C++ allows us to re-define operators in the context of a class:

Vector x(1), y(1);
x.setCoordinate(0, 2); // x is [2]
y = x; // y.operator=(x) (operator= is invoked)
// Now y is also [2]

What if by mistake you try to assign an object to itself?

Vector x(1), y(1);
x.setCoordinate(0, 2); // x is [2]
x = x; // ???

To understand what happens, let’s look at the code
Overloading = for the Vector class:

```cpp
Vector& Vector::operator=(const 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;
}
```

If a vector is assigned to itself, it will... **deallocate its data**. And then try to copy something that does not exist anymore.
**Operator Overloading**

- Fixing the assignment operator:

```cpp
Vector& Vector::operator=(const Vector& vec) {
    if(&vec == this) return *this;

    // The rest will only happen if this != &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 if the dynamic memory has not been allocated yet?

```cpp
Vector& Vector::operator=(const Vector& vec)
{
    if(&vec == this) return *this;
    // The rest will only happen if this != &vec

    if (dimension != 0) delete [] coords;
    // deallocates previous vector, but first checks
    // if there is anything to deallocate

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

    for(int i = 0; i < dimension; i++)
        coords[i] = vec.coords[i];
    return *this;
}
```
OPERATOR OVERLOADING

- We use `setCoordinate` to assign values to the vector:
  ```cpp
  Vector x(2);
  x.setCoordinate(0, 2); // set coord 0 to 2
  // now x is [2, 0]
  x.setCoordinate(1, 3); // set coord 1 to 3
  // now x is [2, 3]
  ```

- We use `getCoordinate` to read a value from vector:
  ```cpp
  cout << x.getCoordinate(0); // x is [2, 3],
  // this will output a 2 (value at coordinate 0)
  ```

- This syntax is ugly, though! Can’t we rather use:
  ```cpp
  Vector x(2);
  x[0] = 2; // set coord 0 to 2, now x is [2, 0]
  x[1] = 3; // set coord 1 to 3, now x is [2, 3]
  cout << x[0]; // this will output a 2
  ```
OPERATOR OVERLOADING

- Accessing vector’s data through `setCoordinate` is really bulky and inconvenient
- Let’s overload the subscript operator to make life easy:

```cpp
class Vector {
    public:
        Vector(int);
        ~Vector();
        Vector(Vector&); // make a deep copy
        Vector& operator=(Vector&); // make a deep copy
        int& operator[](int);
    void print();
    private:
        int * coords;
        int dimension;
};
```
OPERATOR OVERLOADING

This is how we’ll use it:

```cpp
Vector x(1), y(1);
x[0] = 2; // x is [2]
y[0] = 3; // y is [3]
```

This is how we’ll implement it:

```cpp
int& Vector::operator[](int index)
{
    if(index >= 0 && index < dimension)
        return coords[index];
    else {
        cout << "Invalid index" << endl;
        exit(EXIT_FAILURE);
    }
}
```

Why do we return a reference?
OPERATOR OVERLOADING

- Is our overloaded subscript going to work if the object is immutable (constant)?

```cpp
Vector x(1);
x[0] = 2;    // x is [2]
const Vector y = x; // make a constant copy of x
cout << y[0]; // print y - read-only
```

- The compiler will be unhappy: our operator returns a non-constant reference

- Immutability is not guaranteed!

- Solution: **overload** the overloaded operator
Provide two subscript operators:
- One for reading and writing
- One for read-only access

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

    int& operator[](int);
    const int& operator[](int) const;

    void print();
private:
    int * coords;
    int dimension;
};
```
OPERATOR OVERLOADING

- Implementation is the same:

  ```cpp
  const int& Vector::operator[](int i) const
  {
    if(i >= 0 && i < dimension)
      return coords[i];
    else {
      cout << "Invalid index" << endl;
      exit(EXIT_FAILURE);
    }
  }
  ```

- Now [ ] is applicable to constant objects, too.
**Operator Overloading**

- What else can we overload?
- How about...

```cpp
Vector x(1);
x[0] = 2;  // x is [2]
cout << x;  // print x nicely on the screen
```

- `<<` is called “Stream insertion operator”

- What parameters does it take?

```cpp
cout << x;  // interpret as a function call:
cout.operator<<(x);
```

- What is `cout`?
OPERATOR OVERLOADING

- What is `cout`?
  - `cout` is an object of class `ostream`.

- We can write a **stand-alone function** that takes two parameters – an output stream, and an object:

  ```
  ostream& operator<<(ostream& out, Vector& vec);
  ```

- Inside the class, we will declare this function to be a **friend** – this will allow `operator<<` to access private members

- As a matter of fact, many operators, such as `+`, can be overloaded as friend functions, not member functions
**Operator Overloading**

- Put operator prototype outside of the class (Vector.h):

```cpp
class Vector; // forward declaration for the func below
ostream& operator<<(ostream& out, Vector& vec);

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

    friend ostream& operator<<(ostream& out, Vector& vec);

    private:
        int * coords;
        int dimension;
};
```
Implement in Vector.cpp, but without ::

```cpp
ostream& operator<<(ostream& out, Vector& vec) {
    out << "[ ";
    for(int i = 0; i < vec.dimension; i++) {
        out << vec.coords[i] << " ";
    }
    out << "]" << endl;
    return out;
}
```

Why are we returning `ostream&`? For this:

```cpp
cout << "Vector x is " << x << endl;
// chaining operators
```
OPERATOR OVERLOADING

- We can also overload `>>` to provide for convenient input:
  
  ```
  Vector x(5);
  cout << "Please enter 5 coordinates:\n";
  cin >> x; // input directly into x
  ```

- `>>` is called “Stream extraction operator”

- What parameters does it take?
  
  ```
  cin >> x; // interpret as a function call:
  cin.operator>>(x);
  ```

- What is `cin`?
  - You guessed it: an object of type `istream`.
OPERATOR OVERLOADING

- Put operator prototype outside of the class (Vector.h):

```cpp
class Vector; // forward declaration for the func below
istream& operator>>(istream& in, Vector& vec);
```

class Vector {
  public:
    Vector(int);
    ~Vector();
    Vector(Vector&); // make a deep copy
    friend istream& operator>>(istream& in, Vector& vec);
  private:
    int * coords;
    int dimension;
};
```
Operator Overloading

- Implement in Vector.cpp, but without ::

```cpp
istream& operator>>(istream& in, Vector& vec) {
    for(int i = 0; i < vec.dimension; i++) {
        in >> vec.coords[i];
    }
    return in;
}
```

- Now the following is valid:

```cpp
Vector vector(5);
cout << "Please enter 5 coordinates:\n";
cin >> vector; // input directly into vector
cout << vector; // use previously defined <<
```