void push_back (const value_type& val);

vector::emplace_back()

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    vector<string> v = { "Albert","Michael"};

    v.push_back("Peter");
    v.emplace_back("Roger");

    cout << "v contains:";
    for (const string& x : v) {
        cout << ' ' << x;
    }
    cout << endl;

    return 0;
}
```

```
v contains: Albert Michael Peter Roger
```

```
void push_back (const value_type& val);
```

- Vector::push_back() constructs an object outside the vector THEN copies the object to the inside of the vector.
- vector::emplace_back(), on the other hand, constructs a string object in-place (inside the vector).
- Thus with large objects, it's more efficient to use vector::emplace_back() to avoid unnecessary copying

Level	Operator	Associativity
1	::	Left-to-right
2	Unary postfix ++, --, [], (), ., ->	Left-to-right
3	Unary prefix ++, -- , +, -, ~, !, *, &, (type), sizeof, new, delete	Right-to-left
4	. *, ->*	
5	*, / , %	Left-to-right
6	+, -	Left-to-right
7	>>, <<	Left-to-right
8	<=, >=	Left-to-right
9	==, ~=	Left-to-right
10	&	Left-to-right
11	!	Left-to-right
12	^	Left-to-right
13	&&	Left-to-right
14	||	Left-to-right
15	=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=, ?: , throw	Right-to-left

Operator precedence – cont.

- If two or more operators are used I the same expression, then operator precedence is determined by the ***priority level***.

- If they have the same priority level, then precedence is determined by the ***associativity rule*** (third column on previous table, which determines whether precedence is from left-to-right or right-to-left).

- Enclosing all sub-statements in ***parentheses*** (even those unnecessary because of their precedence) improves code readability.

Operator precedence – cont.

Example:

```
x = 5 + 7 % 2 ;
```

Operator precedence – cont.

Example:

```
x = 5 + 7 % 2; // x = 6
```

Operator precedence – cont.

Example:

```
x = 5 + (7 % 2); // x = 6 (same as without parenthesis)
x = (5 + 7) % 2; // x = 0
```

Non-member (free) function Overloading

```
void setCost(double);
void setCost(char);
```

- Same function name, but different parameters

- Compiler chooses the correct overload based on the **declared type(s)** of the arguments you pass it during a function call.
  - Resolved at compile-time.

- If ambiguous → compilation error (two overloads, one takes a char, another takes an int, and you invoke the function with a value of 5!)

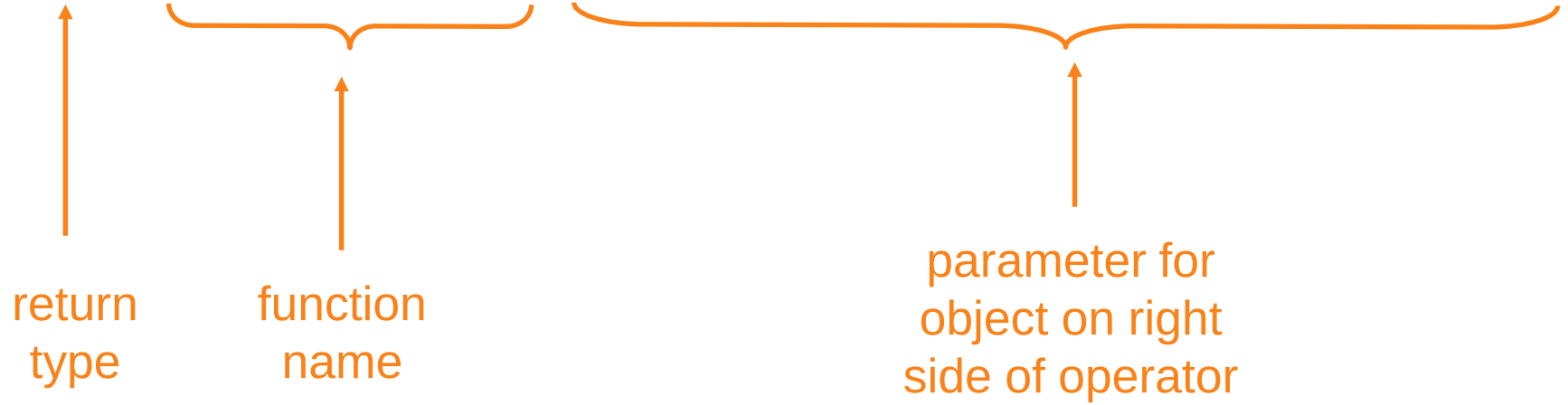
Operator overloading

- Operators such as =, +, and others can be redefined when used with newly defined types (i.e. classes)

- 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

Operator overloading – cont.

```
void operator=(SomeType lhs, SomeType rhs)
```



Operator overloading – cont.

- Operator can be invoked as a function:

```
operator=(object1, object2);
```

- It can also be used in more conventional manner:

```
object1 = object2;
```

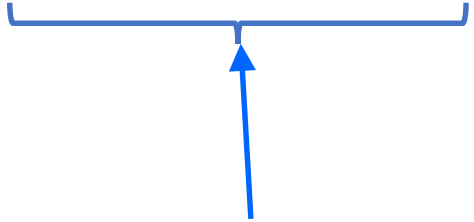
- The following two lines are similar:

```
x = y + z;
x = operator+(y,z);
```

Operator overloading – chaining

- **Return type the same as the left operand** supports **chaining**:

```
  object1 = object2 = object3;
  object1 << object2 << object3;
Ex:
cout << obj1 << obj2;  //evaluated left-to-right
```



```
// "cout<<obj1" is evaluated – return is of type
  ostream&

// The returned ostream& ref is evaluated as the left
  operand in <<obj2
```

Member function overloading

- Constructors may be overloaded

```
class Person{
public:
    Person(const string& aname): name(aname){}
    Person():name("unnamed"){}
Private:
    string name;
}
```

- Non-constructor member functions can also be overloaded:

```
class Person{
public:
    void Initialize(const string& aname){name=aname;}
    void Initialize(){name="unnamed";}
Private:
    string name;
}
```

- Must have unique parameter lists as for constructors

Overloading operators as member functions

Two options:

- Overloaded operator may be defined inline within the class definition, OR
- Prototyped inside the class, whereas the operator function definition goes outside (using the scope operator)

Overloading operators as member functions

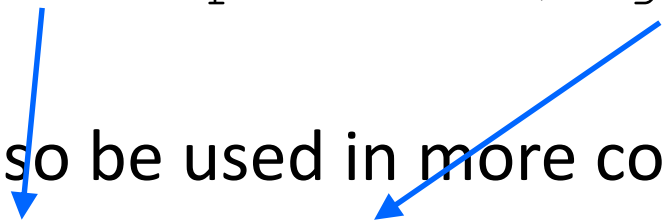
```
void operator=(const SomeType &rval)
```

- **Operator is called via object on left side**

Overloading operators as member functions - invocation

- Operator can be invoked as a member function:

```
object1.operator=(object2);
```



- It can also be used in more conventional manner:

```
object1 = object2;
```

Operator overloading – General Notes

- Overloaded functions (i.e. functions having the same name) can be resolved using input parameters (number and/or type), **but NOT using** the return type.

- Can change meaning of an operator

- Cannot change the number of operands of the operator

- Only certain operators can be overloaded. Cannot overload the following operators:

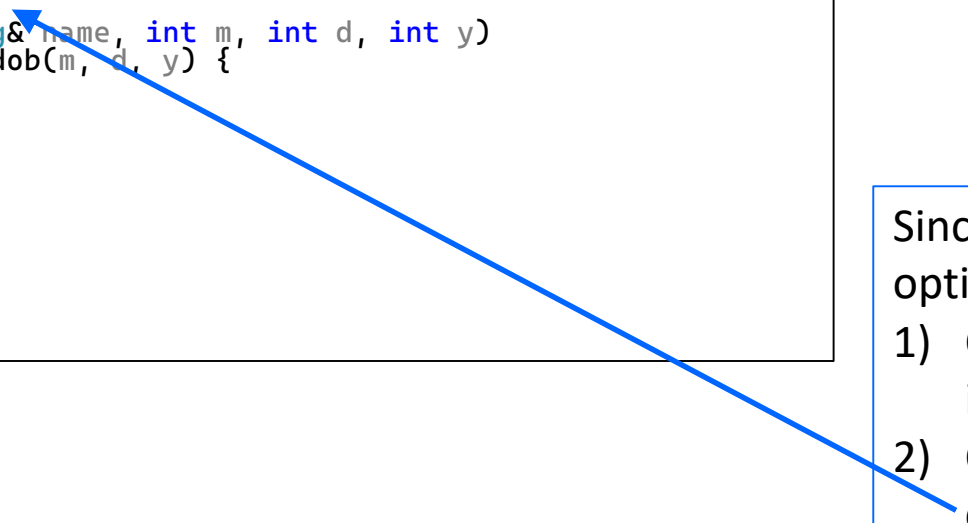
```
?:   .   :: sizeof
```

Overloading in nested classes - ex 1

```
#include <iostream>
#include <string>
using namespace std;

class Date {
    friend ostream& operator<<(ostream& os, const Date& rhs);
public:
    Date(int m, int d, int y) : month(m), day(d), year(y) { }
private:
    int month, day, year;
};


class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs);
public:
    Person(const string& name, int m, int d, int y)
        : name(name), dob(m, d, y) {
    }
private:
    string name;
    Date dob;
};
```



Since cout is of type ostream, then we have two options to overload the output operator <<:

1) Create within the class ostream → But this is inside the standard library
2) Create as a free function ← this is what we choose!

Overloading in nested classes – ex 1 (cont.)

```
ostream& operator<<(ostream& os, const Date& rhs) {
    os << rhs.month << '/' << rhs.day << '/' << rhs.year;
    return os;
}

ostream& operator<<(ostream& os, const Person& rhs) {
    os << "Person: name = " << rhs.name << ", dob = " << rhs.dob;
    return os;
}


int main() {
    Person john("John", 7, 14, 1920);
    cout << john << endl;
}
```

Person: name = John, dob = 7/14/1920

Since cout is of type ostream, then we have two options to overload the output operator << :

1) Create within the class ostream → But this is inside the standard library
2) Create as a free function ← this is what we choose!

Overloading in nested classes – ex 2

```
#include <iostream>
#include <string>
using namespace std;

class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs);
public:
    Person(const string& name, int m, int d, int y)
        : name(name), dob(m, d, y) {
    }
private:
    class Date {
        friend ostream& operator<<(ostream& os, const Date& rhs);
    public:
        Date(int m, int d, int y) : month(m), day(d), year(y) { }
    private:
        int month, day, year;
    };
    string name;
    Date dob;
};
```

Overloading in nested classes – ex 2 (cont.)

```
// Note the use of Person::Date
ostream& operator<<(ostream& os, const Person::Date& rhs)
{
    os << rhs.month << '/' << rhs.day << '/' << rhs.year;
    return os;
}

ostream& operator<<(ostream& os, const Person& rhs) {
    os << "Person: name = " << rhs.name << ", dob = " <<
rhs.dob;
    return os;
}

int main() {
    Person john("John", 7, 14, 1920);
    cout << john << endl;
}
```

Compilation error ---- why?

Overloading in nested classes – ex 2 - fixed

```
#include <iostream>
#include <string>
using namespace std;

class Person {
    friend ostream& operator<<(ostream& os, const Person& rhs);
public:
    class Date {
        friend ostream& operator<<(ostream& os, const Date& rhs);
    public:
        Date(int m, int d, int y) : month(m), day(d), year(y) { }
    private:
        int month, day, year;
    };
public:
    Person(const string& name, int m, int d, int y)
        : name(name), dob(m, d, y) {
    }
private:
    string name;
    Date dob;
};
```

Note that now we declared “Date” as public

dob remains private

Overloading in nested classes – ex 2 - fixed

```
// Note the use of Person::Date
ostream& operator<<(ostream& os, const Person::Date& rhs)
{
    os << rhs.month << '/' << rhs.day << '/' << rhs.year;
    return os;
}

ostream& operator<<(ostream& os, const Person& rhs) {
    os << "Person: name = " << rhs.name << ", dob = " <<
rhs.dob;
    return os;
}

int main() {
    Person john("John", 7, 14, 1920);
    cout << john << endl;
}
```

Person: name = John, dob = 7/14/1920

Date must be declared public in order for this function to be able access the Person::Date datatype.

Overloading in nested classes – ex 3

```
#include <iostream>
#include <string>
using namespace std;

class Person {
    // op<< for a Person is a friend of the Person class
    friend ostream& operator<<(ostream& os, const Person& rhs);

    class Date {
        // op<< for a Date is a friend of the Date class
        friend ostream& operator<<(ostream& os, const Date& rhs) {
            os << rhs.month << '/' << rhs.day << '/' << rhs.year;
            return os;
        }
    public:
        Date(int m, int d, int y) : month(m), day(d), year(y) { }
    private:
        int month, day, year;
    };

public:

    Person(const string& name, int m, int d, int y)
        : name(name), dob(m, d, y) {
    }
private:
    string name;
    Date dob;
};
```

- The operator<< is still a free function (not a member function of Date)
  - It's just defined inline
  - This is because we're still using the "friend keyword"
- But it is a friend of class Date

Overloading in nested classes – ex 3 – cont.

```
ostream& operator<<(ostream& os, const Person& rhs) {
    os << "Person: name = " << rhs.name << ", dob = " << rhs.dob;
    return os;
}

int main() {
    Person john("John", 7, 14, 1920);
    cout << john << endl;
}
```

```
Person: name = John, dob = 7/14/1920
```

Overloading in nested classes – ex 4

```
#include <iostream>
#include <string>
using namespace std;

class Person {
    // op<< for a Person is a friend of the Person class
    friend ostream& operator<<(ostream& os, const Person& rhs);

    class Date {
    // op<< for a Date is a friend of the Date class
        friend ostream& operator<<(ostream& os, const Date& rhs);
    public:
        Date(int m, int d, int y) : month(m), day(d), year(y) { }
    private:
        int month, day, year;
    };

    friend ostream& operator<<(ostream& os, const Person::Date& rhs);

public:

    Person(const string& name, int m, int d, int y)
        : name(name), dob(m, d, y) {
    }
private:
    string name;
    Date dob;
};
```

This overloaded free function needs to be friends of both Person and Date so that it can access the Person::Date (private type) and also Date::month/day/year (private data member of Date class)

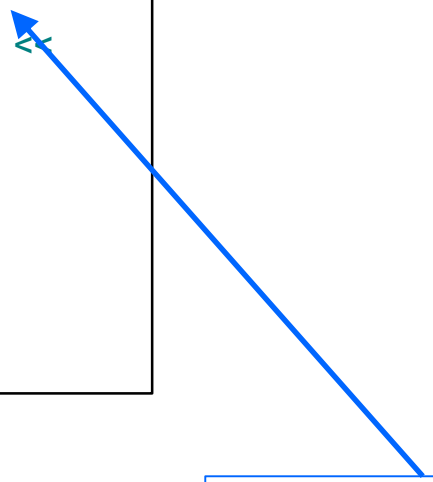
Overloading in nested classes – ex 4 – cont.

```
ostream& operator<<(ostream& os, const Person::Date& rhs) {
    os << rhs.month << '/' << rhs.day << '/' << rhs.year;
    return os;
}

ostream& operator<<(ostream& os, const Person& rhs) {
    os << "Person: name = " << rhs.name << ", dob = " <<
rhs.dob;
    return os;
}

int main() {
    Person john("John", 7, 14, 1920);
    cout << john << endl;
}
```

```
Person: name = John, dob = 7/14/1920
```



Can access the private nested class Person::Date because this function was declared to be a friend of Person.