tr>
<tr>
<td>// The output is 42 :</td>
<td>// The output is 43 :</td>
</tr>
<tr>
<td>// y has not being modified</td>
<td>// y has being modified!</td>
</tr>
</tbody>
</table>

When a reference is returned, you get direct access to the variable that is returned.
Let’s overload a math operator:

class Vector {
    public:
        Vector(int);
        ~Vector();
        Vector(Vector&); // make a deep copy
        Vector& operator=(const Vector&); // deep copy
        Vector operator+(Vector&);

        void setCoordinate(int, double);
        void print();

    private:
        int * coords;
        int dimension;
};
**OPERATOR OVERLOADING**

- This is how we’ll use it:

  ```cpp
  Vector x(1), y(1);
  x.setCoordinate(0, 2); // x is [2]
  y.setCoordinate(0, 3); // y is [3]
  Vector z = x + y; // Now z is [5]
  ```

- This is how we’ll implement it:

  ```cpp
  Vector Vector::operator+(Vector& vec)
  {
      Vector temp(dimension); // result vector
      for(int i = 0; i < dimension; i++) {
          temp.coords[i] = coords[i] + vec.coords[i];
      }
      return temp;
  }
  ```
Let’s overload a pre-increment operator:

```cpp
class Vector {
    public:
        Vector(int);
        ~Vector();
        Vector(Vector&);  // make a deep copy
        Vector& operator=(const Vector&); // deep copy

        Vector operator++();

        void setCoordinate(int, int);
        void print();

    private:
        int * coords;
        int dimension;
};
```
**Operator Overloading**

- This is how we’ll use it:

  ```cpp
  Vector x(1);
  x.setCoordinate(0, 2); // x is [2]
  Vector y = ++x; // Now x should be [3],
                  // and y should be [3].
  ```

- This is how we’ll implement it:

  ```cpp
  Vector Vector::operator++()
  {
    for(int i = 0; i < dimension; i++) {
      ++coords[i];
    }
    return *this;
  }
  ```
Let’s overload a post-increment operator:

```cpp
class Vector {
    public:
        Vector(int);
        ~Vector();
        Vector(Vector&); // make a deep copy
        Vector& operator=(const Vector&); // deep copy
        Vector operator++(int);
    
    void setCoordinate(int, int);
    void print();

    private:
        int * coords;
        int dimension;
};
```
OPERATOR OVERLOADING

This is how we’ll use it:

Vector x(1);
x.setCoordinate(0, 2); // x is [2]
Vector y = x++; // Now x should be [3],
               // and y should be [2].

This is how we’ll implement it:

Vector Vector::operator++(int)
// what is this empty int?
{
    Vector temp(*this); // what’s happening?
    for(int i = 0; i < dimension; i++) {
        coords[i]++;
    }
    return temp; // why not *this?
}
Let’s overload a comparison operator:

```cpp
class Vector {
    public:
        Vector(int);
        ~Vector();
        Vector(Vector&); // make a deep copy
        Vector& operator=(const Vector&);  // deep copy
        bool operator==(Vector&);

    void setCoordinate(int, int);
    void print();

    private:
        int * coords;
        int dimension;
};
```
OPERATOR OVERLOADING

This is how we’ll use it:

```cpp
Vector x(1), y(1);
x.setCoordinate(0, 2); // x is [2]
y.setCoordinate(0, 3); // y is [3]
if(x == y) cout << "identical vectors!\n";
else cout << "different vectors!\n";
```

This is how we’ll implement it:

```cpp
bool Vector::operator==(Vector& vec)
{
    // can you spot any problems?
    bool same = true;
    for(int i = 0; i < dimension; i++) {
        if (coords[i] != vec.coords[i])
            same = false;
    }
    return same;
}
```
Next lecture:
- Overloading [ ], <<, >> operators

Remember to try operator overloading at home
First assignment (homework) will be released next week