# Lecture 6.2

Topics

1. Class Constructor Overloading

---

## 1. Class Constructor Overloading

How many constructors are allowed for a class? The answer is there can be as many constructors as one would "like" to have. The case of constructor without parameter (default constructor) has been examined. Let us consider it again and other constructors, where values are passed to constructors to create objects.

### 1.1 Constructors with No Arguments (default constructor) – Revisited

In many classes, private date members will have the companion `getter/setter` functions. These functions will allow access to most private data members. Let's look at the following example with another version of **`Fraction`** class.

*Example 1*

```
/**
 *Program Name:  cis25L0621.cpp
 *Discussion:    Class with CONSTRUCTOR for
 *                       Data Initialization
 */

#include <iostream>
using namespace std;

class Fraction {
public:
  Fraction() {
    iNum = 0;
    iDenom = 1;
  }

  int getNum() {
    return iNum;
  }

  void setNum(int iOld) {
    iNum = iOld;

    return;
  }

  int getDenom() {
    return iDenom;
  }

  void setDenom(int iOld) {
    if (iOld) {
      iDenom = iOld;
    } else {
      iDenom = 1;
    }
```

```
    return;
  }

private:
  int iNum;
  int iDenom;
};

int main() {
  Fraction frA; // What can we do with this
                // prfrA object?

  // Print information
  cout << "Numerator : " << frA.getNum() << endl;
  cout << "Denominator : " << frA.getDenom() << endl;

  return 0;
}
```

**OUTPUT**

```
Numerator : 0
Denominator : 1
```

In this example, a constructor with no argument was defined; it is called a **default** constructor. When objects being created as shown above, there are implied calls to the default constructors to initialize the objects.

This `Fraction` object will have the initial values of 0 for the numerator and 1 for the denominator. Of course, one can modify this `Fraction` through the calls to `setNum()` and `setDenom()`.

### 1.2 Copy Constructor

An argument for a constructor must not belong to the same class that the constructor is defined for. That means a class **CC** constructor must not take an argument of type **CC**. However, a reference argument is allowed. If a constructor of class **CC** takes an argument of type **CC**&, that is, **CC** reference, then this constructor is called a **copy constructor**.

Consider the following example,

Example 2

```
/**
 *Program Name:  cis25L0622.cpp
 *Discussion:    Class with CONSTRUCTORS for
 *                       Data Initialization
 */

#include <iostream>
using namespace std;

class Fraction {
public:
  Fraction() {
    iNum = 0;
    iDenom = 1;
```

```
  }

  Fraction(const Fraction& frOld) {
    iNum = frOld.iNum;
    iDenom = frOld.iDenom;
  }

  int getNum() {
    return iNum;
  }

  void setNum(int iOld) {
    iNum = iOld;

    return;
  }

  int getDenom() {
    return iDenom;
  }

  void setDenom(int iOld) {
    if (iOld) {
      iDenom = iOld;
    } else {
      iDenom = 1;
    }

    return;
  }

private:
  int iNum;
  int iDenom;
};

int main() {
  Fraction frA; // What can we do with this
                // prfrA object?

  // Print information

  cout << "Numerator : " << frA.getNum() << endl;
  cout << "Denominator : " << frA.getDenom() << endl;

  frA.setNum(2);
  frA.setDenom(3);

  Fraction frB(frA);

   // Print information

  cout << "Numerator : " << frB.getNum() << endl;
  cout << "Denominator : " << frB.getDenom() << endl;

  return 0;
}
```

**OUTPUT**

```
Numerator : 0
Denominator : 1
Numerator : 2
Denominator : 3
```

Note that the argument of this constructor is the reference of some existing `Fraction` object (e.g., `frOld` in the above example); the direct access of the private member (e.g., `frOld.iNum`) is legal for this case.

If the code for the copy constructor is not explicitly provided then the compiler will automatically provide it. The compiler version would have the same effect as,

```
Fraction::Fraction( const Fraction& frOld );
```

### 1.3 Constructors with One Parameter (Argument) – Convert Constructor

A constructor may receive values for initialization; but there should still be no return type for a constructor no matter how many arguments it may have.

A one-parameter (other than **`CC`** &) constructor is called a **convert constructor**. A convert constructor converts type **`CV`** to **`CC`**. Consider the following example.

*Example 3*

```
/**
 *Program Name:  cis25L0623.cpp
 *Discussion:      Class with CONSTRUCTORS for
 *                       Data Initialization
 */

#include <iostream>
using namespace std;

class Fraction {
public:
  Fraction() {
    iNum = 0;
    iDenom = 1;
  }

  Fraction(const Fraction& frOld) {
    iNum = frOld.iNum;
    iDenom = frOld.iDenom;
  }

  Fraction(int iOld) {
    iNum = iOld;
    if (iOld) {
      iDenom = iOld;
    } else {
      iDenom = 1;
    }
  }


  int getNum() {
    return iNum;
```

```
  }

  void setNum(int iOld) {
    iNum = iOld;

    return;
  }

  int getDenom() {
    return iDenom;
  }

  void setDenom(int iOld) {
    if (iOld) {
      iDenom = iOld;
    } else {
      iDenom = 1;
    }

    return;
  }

private:
  int iNum;
  int iDenom;
};

int main() {
  Fraction frA; // What can we do with this
                // prfrA object?

  // Print information

  cout << "Numerator : " << frA.getNum() << endl;
  cout << "Denominator : " << frA.getDenom() << endl;

  frA.setNum(2);
  frA.setDenom(3);

  Fraction frB(frA);

   // Print information

  cout << "Numerator : " << frB.getNum() << endl;
  cout << "Denominator : " << frB.getDenom() << endl;

  Fraction frC(6);

   // Print information

  cout << "Numerator : " << frC.getNum() << endl;
  cout << "Denominator : " << frC.getDenom() << endl;

  return 0;
}
```

## OUTPUT

```
Numerator : 0
```

```
Denominator : 1
Numerator : 2
Denominator : 3
Numerator : 6
Denominator : 6
```

In this example, a constructor with one argument of type **`int`** is defined. This constructor is converting one integer given as argument to a `Fraction` object.

There may be several convert constructors that will allow the object to be created through these conversions.