

# The Class Construct – Part 2

Lecture 23  
Sections 7.7 - 7.9

Robb T. Koether

Hampden-Sydney College

Fri, Oct 25, 2019

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions
  - Inspectors
  - Mutators
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions
  - Inspectors
  - Mutators
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Class Scope

- A member function is given **class scope** by writing the class name followed by `::` before the function name in the function definition.
- For example,

```
double Point::getX() const { ... }
```

- If a function has class scope, then the private members may be accessed freely.
- If a function does not have class scope, then the data members may not be accessed.

# Invoking Member Functions

## Invoking Member Functions

```
int main()
{
    Point p(1, 2);
    double x = p.getX();
    :
}
```

- Outside the scope of the class, (public) member functions may be invoked only through an object of that class.
- The form is *object.function(params)*.
- The dot (.) is the **member access operator**.

# The `Point2` Class

- Example

- `Point2.h`
- `Point2.cpp`
- `Point2Test.cpp`

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions
  - Inspectors
  - Mutators
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Header Files

- Typically, the class definition is placed in a **header file**.
  - Name the file `class-name.h`.
  - Example: `Point.h`.
  - Write only the class construct in the header file.
  - Include any necessary “include” files.
  - Do not add the header file to the project.
  - The header file will be included by other files, as necessary.

```
#include "Point.h"
```



# Implementation Files

- Typically, the member functions are defined in the **implementation file**.
  - Name the file `class-name.cpp`.
  - Example: `Point.cpp`.
  - Write the definitions of all the member functions.
  - You must add the implementation file to the project.

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions**
  - Inspectors
  - Mutators
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions**
  - **Inspectors**
  - Mutators
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Inspectors

- An **inspector** returns the value of a data member (or other attribute).
- An inspector's name may begin with the word “get,” followed by the name of the attribute (a “getter”).
- An inspector is normally declared to be constant.
- An inspector's return type is the type of the data member or attribute being returned.

# Point Class Example

## The `Point` Class

```
class Point
{
    public:

        // Inspectors

        double getX() const;
        double getY() const;
        :
};
```

# Accessing Data Members

## The `Point` Class

```
double Point::getX() const
{
    return x;
}
```

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions**
  - Inspectors
  - **Mutators**
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Mutators

- A **mutator** modifies the value of a data member.
- A mutator's name may begin with the word “`set`,” followed by the name of the data member (a “setter”).
- Normally the return type of a mutator is **`void`**.
- A mutator should verify (as necessary) that the value to be assigned is valid.
- Often mutators are used by the constructors to initialize the data members (when values must be verified).



# Point Class Example

## The `Point` Class

```
class Point
{
    public:

    // Mutators

    void setX(double xval);
    void setY(double yval);
    :
};
```

# Mutators and Constructors

## Mutators and Constructors

```
int main()
{
    Point p(1, 2);    // p = (1, 2)
    p.setY(3);        // p = (1, 3)
    :
}
```

# The `Point3` Class

- Example

- `Point3.h`
- `Point3.cpp`
- `Point3Test.cpp`

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions**
  - Inspectors
  - Mutators
  - Facilitators**
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Facilitators

- A **facilitator** is designed to be invoked by an operator, although it may be invoked directly.
- A facilitator's name is usually the name of the operator that it facilitates.
- Most operators are **binary**, i.e., require two operands, a **left operand** and a **right operand**.
- The invoking object is one of the operands.
- The other operand is given as the parameter.

# Point Class Example

## The Point Class

```
class Point
{
    public:

        // Facilitators

        void output(ostream& out) const;    // Operator <<
        bool isEqual(const Point& p) const; // Operator ==
        :
};
```

# The Point4 Class

- Example

- `Point4.h`
- `Point4.cpp`
- `Point4Test.cpp`

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions**
  - Inspectors
  - Mutators
  - Facilitators
  - **Operators**
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment



# Operators

- An **operator** performs a function that is traditionally represented by a symbol, such as  $+$  and  $*$  for addition and multiplication.
- An operator is implemented as a function.
- A function's name begins with the keyword **operator**, followed by the symbol for the operator.
- For example, **operator+()**.

# Operators

- Typically, an operator is not a member function. (There are exceptions.)
- If an operator is not a member function, then it does not have access to the class's data members.
- That is the reason for the facilitators.
- An operator invokes a facilitator to gain access to the data members and “get the job done.”

# Binary Operators

- A binary operator is normally invoked by writing the operator between two objects of the appropriate types.
- For example,  $p + q$ .
- A binary operator may also be invoked by writing the function name with a parameter list.
- For example, **operator**+( $p$ ,  $q$ ).
- In fact, writing  $p + q$  is interpreted by the compiler as **operator**+( $p$ ,  $q$ ).

# Point Class Example

## The Point Class

```
class Point
{
    // Facilitators

    void output(ostream& out) const;
    bool isEqual(const Point& p) const;
    :
};

// Operators

ostream& operator<<(ostream& out, const Point& p);
bool operator==(const Point& p, const Point& q);
```

# Point Class Example

## The Point Class

```
bool Point::isEqual(const Point& p) const
{
    return (x == p.x) && (y == p.y);
}

bool operator==(const Point& p, const Point& q)
{
    return p.isEqual(q);
}
```

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions
  - Inspectors
  - Mutators
  - Facilitators
  - Operators
- 4 The Point Class**
- 5 The Destructor
- 6 Assignment

# The Complete `Point` Class

- Example

- `Point.h`
- `Point.cpp`
- `PointTest.cpp`

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions
  - Inspectors
  - Mutators
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment



# The Destructor

## The Destructor

```
~class-name();
```

- The class's **destructor** is a member function that “destroys” the object automatically when it passes out of scope.
- It destroys the object by deallocating the memory that it occupied (but it does not erase the memory).
- The name of the destructor *must* be the name of the class, preceded by a tilde (~).

# The Destructor

- Add the `Point` class destructor to the `Point` class.
- Have it write the message `"Point (x, y) is destroyed"`  
(Fill in values for `x` and `y`.)
- Then run the test program `PointTest.cpp`.

# Outline

- 1 Class Scope
- 2 Header Files
- 3 The Member Functions
  - Inspectors
  - Mutators
  - Facilitators
  - Operators
- 4 The `Point` Class
- 5 The Destructor
- 6 Assignment

# Assignment

## Assignment

- Read Sections 7.7 - 7.9.