```
    Circle (double r)
    {   radius = r;
        Count++;
    }
    double getArea()
    { return 3.14*radius*radius; }
    Static int getCount()
    { return count; }
}; //end Circle
int Circle:: count=0; // initialization is outside of the class
// Driver for class Circle
int main()
{   Circle C1(2.4);
    Circle C2(3.8);
    cout << " Total objects: " << Circle:: getCount() <<endl; // 2
    return 0;
} //end main
```

Week 4, Sat: Operator Overloading

When we have two variables we don't have any issue with arithmetic operations.

e.g.
```
int a=15;
int b=10;
int p=0;
```
→
```
p= a*b;
p= a/b;
```

Whenever we have a Compound variable, we need overload operators:

+, −, *, and /.

Example: $C1 = 2-7i$

$C2 = -5+3i$

$C = C1+C2;$ → $C = (2-5)+(-7+3)i$

$C = -3-4i$

- Multiplication of fraction:

$$\frac{a}{b} \times \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$$

$f1$ ← $\frac{a}{b}$ × $\frac{c}{d}$ → $f2$ = $\frac{a \cdot c}{b \cdot d}$ → f

We Cannot write:

$f = f1 * f2$

// Multiplication of Fractions

```
Class Fraction
{  private:
          double num, denom;

   public: // default Constructor
          Fraction ( )
          { num=0.0, denom=1.0; }
          // Constructor with two arguments
          Fraction (double n, double d)
          {  num=n;
             denom = d;
          }
          // We also write getter and setter functions
          friend Fraction operator * (Fraction f1, Fraction f2);//
```

func. prototype

friend → //means we can use this func. without using an object

Fraction → return type

operator * → func. name

```
};// end Fraction
Fraction operator * ( Fraction f1, Fraction f2)
{   Fraction temp; // using default Constructor 0.0/1.0
    temp.num = f1.num * f2.num ;
    temp.denom = f1.denom * f2.denom;
    return temp;
}
// Driver for Fraction
int main( )
{   Fraction fr1 (3.0, 5.0); // 3.0/5.0
    Fraction fr2 (2.0,10.0); // 2.0/10.0
    Fraction product; // using default Constructor
    product = fr1 * fr2;
    Cout <<"The product is: " << product.getNum( ) << "/" <<
          << product.getDenom( ) <<endl; // 6.0/50.0
    return 0;
}// end main
```

We can get fractions from the user

returns the result 6.0/50.0

6.0 / 50

// $5\frac{2}{3} = \frac{17}{3}$

We can overload the insertion operator (<<):

```
//prototype
friend ostream & operator << (ostream &O, const Fraction &f);
                  function name
ostream & operator << (ostream &O , const Fraction &f)
                        cout   optional name   optional name
{
   O << f.getNum() << "/" << f.getDenom();
   return O; // it is not zero
}
//Now in the driver of class:
   cout << product; // 6.0/50.0
```

"Oh"

"Oh"

function Call

Another example for Operator Overloading:

```
Class Distance
{ private:  int feet;
            int inches;
  public:   // default Constructor
            Distance()
            { feet=0; inches=0; }
            // Constructor with arguments
            Distance (int f, int i)
            { feet=f;
              inches=i;
            }
            void display()
            { cout<< "F: "<< feet << "I: "<<inches <<endl;}
            void operator = ( const Distance D)
            {
              feet = D.feet;
              inches = D.inches;
            }
};// end class
```

assignment operator

// D3 = D1

D1: argument

D3 = D1: function Call

D3: function name

```
// Driver for Distance class
int main()
{
    Distance D1(11, 10), D2(5, 11);   // feet, inches
    D1.display();
    D2.display();
    D1 = D2;   // We are passing this argument; function call
    D1.display(); // F: 5   I: 11
    Distance D3;
    D3 = D1; // We are using = operator overloaded
    D3.display();
} //end main
```

- Following Operators Cannot be Overloaded:

  `.` dot

  `::`

  `?:` → ternary operator (short-cut for if-else)

  `sizeof` → returns # of bytes

- Overloading operator extraction (>>):

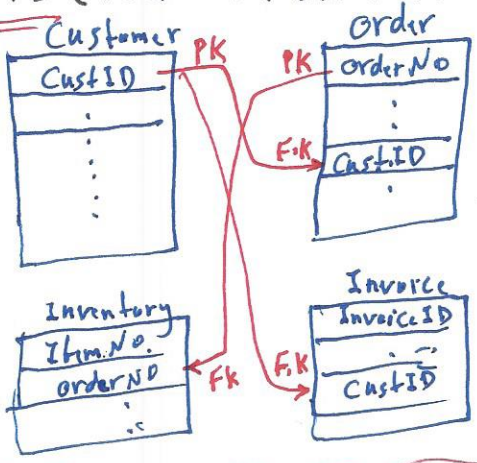
```
//prototype
friend istream & operator >> ( istream & i , Fraction &f);
                               // cin

istream & operator >> (istream &i, Fraction &f)
{   char slash = '/';
    i >> f.num >> slash >> f.denum;   // cin   // We can read: 5/7
    return i;
}
int main()
{
    ...
    cout << "Enter the first fraction: ";
    cin >> fr1;   // 3/5
}
```

`cin >>` → func. name; `fr1` → func. call

Data Structure:

RDBMS (Relational Database Management Systems): CSM10DB

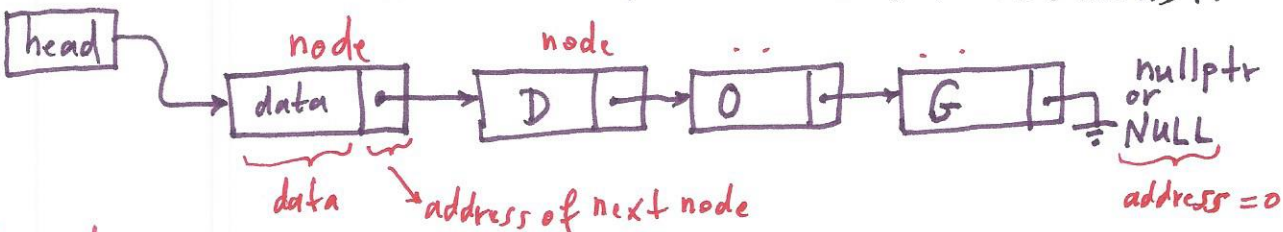


* We write SQL (Structure Query Language)
* MS SQL Server 2022 is free to download and install in your Computer.

A **data structure** is a way to store and organize data in memory of Computer, in order to access and use it efficiently.
We have used arrays so far. Array has a fixed size.

A **Linked List** is a dynamic data structure that can grow and shrink during the program execution, so we can use as much as memory we need.

- Advantages: 1- Use as much as memory you need
  2- Easy to add / delete data
- Disadvantages: 1- Memory overhead, store extra information
  2- Slow access of data. We have traverse the list.



Examples:
- Set of open windows on your Computer's Desk Top.
- Set of files in a folder
- The text being typed in a word document (list of characters) "Welcome"
- Our Brain or Train

A **Node** in a linked list is a block of memory that has data and a pointer to another node (rest of the list).

Node: 

17
Sp. 24
Sat.

* Test #1, Sat. 3-9-24, OOP, inheritance, pointers, virtual funcs., static funcs, operatorOverloading.
  - Multiple choice and T/F Questions, ScanTron 882-E
    You can find review questions under week 4 in Canvas.