# Operator Overloading

# Review: what are operators?

| | |
|---|---|
| grouping operator: | () |
| array access operator: | [] |
| member access operator: | . |
| unary postfix operators: | ++ -- |
| unary prefix operators: | ++ -- + - ! |
| binary arithmetic operators: | * / % + - |
| binary relational operators: | < <= >= > == != |
| binary logical operators: | && \|\| |
| assignment operators: | += -= *= /= %= = |
| I/O operators*: | << >> |

## Operators in C++ are actually functions!

```
5 + 4  ~  add(5, 4)
```

And we know that we can overload functions...

## Which means that we can overload operators, too!

(this is a really cool feature of C++)

# What's operator overloading all about?

C++ comes with predefined behaviors for operators…

- the addition operator (+) adds two numbers and returns the result
- the << and >> operators actually do bitwise shifting of numbers by default

We can add additional behavior via operator overloading

- this allows us to add convenient class-related functionality to certain operators
- for example, the string class overloads the addition operator to concatenate (join) two strings
- the ostream and istream classes overload the << and >> operators to do output and input, respectively
- these are very useful and convenient additions to these classes!

## Two Different Approaches

This class will teach you two approaches to operator overloading:

- global operators implemented *without* privileged access to the class, and
- friend operators (which are also written in the global scope) that have privileged access to private properties of the class

Why you might use one instead of the other:

- the first approach is generally used by someone *other* than the author of the class
- the second is generally used by the author of the class

For this class:

- most of our operators will be implemented as friend-ed operators, unless otherwise specified (like on today's assignment >.>)

# Global Operators

This will work like we expect:

```
// displays “Hi, I am 23 years old”
cout << “Hi, I am ” << 23 << “ years old.\n”;
```

But have you ever tried to do this?

```
string text  = “Hi, I am ”, label = “ years old.\n”;


// this doesn’t work… why not?
cout << ( text + 23 + label );

         error: no match for ‘operator+’ in ‘text + 23’!
```

The + operator hasn’t been overloaded to handle (`string, int`)!

- so let’s teach it how! You’re excited… I can tell! : )

# Global Operators

The <u>function name</u> for an operator is:

```
operator X  // where X is the operator you want to overload
```

General prototype syntax (for *binary* operators):

```
return_type operator X(arg_type LHS, arg_type RHS);
```

For our example, our prototype will look like this:

```
string operator +(const string& LHS, int RHS);
```

Just so you know: `// LHS + RHS  =>  operator +(LHS, RHS)`

- LHS means 'left-hand side' of the operator (will be used as first argument)
- RHS means 'right-hand side' of the operator (will be used as second argument)

# Global Operators

Example of non-class-author operator overloading:

```
// overloading the addition operator for strings and ints

string operator +(const string& LHS, int RHS) {

    string num;

    for (int n = RHS; n > 0; n /= 10) {   // for each digit in the number

        num = char('0' + n % 10) + num;   // convert to char and prepend to num

    }

    return LHS + num; // use preexisting operator +(string, string)

}


void somefunction() {

    string text  = "Hi, I am ", label = " years old.\n";

    cout << ( text + 23 + label ); // this now works! hooray!

}
```

# Global Operators

## Continuing the example:

```
// overloading the addition operator for strings and ints

string operator +(const string& LHS, int RHS); // assume we implement this
```

## Will this work?

```
void somefunction() {

    string text  = “Hi, I am ”, label = “ years old.\n”;

    cout << text << ( 23 + label );

}
                 error: no match for ‘operator+’ in ’23 + label’!
```

# Global Operators

Continuing the example:

```
// overloading the addition operator for strings and ints
string operator +(const string& LHS, int RHS); // assume we implement this
string operator +(int LHS, const string& RHS); // assume we implement this
```

Yay! It works!

```
void somefunction() {
    string text  = “Hi, I am ”, label = “ years old.\n”;
    cout << text << ( 23 + label ); // this now works! hooray!
}
```

Keep LHS and RHS in mind when working with binary operators:

- whatever is on the left side of the operator will be the first argument
- whatever is on the right side will be the second

# Global Operators

## Here’s a question for you:

```
// overloading the addition operator for strings and ints

string operator +(const string& LHS, int RHS); // assume we implement this
```

## Why did this work with just `operator +(string, int)` defined?

```
void somefunction() {

    string text  = “Hi, I am ”, label = “ years old.\n”;

    cout << ( text + 23 + label ); // this works! hooray!

}                    but shouldn’t this be an error?
```

## Remember how these operators actually get evaluated:

```
cout << (text + 23 + label); // text + 23         (string + int == okay)

cout << (temp_str1 + label); // temp_str1 + label  (string + string == okay)

cout << (temp_str2);         // cout temp_str2;
```

# Global Operators

## What we just did:

- we just overloaded the addition operator to be able to append or prepend a number to a `string`
- this is novel behavior that was *not* defined by the author of the `string` class
- we, as application developers, added this functionality because we needed it

## Operator functions have a few differences from other functions:

- their names follow a slightly different syntax (`operator X`)
- they are not called using parentheses, but by being placed between their arguments (`e.g. text + 23`)
- at least one of the arguments to the function MUST be a class object

## Other than that, these are just like other functions you've written

- actually, they're a whole lot cooler. :)

# Global Operators

Assume we created this class:

```
class BagOfMarbles {

    public:

        BagOfMarbles(int n);           // creates a bag with n marbles

        int get_num_marbles() const;   // getter for num_marbles


    private:

        int num_marbles;

};
```

Overload the addition operator to add two `BagOfMarbles` objects:

```
BagOfMarbles operator +(const BagOfMarbles& b1, const BagOfMarbles& b2) {

    // return a new bag containing all the marbles in b1 and b2

    return BagOfMarbles(b1.get_num_marbles() + b2.get_num_marbles());

}
```

## friend operators

Same class, but using a friend operator:

```
class BagOfMarbles {
    public:
        BagOfMarbles(int n);          // creates a bag with n marbles
        int get_num_marbles() const;  // getter for num_marbles
        friend BagOfMarbles operator +(const BagOfMarbles&, const BagOfMarbles&);
    private:
        int num_marbles;
};
```

Overload the addition operator to add two `BagOfMarbles` objects:

```
BagOfMarbles operator +(const BagOfMarbles& b1, const BagOfMarbles& b2) {
    // return a new bag containing all the marbles in b1 and b2
    return BagOfMarbles(b1.num_marbles + b2.num_marbles);
}
```

# friend operators

Use the overloaded operator (same effect either way):

```
int main() {
    BagOfMarbles my_bag(10), your_bag(30);

    // combine our two bags using the overloaded addition operator
    BagOfMarbles our_bag = my_bag + your_bag;

    // display the number of marbles in our combined bag:
    cout << our_bag.get_num_marbles() << endl; // displays 40

    return 0;
}
```

The difference between friend & non-friend operators is only in their implementations!

# friend operators

Implementation of operator as non-friend:

```
BagOfMarbles operator +(const BagOfMarbles& b1, const BagOfMarbles& b2) {
    // must use public get_num_marbles() method to access private ‘num_marbles’
    return BagOfMarbles(b1.get_num_marbles() + b2.get_num_marbles());
}
```

Implementation of operator as friend (note the difference):

```
BagOfMarbles operator +(const BagOfMarbles& b1, const BagOfMarbles& b2) {
    // can directly access the private ‘num_marbles’ property of b1 and b2!
    return BagOfMarbles(b1.num_marbles + b2.num_marbles);
}
```

To make it a friend, just add this prototype to the class declaration:

```
friend BagOfMarbles operator +(const BagOfMarbles&, const BagOfMarbles&);
```

# friend operators

## friend operators:

- are written just like global operators (friend operators *are* global operators)
- must be prototyped in the class declaration, prefixed with the friend keyword
- even though they're prototyped in the class declaration, they do not actually belong to the class that they're friends with. This means NO `Classname::` prefix!
- however, they can still access private properties and methods of objects belonging to that class. Friendship has its benefits!
- the implementation of the operator <u>should not</u> have the friend keyword!

## Be aware:

- there is no 'calling object' in the context of a friend operator function
- you can only access properties and methods of a class by using the dot operator on an object of the class (either passed as an argument or created locally in the function)

# Symmetric I/O Operators

Assume we want to add symmetric I/O operators to this class:

```
class Point {
    public:
        Point();                  // creates a point at the origin (0, 0)
        Point(int x, int y);   // creates a point with coordinates (x, y)
    private:
        int x_coord, y_coord;
};
```

We want our points to be output like this: `(x_coord, y_coord)`

- to be considered symmetric, our input operator (>>) must be able to read what is printed by the output operator (<<), resulting in an identical object

# Symmetric I/O Operators

Add friend operator prototypes to the class declaration:

```
class Point {
    public:
        Point();                // creates a point at the origin (0, 0)
        Point(int x, int y);   // creates a point with coordinates (x, y)
        // friend output operator
        friend ostream& operator <<(ostream& out, const Point& p);
        // friend input operator
        friend istream& operator >>(istream& in, Point& p);
    private:
        int x_coord, y_coord;
};
```

# Symmetric I/O Operators

For reference, here is the output operator’s prototype:

```
friend ostream& operator <<(ostream& out, const Point& p);
```

Here is a possible implementation for the output operator:

```
ostream& operator <<(ostream& out, const Point& p) {
    return out << “(“ << p.x_coord << “, ” << p.y_coord << “)”;
}
```

Notice that the friend keyword only appears in the prototype!

- however, because the operator is a friend of the Point class, its implementation can directly access p’s private `x_coord` and `y_coord` properties

## Symmetric I/O Operators

Here is a possible implementation for the input operator:

```
// reads a Point in the form (x, y). Assume that x and y cannot be negative
istream& operator >>(istream& in, Point& p) {
    int x, y;   // temporary variables to hold the coordinates
    char trash; // for the comma and parentheses, which aren’t needed

    // reads in “(x, y)” and ensures that x and y are not negative
    if (!(in >> trash >> x >> trash >> y >> trash) || x < 0 || y < 0) {
        in.setstate(ios::failbit); // if x or y are negative, set failure state
    } else {
        p.x_coord = x; // x and y are valid, so directly modify
        p.y_coord = y; // the x_coord and y_coord values of p
    }
    return in; // always return the stream object
}
```

# Symmetric I/O Operators

Three steps to follow for your input operations:

- first read the appropriate values from the input stream into temporary local variables (you may need some extra variables to store ‘trash’ values, such as the parentheses and comma in the previous example)
- if the input operation succeeded, ensure that the values read in make sense according to the rules of your class. If not, set the input stream into failure state and return the stream object
- only after you have checked that the input succeeded and that the values make sense should you modify the object