OPERATOR OVERLOADING

FUNCTION OVERLOADING

Purpose: specify more the one definition for a function name

Example:

```
int sum(int a, int b) { return a+b; }
int sum(int a, int b, int c) { return a+b+c; }
double sum(double a, double b) { return a+b; }

sum(5, 6);        ← call the first function
sum(3, 4, 5);     ← call the second function
sum(5.3, 9.8);    ← call the third function
```

Rule: the number and type of parameters determine the function call

OPERATORS

Operator: a symbol for a function call such as +, ==, !

Operand: an input for an operation such as 5, ”hi”, a

Examples:

-5	negate an integer operand
7-5	subtract integer operands
5+7	sum integer operands
“Hi” + “There”	concatenate two string operands

Rules: operators act upon operands
operation type is determined from the operator and operands

OPERATOR TYPES

Unary:	operators with one operand (-, ++, !...)
Binary:	operators with two operands (-, ==, >, &&…)
Arithmetic:	operators that perform common math operations (+, -, /, %...)
Assignment:	operators that store data in variables (=, +=, %=…)
Relational:	operators that compare values (==, >, <=…)
Logical:	operators that execute logic (!, ||, &&…)

OPERATOR OVERLOADING

Purpose: redefine what an operator does for objects of a specific class

Example:

5+7 == 12 — sum two integers

"5"+"7" == "57" — concatenate two strings

\+ operator adds when operands are integers

\+ operator concatenates when operands are strings

the + operator is overloaded for the string class

Rule: overloaded operators must be useable in an expected manner

OPERATOR OVERLOADING METHODS

member: overload as a member function
calling object is first operand, parameter object is second
best method to preserve encapsulation

non-member: overload as a non-member function
both operands are parameter objects
facilitate implicit conversion

friend: overloaded as a non-member friend function
both operands are parameter objects
friend functions have direct class access like members

UNARY MEMBER OVERLOAD

Example: - unary operator negates a single operand

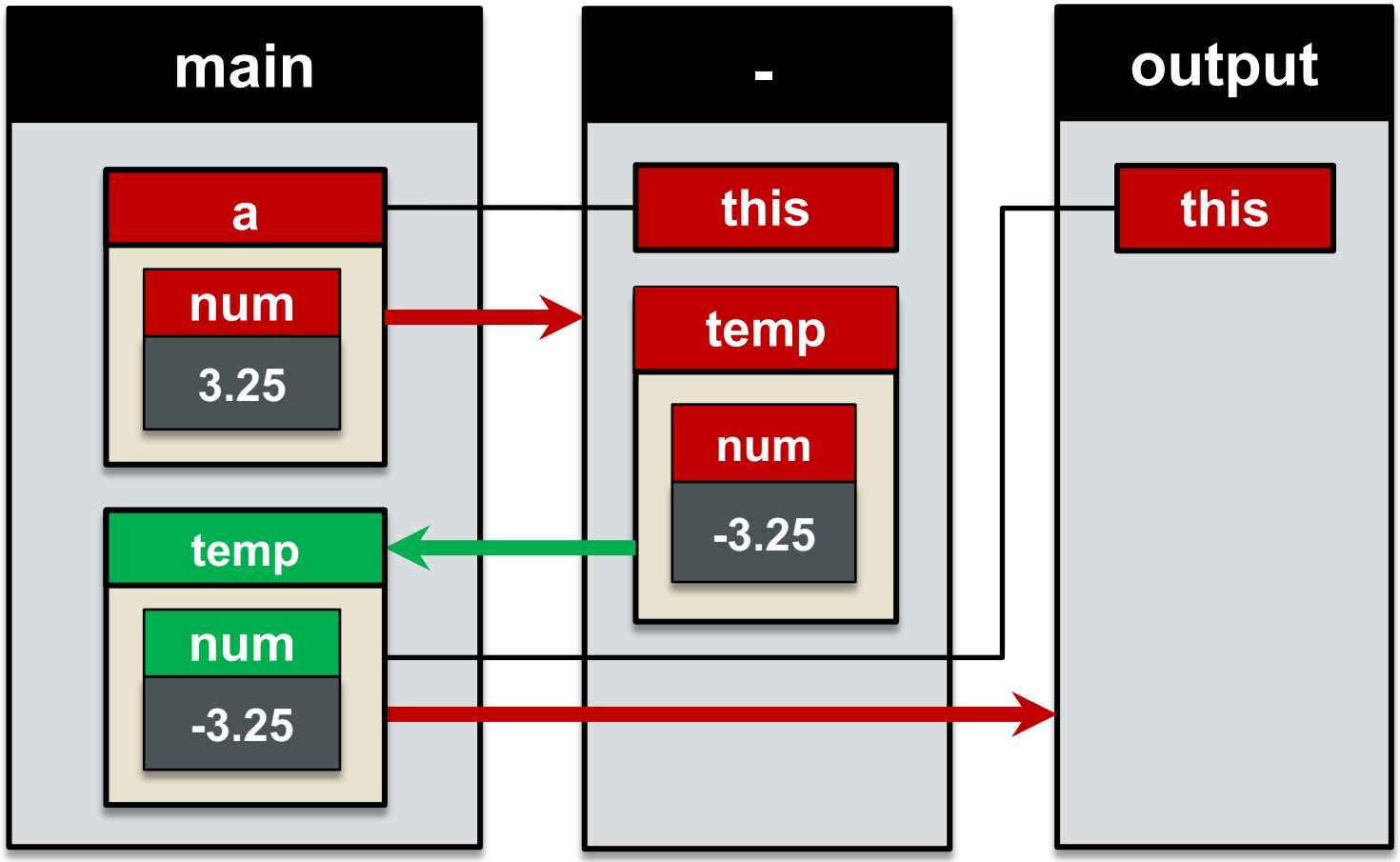
Usage:

```
Number a{3.5};      instantiate Number with value 3.5
cout << -a;         prints -3.5
cout << a;          prints 3.5
```

Notes:

1. - has been overloaded for class Number
2. a is the only operand so this is a unary - operation
3. - does not modify the original value of a

(-a).output()



1 a calls -
2 temp created from a.num
3 return temp by value
4 temp calls output
5 output prints temp.num

(note: diagram is somewhat simplified)

UNARY MEMBER OVERLOAD

Definition:

```
Number operator-( ) const {
    return Number(-num);
}
```

Notes:

1. member function: a calling object is required
2. no parameters: calling object only operand, unary overload
3. const member function: calling object cannot be modified
4. num is a data member of the calling object
5. -num implies that - is overloaded for the num variable type
6. Number(-num) creates a negated temporary object
7. returns the negated temporary object

SUBSCRIPT MEMBER OVERLOAD

Example: [] operator returns data from within an object

Usage:

```
Set s{};          instantiate Set
s[0] = 5;         assign a data member of s to 5
cout << s[0];     return the value of a data member of s
```

Notes:

1. the 0 in [0] is an index value, just like an array
2. the meaning of the index value must be defined
3. [] can read or write Set object data
4. for const Set objects, [] is read only

SUBSCRIPT MEMBER OVERLOAD

Definition:

```
int& operator[](int index) {
    assert(index >=0 && index <=4);
    return values[index];
}
```

Notes:

1. member function: a calling object is required
2. one parameter: two operands, binary overload
3. assert validates index's range as specified by coder
4. returns data at values[index] by reference
5. return type must match values type
6. must return by reference to be writable

SHORTCUT MEMBER OVERLOAD

Example: += operator assigns the sum of both operands

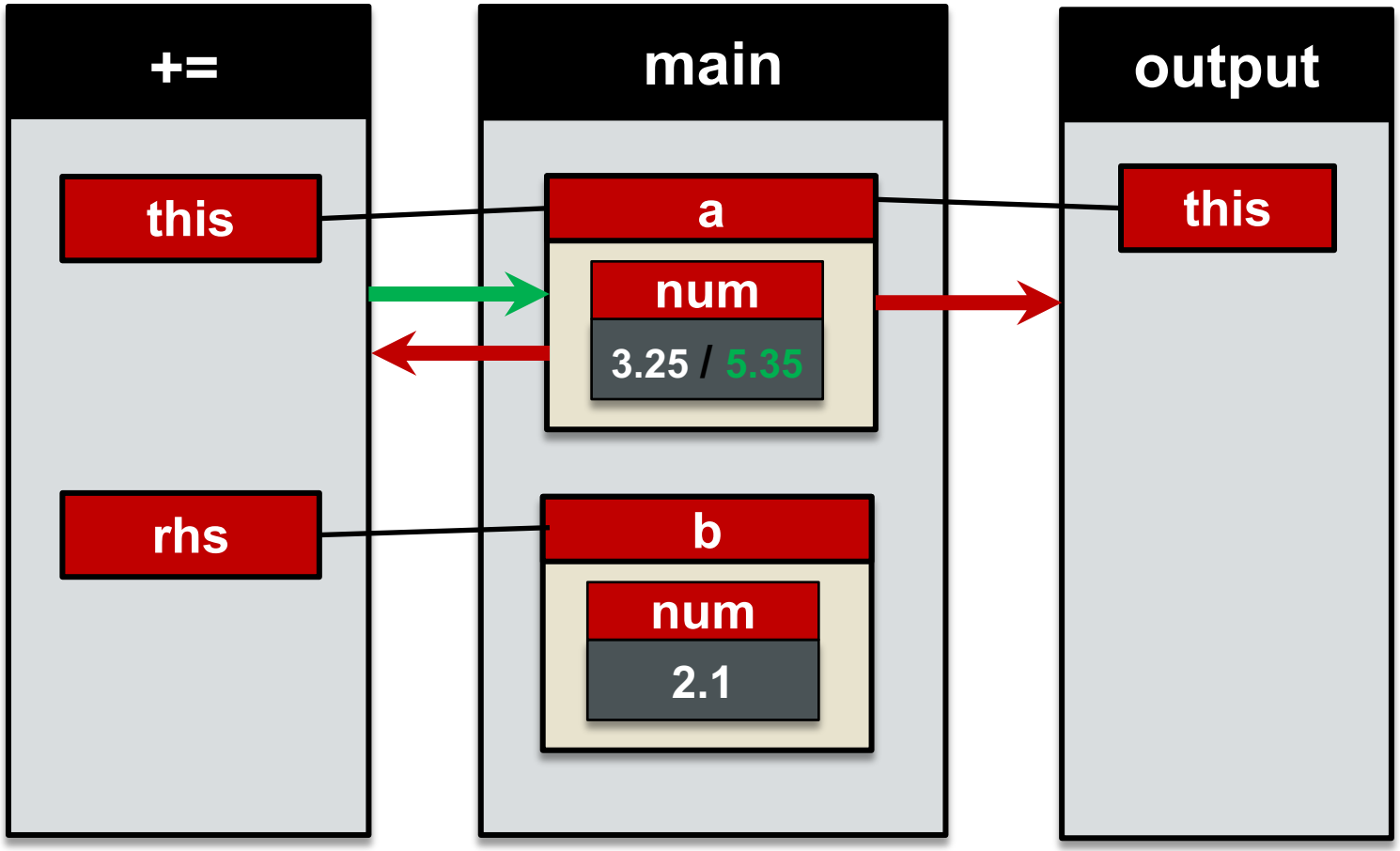
Usage:

```
Number a(3.25), b(2.1);     instantiate two Number objects
cout << (a+=b);             assign sum of two Number objects
cout << a;                  a is modified due to assignment
cout << b;                  b is not modified
```

Notes:

1. += has been overloaded for class Number
2. member function: a is calling object, b parameter object
3. a is modified since it is assigned the sum
4. b is not modified
5. the function returns the modified object a

(a+=b).output()



1 a calls += with b parameter
2 sum num and rhs.num
3 store sum in a
4 += returns modified a
5 a calls output
5 output prints a.num

(note: diagram is somewhat simplified)

SHORTCUT MEMBER OVERLOAD

Definition:

```
Number& operator+=(const Number &rhs) {
    num += rhs.num;
    return *this;
}
```

Notes:

1. num is a data member of the calling object
2. rhs.num is a data member of the parameter object
3. num += rhs.num stores their sum of into num
3. *this is the calling object of the function
4. returns a reference to the modified calling object

RELATIONAL MEMBER OVERLOAD

Example: == operator compares operands for equivalence

Usage:

```
Number a(3.25), b(2.1);     instantiate two Number objects
cout << (a==b);             compare two Number objects
```

Notes:

1. == has been overloaded for class Number
2. member function: a is calling object, b parameter object
3. a and b are not modified
4. function expected to return true or false

RELATIONAL MEMBER OVERLOAD

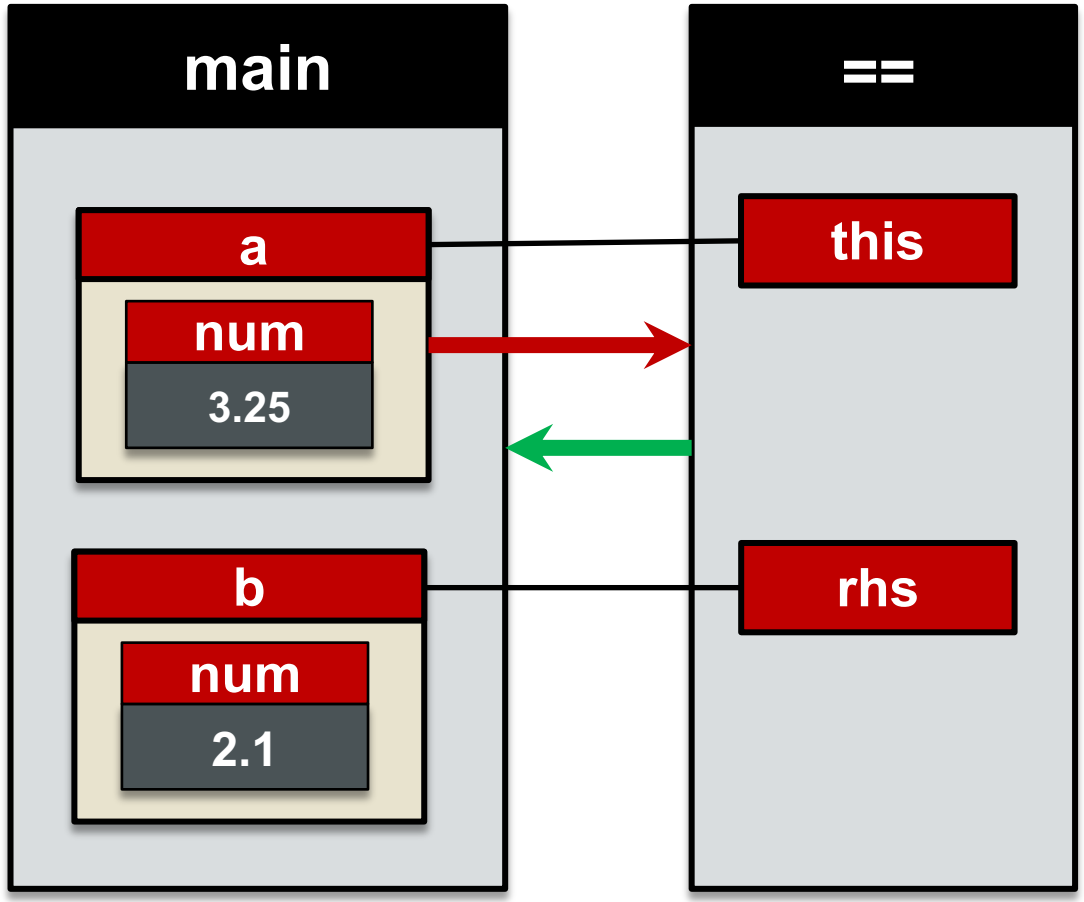
Definition:

```
bool operator==(const Number &rhs) const {
    return num == rhs.num;
}
```

Notes:

1. const member function: calling object cannot be modified
2. num is a data member of the calling object
3. rhs.num is a data member of the parameter object
4. returns true or false based upon equivalence

a==b MEMBER



1 a calls == with b parameter
2 num == rhs.num
3 == returns true or false

(note: diagram is somewhat simplified)

RELATIONAL NON-MEMBER OVERLOAD

Definition:

```
bool operator>(const Number &lhs, const Number &rhs) {
      return ( lhs.getNumber( ) > rhs.getNumber( ) );
}
```

Notes:

1. non-member function: both operands are parameters
2. left hand side is first parameter, right hand side second
3. accessor getNumber is required for non-member
4. returns true or false based upon equivalence

FRIEND RELATIONAL OVERLOAD

Declaration:

```
friend bool operator>(const Number &lhs, const Number &rhs);
```

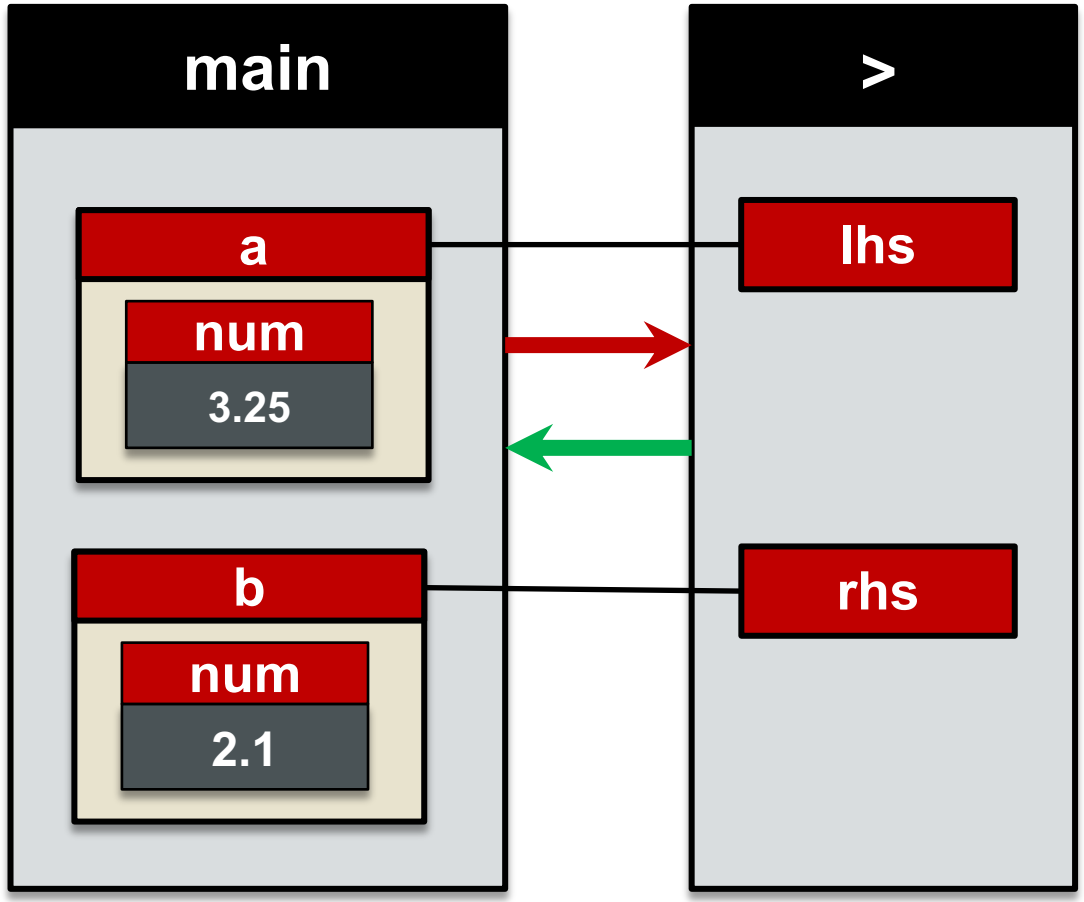
Definition:

```
bool operator>(const Number &lhs, const Number &rhs) {
    return ( lhs.num > rhs.num );
}
```

Notes:

1. non-member function with member level access
2. declared inline, defined out-of-line
3. lhs.num > rhs.num creates a temporary object
4. accessor getNumber is not required

a>b NON-MEMBER



1 > called with a and b parameters
2 lhs.getNum > rhs.getNum
3 > returns true or false

(note: diagram is somewhat simplified)

ARITHMETIC MEMBER OVERLOAD

Example: + operator returns the sum of both operands

Usage:

```
Number a(3.25), b(2.1);      instantiate two Number objects
cout << (a+b);               sum the two Number objects
cout << (a+b+a+c);           sum a chain of Number objects
a.output();                  operands are not modified
```

Notes:

1. + has been overloaded for class Number
2. member function: a is calling object, b parameter object
3. a and b are not modified
4. the function returns a new object, the sum of a and b

ARITHMETIC MEMBER OVERLOAD

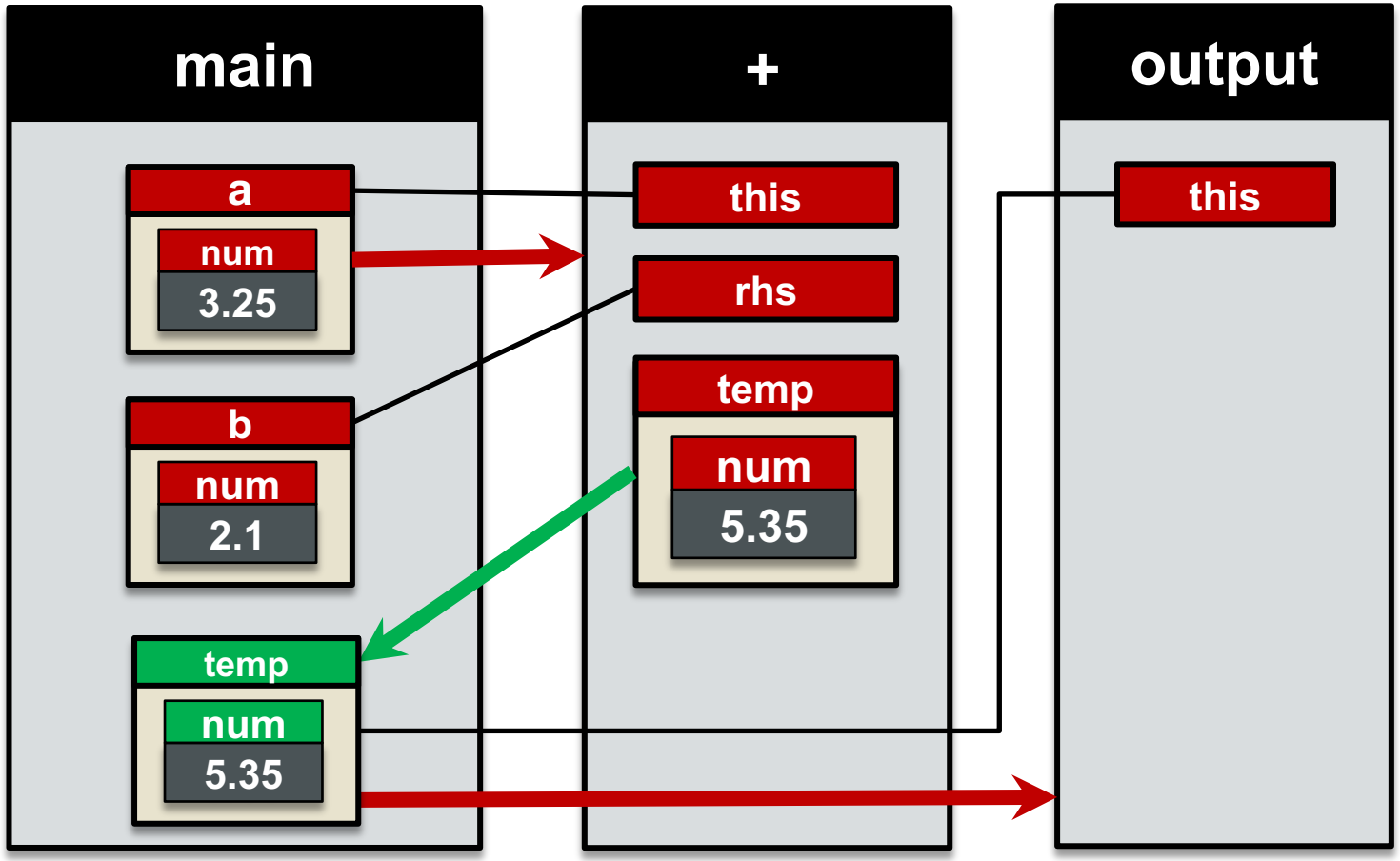
Definition:

```
Number operator+(const Number &rhs) const {
    return Number(num + rhs.num);
}
```

Notes:

1. const member function: calling object cannot be modified
2. num is a data member of the calling object
3. rhs.num is a data member of the parameter object
4. Number(num + rhs.num); creates a sum temporary object
5. returns a copy of the sum temporary object

(a+b).output()



1 a calls +
2 sum num and rhs.num
3 create temp from sum
4 + returns temp
4 temp calls output
5 output prints temp.num

(note: diagram is somewhat simplified)

ARITHMETIC NON-MEMBER OVERLOAD

Definition:

```
Number operator*(const Number &lhs, const Number &rhs) {
    return Number( lhs.getNumber( ) * rhs.getNumber( ) );
}
```

Notes:

1. non-member function: both operands are parameters
2. left hand side is first parameter, right hand side second
3. Number(lhs.getNumber( ) * rhs.getNumber( ))
   creates a temporary object
4. accessor getNumber is required for non-member
5. returns a copy of the temporary object

FRIEND ARITHMETIC OVERLOAD

Declaration:

```
friend Number operator*(const Number &lhs, const Number &rhs);
```

Definition:

```
Number operator*(const Number &lhs, const Number &rhs) {
    return Number( lhs.num * rhs.num );
}
```

Notes:

1. non-member function with member level access
2. declared inline, defined out-of-line
3. Number(lhs.num* rhs.num) creates a temporary object
4. accessor getNumber is not required

AUTOMATIC TYPE CONVERSION

Purpose: implicit conversion of parameter data from one type to another

Example: double sum(double a, double b) { return a + b; }

```
sum(5.5, 10.3)   ← normal function call
sum(5, 7);       ← integers are implicitly casted to doubles
sum("5", "7")    ← compiler error:
                     string can't automatically  convert to double
```

Benefit: legally convertible types can be used as parameters

AUTOMATIC TYPE CONVERSION MEMBER

Example:

```
Number(double n);      ← matching constructor for conversion
Number operator+(const Number &rhs);
void output( ) const;

Number a(3.25), b(2.1);

a+b      ← a is the calling object, b is a parameter object
a+5      ← legal, because 5 is a parameter
5+a      ← not legal, because 5 is a calling object
```

Note: a matching constructor is required for 5 to convert to Number, 5 converts to double, which is used to construct Number

AUTOMATIC TYPE CONVERSION NON-MEMBER

Example:

```
Number(double n);      ← matching constructor for conversion
Number operator+(const Number &lhs, const Number &rhs);
void output( ) const;

Number a(3.25), b(2.1);

a+b      ← a and b are parameter objects
a+5      ← legal, because 5 is a parameter
5+a      ← legal, because 5 is a parameter
```

Note: friend functions recommended for automatic type conversion

INSERTION OPERATOR OVERLOAD

Example: << operator builds an output stream

Usage:

```
Number a(3.25), b(2.1);     instantiate two Number objects
cout << a;                  output one Number object
cout << a << b;             output multiple Number objects
```

Notes:

1. << has been overloaded for class Number
2. a and b are not modified
4. function returns a stream object containing the Number

INSERTION OPERATOR OVERLOAD

Declaration:

```
friend ostream& operator<<(ostream &out, const Number &n);
```

Definition:

```
ostream& operator<<(ostream &out, const Number &n) {
    out << n.num;
    return out;
}
```

Notes:

1. friend function
2. ostream object is first parameter, Number second
3. Number is a constant parameter
4. out << n.num requires that << is overloaded for num
5. out << n.num inserts num into the stream object
6. modified ostream object returned by reference

EXTRACTION OPERATOR OVERLOAD

Example: >> operator builds an input stream

Usage:

```
Number a{}, b{};        instantiate two Number objects
cout >> a;              input into one Number object
cout >> a >> b;         input into multiple Number objects
```

Notes:

1. >> has been overloaded for class Number
2. a and b are modified
3. function returns a stream object containing the Number

EXTRACTION OPERATOR OVERLOAD

Declaration:

```
friend istream& operator>>(istream &in, Number &n);
```

Definition:

```
istream& operator>>(istream &in, Number &n); {
    in >> n.num;
    return in;
}
```

Notes:

1. friend function
2. istream object is first parameter, Number second
3. in >> n.num requires that >> is overloaded for num
4. in >> n.num inserts last input into num
5. Number is modified by this function

PREFIX OPERATOR OVERLOAD

Example: prefix ++ increments an operand

Usage:

```
Number n{5};          instantiate two Number objects
cout >> ++n;          prefix increment Number
```

Notes:

1. n is modified
2. function returns a modified Number

PREFIX OPERATOR OVERLOAD

Definition:

```
Number& operator++( ) {
    ++num;
    return *this;
}
```

Notes:

1. member function: calling object is single operand
2. prefix ++ must be overloaded for num
3. return modified Number object by reference

POSTFIX OPERATOR OVERLOAD

Example: postfix ++ increments an operand

Usage:

```
Number n{5};          instantiate two Number objects
cout >> n++;          postfix increment Number
```

Notes:

1. n is modified
2. function returns a copy of Number before increment

POSTFIX OPERATOR OVERLOAD

Definition:

```
Number operator++( int) {
    Number temp{num};
    ++num;
    return *this;
}
```

Notes:

1. member function: calling object is single operand
2. compiler uses int parameter to tell which ++ is postfix
3. temp copy stores original Number state
4. prefix ++ must be overloaded for num
5. return a copy of original Number object
6. Number has been modified to store incremented value