# 14.5 OPERATOR OVERLOADING, CONT'D

# LAST DAY TO DROP

Last day to withdraw (i.e., drop) from any class this semester is

## April 19th by 5pm

# QUIZ 6 - FRIDAY

Take quiz 6 online anytime between 12am and 11:59pm.

10 minutes once you start.

# WARM UP

- What is the `THIS` pointer?
  - Hidden pointer passed to member functions that references the entire object (instance of the class)
- How can we use the `THIS` pointer in class methods?
  - Make clear that instance/object data member is referenced

    ```
    this->width
    ```
- What is operator overloading?
  - defining a new meaning for the operator
  - Implementing the code for the operator as member function (method)
- Why is operator overloading useful/needed?
  - It provides a more natural way to use the objects

    ```
    inWallet + inBank
    ```

# OVERLOADING BINARY OPERATORS

```
// operator# used with someClass objects
// # can be +, -, *, /, ==, etc.
//
SomeClass leftobject, rightobject;

Rectangle box1, box2;

// overloaded binary operator + allows this syntax…
box1 + box2;

// compiler translates into a call to function operator+
leftobject.operator#(rightobject)
box1.operator+(box2);
```

# EXAMPLE: OPERATOR+ (AS MEMBER FUNCTION)

```
// Prototype
Rectangle operator+(const Rectangle &) const;

// Implementation
Rectangle Rectangle::operator+(const Rectangle &rightobj) const
{
   Rectangle temp;
   temp.length = this->length + rightobj.length;
   temp.width = this->width + rightobj.width;

   return temp;
}
```

```
Rectangle r1, r1;
r1 + r2
r1.operator+(r2);
```

# IN CLASS EXERCISE

Method Operator+ implementation

```
int x=10;
int y=20;
cout << x + y;
```

```
Rectangle Rectangle::operator+(const Rectangle &rect) const
{
   Rectangle temp;
   temp.length = this->length + rect.length;
   temp.width = this->width + rect.width;
   return temp;
}
```

length, width…are data members of the object on which this function, operator+, is being called

# IN CLASS EXERCISE

Method Operator== implementations

```
int x=10;
int y=20;
if (x > y)
```

```
bool Rectangle::operator==(const Rectangle &rect) const
{
   if (this->length == rect.length && this->width == rect.width)
        return true;
   else
        return false;
}
```

# NOTES ON OVERLOADED OPERATORS

Can change meaning of an operator

Cannot change the <u>number of operands</u> of the operator

Only certain operators can be overloaded.

Cannot overload the following operators:

| | |
|---|---|
| ?: | (conditional – ternary operator) |
| . | (member selection) |
| .* | (member selection with pointer-to-member) |
| :: | (scope resolution) |
| `sizeof` | (object size information) |

# 14.2 FRIENDS OF CLASSES

# FRIEND FUNCTIONS

Your friends can access your FB news feed, Not everyone.

Giving access to some specific function (or another class) outside of the class is better than giving access to everyone outside of the class…

Sometimes it is required…

# FRIENDS OF CLASSES

**Friend**: a function (or class) that is not a member of some class, but has access to the private members of some class

A friend function can be a stand-alone function or a member function of another class

*Friend* function is declared in the class definition with `friend` keyword before the function prototype

# FRIEND FUNCTION DECLARATIONS

Prototype for a friend that is a stand-alone, non-member function:

```
friend void myFriend(int);
```

Prototype for a friend that is a member function of another class:

```
friend void AnotherClass::myFriend(int);
```

# FRIEND EXAMPLE

```
class MyClass
{
    friend void printCnt(const MyClass &);
public:
    MyClass();
    void setStudents(int);
    int  getStudents() const;
private:
    int studentcnt;
};
```

const not allowed on non-member functions!

```
MyClass::MyClass()
{
    studentcnt=0;
}
void MyClass::setStudents(int studentcnt)
{
    this->studentcnt=studentcnt;
}
int  MyClass::getStudents() const
{
    return studentcnt;
}
void printCnt(const MyClass &object)
{
    cout << object.studentcnt << endl;
}
```

This direct access to data member is not normally allowed! Since this function is declared in the class definition as a friend, special access is given.

Note no class scope resolution since this is **NOT** a class member function!

# FRIEND EXAMPLE

```
int main()
{
    MyClass cs2020;
    cs2020.setStudents(95);

    cout << "Printing student count via the friend class. " << endl;
    cout << "Student count is ";
    printCnt(cs2020);       // non-member friend function being called
                            // able to access private data member studentcnt
    cout << endl;

    return 0;
}
```

This is how member functions are called, i.e., object is needed

This is how non-member functions are called, i.e., object is needed

# FRIEND CLASS DECLARATIONS

Class as a friend of a class:

```
class MyFriend
{
public:
     void member1(int);
     int  member2();
};

class MyClass
{
friend void MyFriend::member1(int);
friend int  MyFriend::member2();

public:
   // member functions
private:
   // data
};
```

# FRIEND CLASS DECLARATIONS

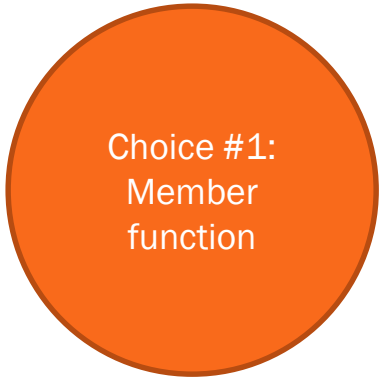
Class as a friend of a class:

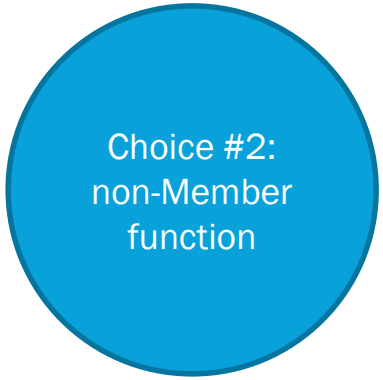
```
class MyFriend
{

};



class MyClass
{
friend class MyFriend;

public:
   // member fucntions
private:
   // data
};
```

Declares entire class, *MyFriend*, as a friend to class *MyClass*.

# OPERATOR OVERLOADING

Remember when I told you we have a choice of how to implement the operator overloading?




Non-member function implementation ➔ implementation as a FRIEND function.

# MEMBER VS. NON-MEMBER

```
Rectangle r1, r1;

r1 + r2

// implemented as member
// translated to…
//
r1.operator+(r2);
```

vs.

```
Rectangle r1, r1;

r1 + r2

// implemented as non-member
// translated to…
//
operator+(r1, r2);
```

# OVERLOADING BINARY OPS AS NON-MEMBER FUNCTIONS

```
// Generic Prototype
friend returnType operator#(const className &, const className &);

// Generic Implementation
returnType operator#(const className &left, const className &right)
{
    // algorithm to perform the operation
    return value;
}
```

# EXAMPLE: OPERATOR+ (AS NON-MEMBER FUNCTION)

```
// Prototype
friend Rectangle operator+(const Rectangle &, const Rectangle &);

// Implementation
Rectangle operator+(const Rectangle &left, const Rectangle &right)
{
    Rectangle temp;
    temp.length = left.length + right.length;
    temp.width = left.width + right.width;

    return temp;
}
```

```
Rectangle r1, r1;
r1 + r2

// non-member
operator+(r1, r2);
```

# OVERLOADING STREAM INSERTION & STREAM EXTRACTION

<< stream insertion operator

>> stream extraction operator

Want to be able to…

```
Rectangle box1(20, 10);
cout << box1;                    // show box to console window
outfile << box1;

operator<<(left, right);         // really how compiler sees it…
```

# OVERLOADING WITH FRIEND FUNCTION

GENERAL RULE:

If left hand side object is <u>not</u> of class type ➔ <u>must</u> implement the overloaded operator function as a friend function…

i.e., a non-member function!

```
cout << inWallet;
```

instance of ostream class

instance of Money class

# OVERLOADING STREAM INSERTION (<<)

```
// Generic Prototype
friend ostream& operator<<(ostream &, const className &);

// Generic Implementation
ostream& operator<<(ostream &osObject, const className &right)
{
    // local declaration, if any
    // output the members of right object to
    // osObject << …

    return osObject;
}
```

# EXAMPLE: OVERLOADED <<

```
// Prototype
friend ostream& operator<<(ostream &, const Rectangle &);

// Implementation
ostream& operator<<(ostream &left, const Rectangle &right)
{
    left << right.length << right.width;

    return left;
}
```

```
Rectangle r1, r1;
cout << r1;
// compiler sees as
operator<<(cout, r1);
```

# EXAMPLE: OVERLOADED <<

```
// Prototype
friend ostream& operator<<(ostream &, const Rectangle &);

// Implementation
ostream& operator<<(ostream &left, const Rectangle &right)
{
    left << "Length: " << right.length << "Width:  " << right.width;

    return left;
}
```

```
Rectangle r1, r1;
cout << r1;
// compiler sees as
operator<<(cout, r1);
```

# OVERLOADING STREAM EXTRACTION(>>)

```
// Generic Prototype
friend istream& operator>>(istream &, className &);

// Generic Implementation
istream& operator>>(istream &isObject, className &right)
{
    // Local declaration, if any
    // Read the data into right object
    // isObject >> …

    return isObject;
}
```

# EXAMPLE: OVERLOADED EXTRACTION(>>)

```
// Prototype
friend istream& operator>>(istream &, Rectangle &);

// Implementation
istream& operator>>(istream &left, Rectangle &right)
{
   left >> right.length >> right.width;

   return left;
}
```

```
Rectangle r1, r1;
cin >> r1;
// compiler sees as
operator>>(cin, r1);
```

# EXAMPLE

```
istream& operator>>(istream& in, Student& right)
{
   in >> right.gpa;
   in.ignore();
   getline(in, right.name);
   return in;
}
```

```
Student s1;
cin >> s1;
// compiler sees as
operator>>(cin, s1);
```

# OP OVERLOADING MEMBER AND NON-MEMBER IMPLEMENTATION?

| Overloaded operator | member function | non-member function |
|---|---|---|
| +, -, *, ==, >, etc... | YES | YES |
| <<, >> | NO | YES |
| = | ?? | ?? |