# More on C++ Classes

## Destructors

A destructor is a special method of a class that is executed automatically when an instance of the class is destroyed. Unlike constructors, there can be only be one destructor for each class. The name of the destructor is the tilde character (~) followed by the name of the class. The destructor has no return type and takes no parameters. One common task performed in the destructor is to release resources held by the object. This may be releasing memory that was dynamically allocated, closing open files, closing database connections, closing network connections, etc.

Destructors are normally called in reverse order as the constructors. A global object's constructor is called before any functions, including the main function. The destructor for a global object is called when main ends. The Constructor for an automatic object is called when execution reaches the point where the object is created. Its destructor is called when execution leaves the object's scope. The constructor for a static object is called only once, when execution reaches the point where the object is created. Its destructor is called when the main function ends.

Programming Examples: CreateAndDestroy1.cpp & CreateAndDestroy2.cpp

## Returning a Reference to Private Data

One common error is to return a reference to a private data member. Doing this gives direct access to the class's private data. Once a reference to private data has been handed out, the data can be changed, bypassing any data validation the class may have.

Revisiting the **`Time`** class example from earlier in the semester, the pitfalls of returning a reference to private data is illustrated.

In this program the **`setHours`** method returns a reference to the private data **`hours`** and the **`setMinutes`** method returns a pointer to the private data **`minutes`**. The program then uses these references and associated pointers to set the time to invalid values in several different ways.
Programming Examples: Time.h, Time.cpp, and TimeTester.cpp

## Copying Objects

One object can be copied to another object using the assignment operator =. When this is done a member wise assignment is performed. This means that the value of each data member is copied from the source object to the destination object.

Programming Examples: Date.h, Date.cpp, and DateTester.cpp

Note that while copying object like this is acceptable for simple data types, it can cause problems when used with data members that are themselves instances of classes.

## Operator Overloading

Just like the methods of a class can be overloaded, many of the operators in C++ can be overloaded to behave in a way you define for your class. It is a good idea to use operator overloading when it makes your class clearer and easier to use. When using operator overloading you should try to make the operator mimic the functionality of the built-in operators. Each operator that is used by your class is actually a function included in your class. These functions have special names. For instance, the function used for overloading the less than operator is **`{operator<}`**. For example, suppose we have a class called Rectangle and that we have overloaded the less than operator. Then comparing two Rectangle objects **`r1`** and **`r2`** can be done in either of the following two ways; Both are equivalent.

```
r1 < r2;
r1.operator<(r2);
```

The examples in this section will use a class named **`Rectangle`** that has two private **`double`** data members, **`height`** and **`width`**. The class include the usual accessors and mutators. No data validation is performed, which means the height and length can be negative.
Programming Examples: Rectangle.h, Rectangle.cpp, and RectangleTester.cpp

## Copy Constructors

It is often desirable to create a new object, a Rectangle in this case, like this

```
Rectangle r1(r2)
```

This will make a new Rectangle **`r1`** that is identical to Rectangle **`r2`**. To do this, we need to initialize the instance fields of **`r1`** using a *member initialization list*.

## `this` and `friend`

Every instance of a class (i.e. every object) maintains a pointer to itself. The name of the pointer is **`this`** and is used when overloading certain operators.
A function that is defined outside the scope of a class is called a *friend* function. Although a friend function is a non-member function, it still has access to the private data members of the class. To add a friend function to a class you include the keyword **`friend`** before the function prototype in the class header.

## Stream Insertion (<<) and Stream Extraction (>>) Operators

The function for overloading the stream insertion and extraction operators cannot be a member function of the class. To see why, consider the following expression for displaying a rectangle object.

```
cout << r1;
```

As we saw, this is equivalent to

```
cout.operator<<(r1);
```

which is clearly not a member of the **`Rectangle`** class. This means, that in the **`Rectangle`** class the insertion and extraction operators must be defined as friend functions. These two overload functions take two parameters, a reference to the input/output stream and a constant reference to a **`Rectangle`** object. They return a reference to the input/output stream to allow for this

```
cout << r1 << r2;
```

For the **`Rectangle`** class we just display the height and width of the rectangle. The overloaded input stream accepts a height and width from the input stream and returns a reference to the input stream to allow for

```
cin >> r1 >> r2
```

## Binary Relational operators

The Rectangle class can compare two rectangles by considering their areas. A rectangle with the smaller area is considered to be less than a rectangle with the larger area. The less than operator is overloaded like this:

```
bool Rectangle::operator<(const Rectangle &right) const {
     return (*this).area() < right.area();
}
```

The **const** in the parameter means that this function cannot modify the parameter **right**, which is a reference to another **Rectangle** object. The **const** following the method signature indicates that the method cannot modify any of the instance fields of the **this** object. The remaining relational operators are defined in terms of the *less than* operator using the technique.

### Assignment Operator =

If an object is assigned to itself we need do nothing. Therefore, the first thing we do is to check of the right-hand side of the operator is equal to this object. If the two are different, then we copy the instances fields from the right-hand operand to this. A reference to a **Rectangle** is returned to allow us to do this

```
r1 = r2 = r3;
```

### Compound Assignment Operators +=, -=, *=, /=

The important thing to remember here is that when we use one of these operators like this

```
r1 += r2;
```

it is equivalent to

```
r1.operator+=(r2);
```

which changes **r1**. Hence, we need to get the values of the instance fields from the right-hand side (passed as a parameter) and use that information to modify the instance fields of this object. For the **Rectangle** class we do this using the accessor and mutator functions. Notice that since the right-hand side is not expected to be updated, we pass it as a **const** reference.

### Binary Arithmetic Operators +, -, *, /

The binary arithmetic operators don't modify either the right or the left operator. Rather, they create a *new* **Rectangle** object, set with the correct values, and return it. To avoid duplicating work, we simply create a new **Rectangle** that is a duplicate of this rectangle, then use the corresponding compound assignment operator to combine it with the right-hand side **Rectangle**, and finally return the new **Rectangle** object.

### Unary Pre/Post Increment/Decrement Operators

These are the operators that have side effects. That means that we do two things: (i) we need to modify the **Rectangle** and (ii) we then create and return a new **Rectangle**. First, we will look at the pre-increment (**++r**) and pre-decrement (**--r**) operators. Since these are unary operators they take no arguments. Here we mimic the behavior of the built-in data types. We first update the values of **Rectangle r**, make a copy of **r**, and finally return the copy.

To overload the post-increment (**`r++`**) and post-decrement (**`r--`**) operators, we need a way to distinguish the overload function. To do this, the post-increment and post-decrement each have a dummy **`int`** parameter. In these functions we create the new copy of **`r`** first, then update **`r`**, and finally return the copy.

## Unary Plus and Minus Operators

These operators take parameters and return a new **`Rectangle`** object. For the unary plus operator, the new object is an exact copy of the Rectangle. For the unary minus operator, the new object is the "minus" of the Rectangle.