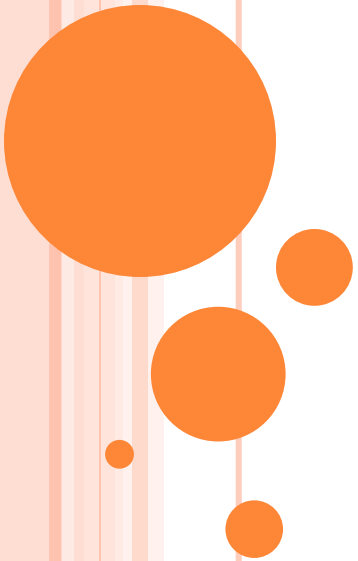


More on Overloading

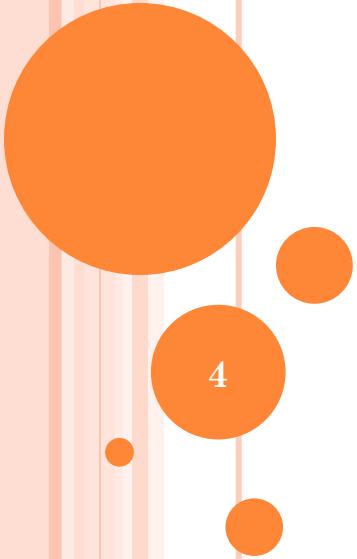
CS A250 – C++ Programming II

Operator Overloading

- We have seen how to overload operators as
  - **member** functions
  - **non-member** functions
  - **friend** functions
- **Operator overloading** (known also as **syntactic sugar**) improves code **readability**

Operator Overloading

- We will look at **overloading operators** that require a somewhat more complex implementation:
  - **Increment** and **decrement** operators
  - **Assignment** operator
    - Including the **DArray class**



Overloading Prefix and Postfix

Overloading ++ and --

- The **prefix** and **postfix** versions of the **increment** and **decrement** operators can all be overloaded.
  - Each overloaded operator has a distinct **signature**, so that the compiler is able to determine whether it is a **prefix** of a **postfix**.

The Prefix Operator

- With the class **Pair** in mind, we want to be able to use a function call such as:

```
Pair p(4,7);
++p;  // prefix
```

that will change **p** to (5, 8).

The Prefix Operator (cont.)

- Overloading the **prefix increment** (or **decrement**) **operator** as a **member function**:

```
Pair& operator++(); //declaration

Pair& Pair::operator++() //definition
{
        ++first;
        ++second;
        return *this; //return reference to
                      //incremented object
 }
```

The Postfix Operator

- With the class **Pair** in mind, we want to be able to use a function call such as:

```
Pair p(5,7);
p++;  // postfix
```

that will change **p** to (6, 8).

The Postfix Operator (cont.)

- Overloading the **postfix increment** (or **decrement**) **operator** as a **member function**:
  - Need to let the compiler know it is a ***postfix***
  - We add a "dummy parameter"

The Postfix Increment Operator (cont.)

```
Pair operator++( int ); //declaration

Pair Pair::operator++( int ) //definition
{
   Pair temp = *this; //hold current state of object
   ++first;
   ++second;
   return temp;  //return temporary object
}
```

Note on Efficiency

- The additional object that is created by the **postfix increment** (or **decrement**) operator can result in a **significant performance problem** (*especially* in a loop)

- You should use the **postfix increment** (or **decrement**) operator only when the logic of the program requires post-incrementing (or post-decrementing).

Example

- Project: Pair class

The DArray Class

A Pointer as a Member Variable

- Having a **pointer** as a **member variable** means that the pointer will be pointing at a **dynamic variable** that is stored in the **heap**
  - The dynamic variable is **not** *physically* part of the object
  - BUT the pointer is part of the object

- We have already seen this when implementing linked lists.

A Pointer as a Member Variable (cont.)

- The class **DArray** creates objects that contain three variables:
  - An **int** to store the **capacity** of the array
  - An **int** to store the **number of elements** in the array
  - A **pointer** that will **point** to an **array of integers**

Class DArray

```
const int CAP = 100;

class DArray
{
public:
    DArray( );
    // other functions
    ~DArray( );

private:
    int *a;      //will point to an array of integers
    int capacity;
    int noOfElem;
};
```

```
// default constructor

DArray::DArray()
{
    capacity = CAP;
    noOfElem = 0;

    a = new int[capacity];
}
```

Class DArray (cont.)

Object of the class DArray

```
int * a = [array address]
int capacity = 10
int noOfElem = 6
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
35	67	91	42	73	15				

[array]

Class Darray (cont.)

Object of the class DArray

```
int * a = [array address]
int capacity = 10
int noOfElem = 6
```

What does that mean?

It means that you need to think carefully when adding a **const** modifier to a **member function** of the class **Darray.**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
35	67	91	42	73	15				

[array]

The `const` Modifier on Functions

- **Recall:**
  - You add a `const` to a **member function** when the member variables of the class will **not** be modified.

The const Modifier on Functions (cont.)

Object of the class DArray

```
int * a = [array address]
int capacity = 10
int noOfElem = 6
```

Should a member function that replaces one element in the array be **const**?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
35	67	91	42 81	73	15				

[array]

The const Modifier on Functions (cont.)

Object of the class DArray

```
int * a = [array address]
int capacity = 10
int noOfElem = 6
```

Should a member function that replaces one element in the array be **const**?

Yes! The member variables of the object will not be modified.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
35	67	91	42 81	73	15				

[array]

42 was simply replaced by 81.

The const Modifier on Functions (cont.)

Object of the class DArray

```
int * a = [array address]
int capacity = 10
int noOfElem = 6
```

Should a member function that deletes an element in the array be **const**?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
35	67	91	81	73	15				

[array]

The const Modifier on Functions (cont.)

Object of the class DArray

```
int * a = [array address]
int capacity = 10
int noOfElem = 6  5
```

Should a member function that deletes an element in the array be **const**?

No. The number of elements will be decremented, modifying the member variable of the class.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
35	67	91	81 73	73 15	15				

[array]

81 was deleted by shifting all elements to the right of 81.
The number of elements in the array is now 5.

The const Modifier on Functions (cont.)

Object of the class DArray

```
int * a = [address of array]
int capacity = 10
int noOfElem = 6
```

Do **not** forget that **a** is a pointer!

```
bool Darray::compareElemAtIdx (const Darray& otherArray,
                                                int idx) const
{
      return (a[idx] == otherArray.a[idx];
}
```

This pointer is referring to the **calling object**.

This pointer is referring to the object **otherArray**.

The DArray Class

- **Project:** DArray class

Overloading the Assignment Operator

Assignment Operator

- A **default** **overloaded assignment** operator is available, **BUT**
  - If **dynamic variables** are used, you should overload the assignment operator.

- **Overloading the assignment operator** also allows for some specialized uses
  - Example:

```
object1 = object2 = object3;
```

- Function must be a **member of the class**
  - **Cannot** be a non-member and **cannot** be a friend.

Preventing Self Assignment

- It is important to ***prevent*** **self assignment**
  - This is to avoid that the **operator=** deletes the dynamic memory associated with the object before the assignment is completed.
  - This would lead to "*fatal runtime errors*".

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
             if (capacity != rightSide.capacity)
             {
                 delete [ ] a;  //release space
                 a = new int[rightSide.capacity]; //re-create array
                 capacity = rightSide.capacity;
             }
             for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                  a[i] = rightSide.a[i];
             noOfElem = rightSide.noOfElem ;
     }
     else
             cerr << "Attempted assignment to itself.";

     return *this;
}
```

Let's look at this in detail…

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
    if                                  id self assignment
    {
                                        Side.capacity)
                {
                    delete [ ] a;  //release space
                    a = new int[rightSide.capacity]; //re-create array
                    capacity = rightSide.capacity;
                }
                for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                     a[i] = rightSide.a[i];
                noOfElem = rightSide.noOfElem;
    }
    else
                cerr << "Attempted assignment to itself.";

    return *this;}
```

We return a **reference** to the object.

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
             if (capacity != r
             {
                  delete [ ] a;
                  a = new int[r                    create array
                  capacity = ri
             }
             for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                  a[i] = rightSide.a[i];
             noOfElem = rightSide.noOfElem;
     }
     else
             cerr << "Attempted assignment to itself.";

     return *this;
}
```

Make sure the calling object and the parameter are **not** the same array.

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
              if (capacity != rightSide.capacity)
              {
                   delete [ ] a;  //release space
                   a = new int[rightSide.capacity]; //re-create array
                   capacity = rightSide.capacity;
              }
              for (int i
                    a[i]
              noOfElem =
     }
     else
              cerr << "A

     return *this;
}
```

The **array parameter** needs to have the **same capacity** of the **calling object**.

If not, clear the memory that holds the array parameter and re-create a new array with the same capacity of the calling object.

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
             if (capacity != rightSide.capacity)
             {
                 delete [ ] a;  //release space
                 a = new int[righ                    array
                 capacity = right
             }
             for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                  a[i] = rightSide.a[i];
             noOfElem= rightSide.noOfElem;
     }
     else
             cerr << "Attempted assignment to itself.";

     return *this;
}
```

Start copying elements…

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
             if (capacity != rightSide.capacity)
             {
                 delete [ ] a;  //release space
                 a = new int[rightSide.capacity]; //re-create array
                 capacity = rightSide.capacity;
             }
             for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                  a[i] = rightSide.a[i];
             noOfElem = rightSide.noOfElem;
     }
     else
             cerr << "At

     return *this;
}
```

Update the number of elements for the array parameter.

Overloading the Assignment Operator

```
DArray& DArray::operator=(const DArray& rightSide)
{
     if (&rightSide != this) //avoid self assignment
     {
             if (capacity != rightSide.capacity)
             {
                 delete [ ] a;  //release space
                 a = new int[rightSide.capacity]; //re-create array
                 capacity = rightSide.capacity;
             }
             for (int i = 0; i < rightSide.noOfElem; ++i) //copy
                  a[i] = rightSide.a[i];
             noOfElem = rightSide.noOfElem;
     }
     else
             cerr << "Attempted assignment to itself.";

     return *this;
}
```

CAUTION!

- The **overloaded assignment operator** works **only** if the object is declared as a **separate statement**:

```
Darray a1;
// insert elements in a1
Darray a2;
a2 = a1;
```

- This is because the object **a2** needs to be constructed first.

More on Overloading (end)