Ranged for support (for our Vector class)

```
for (const int* iter = vec.begin(); iter != vec.end(); ++iter){
    int val = *iter;
    cout << val << ' ';
}

for (int val : vec)
    cout << val << ' ';
```

Ranged for support – cont.

We start by using a pointer as an iterator (later we will implement an iterator class)

Thus, to support ranged for loops

- The Vector class must provide begin() and end() functions that return an iterator (a pointer in our case)
  - One version that is a getter (returns a const int*)
  - Another version that is a setter (returns int*)

- The iterator must be able to support the following:
  - Increment and decrement the iterator
  - Compare operations
  - Dereference operations

  - All 3 are readily available for pointers

Operator overloading continued

- Can't change the meaning of operators for built-in types
  1 + 1 always equals 2
- can't create new operators
  2 ** 3 does not exist in C++
- Cannot overload ternary "conditional" operator
  test() ? a : b
- Cannot change the order of precedence or the associativity rules.
- Cannot change the arity of operators, e.g.:
  - “<<“ is always binary
  - “!” is always unary

Operator overloading continued

- 2 ^ 3 → That's bitwise xor, not exponentiation.

- Short circuiting behaviour:
  f() && g()
  f() || g()

- The output operator for built-in types is overloaded as a member function:

```
cout << x;              //is equivalent to
cout.operator<<(x);
```

What to return? Reference or value

- You are NOT required to return the same kind of result as the built-in operator (that operates on intrinsic data types).
- You are not required to implement a behavior similar to a built-in operator.

- However, it is STRONGLY recommended that you do so (unless you have a good reason not to)

Returning a value

- Some built-in operators return a value,
  - They **may or may not modify the operands**, e.g.
    - Binary operators +, -, *, /, ==, <, etc.
    - Unary operator- (e.g. x= -4;) , ++ (post-increment), etc.

  - When chained (in a compound expression), the return value is considered an intermediate variable, e.g.,

```
int x=4,y=5,z=0;
z = x+y+z;        // executes at run-time as: temp=x+y; z=temp+z;
                  // temp is the intermediate variable
```

  - The return value cannot be on the left side of an assignment operator, e.g.,

```
int x=4,y=5,z=0;
(x+y) = z;        // compilation error – intermediate
                  // variables (aka rvalue) cannot be assigned!
```

Returning a reference

- Other built-in operators return a **reference**,
  - They **typically modify an operand**, e.g.
    - Binary operators <<, >>, =
    - Unary operators ++ (pre-increment), -- (pre-decrement), +=, etc.

  - When chained (in a compound expression), the returned reference can be used in the next sub-expression, e.g.,

```
int x=4,y=5,z=0;
x=y=z;                  // executes at run-time as: y=z; x=y;
                        // The first assignment (y=z) results in y getting the value of z and
                        //   returns (and replaced by) a reference to y
                        // The next assignment (x=y) results in x getting the value of y which is
                        //   also the value of z.
```

  - The return value can be on the left side of an assignment operator; a reference to variable is treated as if it's that variable, e.g.,

```
int x=4,y=5,z=0;
(x=y) = z;  // okay!
            // (x=y) returns a reference to x which, just like any other variable,          //
            // can be on the left of an assignment operator  (aka lvalue),
```

Example - addition

Example – addition operator:

```
int x=4,y=5,z=0;
z=x+y;
```

- Values of x and y are not changed
- Value of z changes to (x+y)

Thus:

- Returns a **value** that's equal to the sum of lhs and rhs
  - Shall not return a reference
- Shall NOT change the values of lhs or rhs

Example – post-increment

Example : **post-increment**:

```
int x=4,z=0;
z=x++;
```

- Value of x changes to 5 (i.e. x+1)
- Value of z changes to 4 (i.e. original value of x)

Thus:

- Return a **value** that's equal to the original value of operand
  - Shall not return a reference
- Changes the values of operands

Example – pre-increment

Example: **pre-increment**:

```
int x=4,z=0;
z=++x;
```

- Value of x changes to 5 (i.e. x+1)
- Value of z changes to 5 (i.e. new value of x)

Thus:

- Returns a **reference** to operand
- Changes the values of operands

Example – operator += (combination op.)

Example : **operator+=**

```
int x=4,y=5,z=0;
z=x+=7;                    // translates to x=x+7; z=x; i.e. right-to-left
```

- Value of x changes to 11 (i.e. x+7)
- Value of z changes to 11 (i.e. new value of x)

Thus:

- Returns a **reference** to operand
- Changes the values of operands