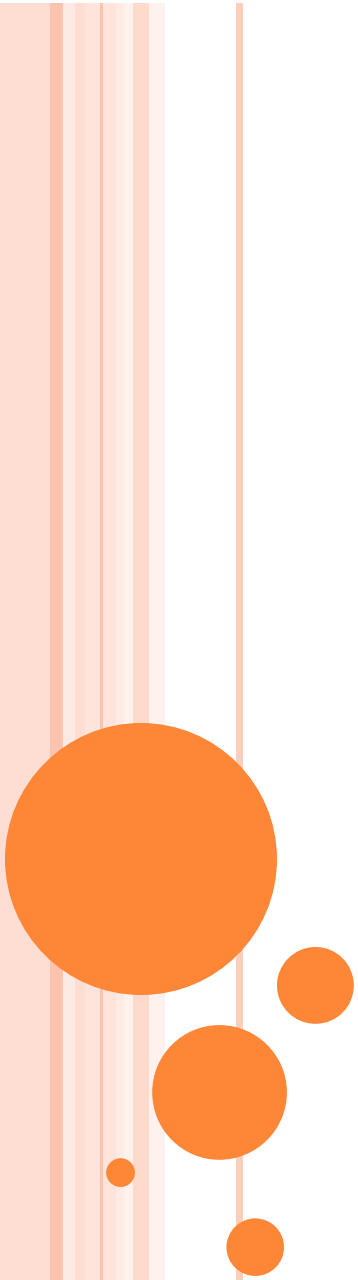

Operator Overloading

CS A250 – C++ Programming II

Function vs. Operator Overloading

- Do not confuse
  - **Function overloading**
    - **Functions** that have same name
  - **Operator overloading**
    - Customizing an **operator** to make it work with a class you implemented.

Overloading Functions

- ***Same*** function name
- ***Different*** parameter lists
- Two ***separate*** function definitions
- Function "**signature**"
  - Function **name** & **parameter list**
  - Must be "unique" for each function definition
- Allows same task performed on different data

Overloading Example: Average

- Function computes average of 2 numbers:

```
double average(double n1, double n2)
{
        return ((n1 + n2) / 2.0);
}
```

- Now compute average of 3 numbers:

```
double average(double n1, double n2, double n3)
{
        return ((n1 + n2 + n3) / 3.0);
}
```

- Same name, two functions

Overloading Example (cont.)

- Which function gets called?
- Depends on function call itself:
  - `avg = average(5.2, 6.7);`
    - Calls **two-parameter** **average()**
  - `avg = average(6.5, 8.5, 4.2);`
    - Calls **three-parameter** **average()**
- Compiler resolves invocation based on **signature** of function call
  - "Matches" call with appropriate function
  - Each considered separate function

Member vs. Non-Member Functions

- A **member** function
  - Belongs to a **class**
  - Can access **private member variables** ***directly***
  - **NO** need to use accessor functions to access its own private member variables.

- A **non-member** function
  - Does **NOT** belong to a **class**
  - **Cannot** access private member variables directly
  - **Must** use **accessor functions** to access **private member variables** of a class.

Friend Functions

- A **friend** function
  - Is a **non-member** function
    - **BUT** has access to ***all*** the members (**public** and **private**) of the class where the function is declared.

- The word ***friend***
  - Is added ***only*** in the **function declaration**
  - Is written ***inside*** the **class definition**, typically ***before*** the **public declaration**

- Important for **overloading operators**

Types of Functions (summary)

- **Member functions**
  - Member of a **class**
  - Can access ***private member variables*** **directly**
- **Friend function**
  - **Not** a member of a class
  - Can access ***private member variables*** **directly**
- **Non-member function**
  - **Cannot** access private member variables
  - Need to use ***accessor*** *functions*

REVIEW: CONST FUNCTIONS

- When to make function `const`?
  - Constant functions ***not*** allowed to alter class member variables
  - Constant objects can **ONLY** call constant member functions
- **Good style** dictates:
  - Any **member function** that will NOT modify data should be made `const`
- Use keyword `const` ***after*** function declaration and heading

Operator Overloading

- We use **operators** such as +, -, %, ==, etc.
  - These are just *functions*:

```
int n = x + 7;     // all the same
int n = add(x,7);
int n = +(x,7);
```

  - where "**+**" is the ***binary operator*** with **x** and **7** as ***operands***
    OR
  - "**+**" is the ***function name***
  - **x** and **7** are the ***arguments***
  - Function "**+**" ***returns*** "sum" of its arguments

Operator Overloading (cont.)

- We can **overload operators** to work with **OUR** types
  - The class **Pair** creates objects that contain two integers
  - We create two objects of the class **Pair**

```
Pair p1(1,20), p2(3,40);
```

  - We would like to check if they are **equal**

```
if (p1.getFirst() == p2.getFirst() &&
      p1.getSecond() == p2.getSecond())
{
    // do something…
}
```

Operator Overloading (cont.)

- If we **overload** the **comparison operator** **==** then our statement can be simplified to:

```
if ( p1 == p2 )
{
    // do something…
}
```

- Obviously, without overloading the comparison operator, the statement above will produce an error.

Overloading Operator ==

```
if ( p1 == p2 )
```

- We could produce the same results with a function **equal**:

```
if ( p1.equal(p2))
```

Overloading Operator == (cont.)

```
if ( p1.equal(p2))
```

- You implement the function as follows:

```
bool Pair::equal(const Pair& otherPair) const
{
     return ( first == otherPair.first &&
                  second == otherPair.second );
}
```

Overloading Operator == (cont.)

- To **overload** the **operator** **==**, simply change the **name** of the function:

```
bool Pair::operator ==(const Pair& otherPair) const
{
      return ( first == otherPair.first &&
                  second == otherPair.second );
}
```

Which Operators Can Be Overloaded?

- **Built-in** operators
  - `+`, `-`, `=` , `%`, `==`, `/`, `*` (and more)
  - Already work for C++ built-in types

- Both **binary** and **unary** operators can be overloaded
  - Binary has 2 operands → 3 + 4
  - Unary has 1 operand → ++3

Overloading the Subscript Operator [ ]

- The **subscript operator [ ]** can also be overloaded
  - **Must** be a **member** function

```
List myList;
myList.addElement(10);
myList.addElement(20);
myList.addElement(30);

cout << myList [1];  // would not work
     if //operator [] is not overloaded
```

Note that the class **List** is an **abstract class**—it could be implemented as a linked list, as a vector, as an array, etc.

OVERLOADING OPERATOR +

- The next slides show an example of overloading the **operator +** of the **Pair** class as a **member**, as a **friend**, and as a **non-member** function.

- **NOTE** that the **operator +** for the **Pair** class should be overloaded as a **member function** and **not** as a friend or a non-member function

  - **BUT** the idea is to show the ***differences*** in implementation.

Overloading Operator + (cont.)

- We will consider three different functions:
  - **`memberPlus`**
    - A **member** function of the class **Pair**
  - **`friendPlus`**
    - A **friend** function of the class **Pair**
  - **`nonMemberPlus`**
    - A **non-member** function
    - Would be implemented outside the class **Pair**

OVERLOADING OPERATOR + (CONT.)

- Create the objects and call each functions

```
Pair p1(10,11), p2(20,22);
Pair p3, p4, p5;

p3 = p1.memberPlus(p2);
p4 = friendPlus(p1,p2);
p5 = nonMemberPlus(p1,p2);
```

All three functions produce the **same** result.

Overloading Operator + (cont.)

Declarations

Member

```
Pair operator+( const Pair& p2 ) const;
```

Friend

```
friend Pair operator+( const Pair& p1,
                       const Pair& p2 );
```

Non-member

```
Pair operator+( const Pair& p1,
                const Pair& p2 );
```

OVERLOADING OPERATOR + (CONT.)

Definition as a member function

*A **member** function has a **calling object** and passes **only** the second pair as a parameter.*

```
Pair Pair::operator+(const Pair& otherPair) const
{
        //construct the object
    Pair tempPair (first + otherPair.first,
                   second + otherPair.second);

        // return the object
    return tempPair;
}
```

*A **member** function has **direct** access to member variables.*

OVERLOADING OPERATOR + (CONT.)

Definition as a friend function

*A **friend function** is **NOT** a member of the class; therefore, it does **not** have a calling object and needs to pass both pairs.*

```
Pair operator+(const Pair& leftPair,
                    const Pair& rightPair)
{
        //construct the object
    Pair tempPair (leftPair.first + rightPair.first,
                leftPair.second + rightPair.second);

        // return the object
    return tempPair;
 }
```

*A **friend** function has **direct** access to member variables.*

OVERLOADING OPERATOR + (CONT.)

Definition as a non-member function

*A **non-member function** does **not** have a calling object, because it is **not** a member of the class, and needs to pass both pairs.*

```
Pair operator+(const Pair& leftPair,
                    const Pair& rightPair)
{
       //construct the object
    Pair tempPair (leftPair.getFirst() +
                              rightPair.getFirst(),
              leftPair.getSecond() +
                              rightPair.getSecond());
       // return the object
    return tempPair;
 }
```

*A **non-member** function needs to use the **accessor functions** to get the values of objects.*

Overloading Operator + (cont.)

Function call

It is the **same** for member, friend, and non-member function:

```
Pair p1(1,2);
Pair p2(3,4);
Pair p3 = p1 + p2;
```

Member

```
Pair p3 = p1 + p2;

    p1 is the calling object
    p2 is the parameter
    p3 is the return value
```

Overloading Operator + (cont.)

Function call

It is the **same** for member, friend, and non-member function:

```
Pair p1(1,2);
Pair p2(3,4);
Pair p3 = p1 + p2;
```

Friend

```
Pair p3 = p1 + p2;

    p1 is the parameter
    p2 is the parameter
    p3 is the return value
```

Overloading Operator + (cont.)

Function call

It is the **same** for member, friend, and non-member function:

```
Pair p1(1,2);
Pair p2(3,4);
Pair p3 = p1 + p2;
```

Non-member

```
Pair p3 = p1 + p2;

    p1 is the parameter
    p2 is the parameter
    p3 is the return value
```

OVERLOADING OPERATOR + (CONT.)

- Keep in mind:
  - The **operator +** should be overloaded as a **member function.**
  - The previous examples were created to show you the differences in implementation.

OVERLOADING << AND >>

- Enables **input** and **output** of objects
  - **<<** **insertion** operator
  - **>>** **extraction** operator
- Improves ***readability***
  - Like all operator overloads do

```
cout << myObject;
cin >> myObject;
```

  - Instead of:

```
myObject.print();
```

OVERLOADING << AND >> (CONT.)

- The function that overloads the insertion or the extraction operator **must** be a **non-member** function

```
Pair p1(1,20);
cout << p1;
```

- **Why?** Because the **leftmost** operand (that is, **cout**) is an object of the type **ostream** and **not** of the type **Pair**

- To have ***direct*** **access** to **private** member variables we use a **friend** function

OVERLOADING << AND >> (CONT.)

- Place the declaration **inside** the class definition and ***before*** the `public:` section

```
friend std::ostream& operator<<(std::ostream& out,
                                const Pair& p);
```

- Place the definition in the **cpp** file **without** using the **class name** and **scope resolution**

```
ostream& operator<<(ostream& out, const Pair& p)
{
  out << "(" << p.first << "," << p.second << ")";
  return out;
}
```

A Few Rules and Exceptions

- When overloading an operator:
  - At least **one parameter** must be of **class type**
  - You **cannot** create a new operator
  - You **cannot** change the number of arguments an operator has

- **Cannot** overload
  - The dot operator (`.`)
  - Scope resolution (`::`)
  - Conditional operator (`?` `:`)

A Few Rules and Exceptions (cont.)

- Do ***not*** overload `&&` and `||`
- Only **member** functions can overload these operators

```
( )  [ ]  ->  =
```

A Few Rules and Exceptions (cont.)

- If the ***leftmost*** **operand** is
  - An object of a **different type** from the class
    - The function that overloads the operator must be a **non-member**
    - Create a **friend** function instead
    - As seen with **cout**
  - An object of the **same type** of the class
    - The function that overloads the operator **should** be a **member**

A Few Rules and Exceptions (cont.)

- Overloading **increment** and **decrement operators**
  - Need 2 versions
    - **Prefix** notation: `++n`
    - **Postfix** notation: `n++`

- Overloading the **assignment operator =**
  - Automatically overloaded
  - **BUT**, need to write your own if using **pointers**

(These operators will be addressed next time)

Operator Overloading (end)