Objective:

- Introducing the 'class' keyword and a bit of logic as always to keep your brains working ☺

Challenge – 1: *Die Class* (4)

Dice traditionally have six sides, representing the values 1 to 6. Some games, however, use specialized dice that have a different number of sides. For example, the fantasy role playing-game Dungeons and Dragons uses dice with 4, 6, 8, 10, 12, and 20 sides.

Suppose you are writing a program that needs a roll simulated dice with various numbers of sides. A simple approach would be to write a class named Die which stores the number of sides in it. The class would also have appropriate methods to roll the die and getting the die value.

Considering the above discussion, a class Die is designed below for which you need to provide the implementation of its methods.

```
class Die
{
    int sidesCount;                          //count of sides of the die
    int value;                               //The die's value
public:
    void initializeDie(int sides = 6);       //initializes the data members.
                                             Argument received represents the sides of
                                             die, if nothing passed then assume it be 6.
                                             After initializing sidesCount, initialize
                                             value appropriately.
    void roll();                             //Updates the die value randomly.
    int getSidesCount();                     //returns the number of sides of die.
    int getValue();                          //returns the die value.
};
```

Sample Run:

#	Sample Code	Console Output
1	int main()	
2	{	
3	Die d1, d2, d3;	
4	d1.initializeDie();	
5	d2.initializeDie();	
6	d3.initializeDie();	
7	cout << d1.getValue();	4
8	cout << d2.getValue();	1
9	cout << d3.getValue();	6
10	d2.roll();	
11	cout << d1.getValue();	4
12	cout << d2.getValue();	3
13	cout << d3.getValue();	6
14	return 0;	
15	}	

Challenge – 2: *Roman Numeral* (12(8+4))

Consider the following class named *RomanNumeral*, which is responsible for storing and converting roman numbers into decimal number and vice versa.

The *RomanNumeral* has following two data members:

- *romanNumber*: which store the Roman number in it. E.g. MDCLXVI
- *decimalNumber*: which store the equivalent decimal number of roman number stored in *romanNumber*

The *RomanNumeral* has appropriate getter/setter for the data members along with two utility functions, which are responsible for converting roman number to decimal and decimal to roman number. i.e.

1. void convertDecimalNumberToRoman();
   It takes decimal number from *decimalNumber* data member as input and store its output in data member *romanNumber*.
2. void convertRomanNumberToDecimal();

It takes roman number from romanNumber data member as input and store its output in data member decimalNumber.

Code Skeleton

```
class RomanNumeral
{
    char romanNumber[100];
    int decimalNumber;
    void convertRomanNumberToDecimal ( );
    void convertDecimalNumberToRoman ( );
public:
    int getDecimalNumber()
    {
        return decimalNumber;
    }
    const char * getRomanNumber()
    {
        return romanNumber;
    }
    void setDecimalNumber( int num )
    {
        decimalNumber = num;
        convertDecimalNumberToRoman();
    }
    void setRomanNumber( const char * rn )
    {
        strcpy(romanNumber, rn);
        convertRomanNumberToDecimal();
    }
};
```

//Sample Run

```
int main()
{
    RomanNumeral rn;
    rn.setRomanNumber("MDCLXVI");

    cout << rn.getRomanNumber() << '\n';
    // display on console: MDCLXVI

    cout << rn.getDecimalNumber() << '\n';
    // display on console: 1666

    rn.setDecimalNumber(2012);
    cout << rn.getRomanNumber() << '\n';
    // display on console: MMXII

    cout << rn.getDecimalNumber() << '\n';
    // display on console: 2012

    return 0;
}
```

D = 68 C = 67 I = 73 L = 76 M = 77 65 48 = A 90 X = 88

The following content may help you understand the basics of Roman numeral and the conversion process of Roman to Decimal and Decimal to Roman.

Everybody knows, how the Roman Numeral System (invented by Stone Mason) works, though the only time we usually get to show our skills is when the end-credits roll on the big or small screen and the date of film or program production flashes up. It consists of 7 symbols (*I, V, X, L, C, D, M*), each standing for a value.

1 5 10 50 100 500 1000

A number is Roman numeral is just a string of these numerals written in descending order (e.g. M's first followed by D's, etc.). To convert from Roman numerals to decimal, we just walk through the string, adding up the values of the numerals. So, MMVIII represents 2*1000+1*5+3*1 = 2008.

The only complication is the subtractive rule. This rule makes it possible to be written in a more compact form. It allows a numeral of lower value to be placed before one of a higher value to indicate that the lesser should be subtracted from the greater. Thus *IV* means *5 – 1 = 4*, and *XC = 100 – 10 = 90*. Unfortunately, it's not that simple. The rule is that only *I, X and C* can be used subtractively and then only in front of numerals up to ten times as big. So *I* can only be used in front of *V* and *X*, *X* in front of *L* and *C*, and *C* in front of *D*, and *M*.

Pseudo Code Converting Numerals to Decimal:

It's the subtracted rule that makes things complex to develop an algorithm. But soon as you start thinking backwards, that is working from right to left, the difficulty evaporates:

1. Read the character array from right to left.
2. Convert the current character/roman-numeral into its value.
3. Keep a running total, and a record of maximum numeral encountered so far.
4. Take characters one by one:
    - 4.1. If the character is greater than the maximum, add it to the running total and update the maximum numeral.
    - 4.2. If character is less than the maximum, subtract it from the running total.

Numeral	Value
I	1
IV	4
V	5
IX	9
X	10
XL	40
L	50
XC	90
C	100
CD	400
D	500
CM	900
M	1000

Pseudo Code Converting Decimal to Numerals:

For Roman numeral 'X'

1. From the table on the right, find the highest decimal value *V* that is less than or equal to the decimal number *X* and its corresponding roman numeral *N*:
2. Store the roman numeral *N* that you found and subtract its value *V* from *X*.
   X = X – V
3. Repeat stages 1 and 2 until you get zero result of *X*.

Examples: Conversion: Roman Number to Decimal

Example # 1

Starting with Roman String = MMXV Max Numeral = V Maximum Numeral Value = 5

Iteration #	Roman Character	Maximum Numeral	Maximum Numeral Value	Temporary Result
1	X	X	10	15
2	M	M	1000	1015
3	M	M	1000	2015

Example # 2

Starting with Roman String = CCCLIX Max Numeral = X Maximum Numeral Value = 10

Iteration #	Roman Character	Maximum Numeral	Maximum Numeral Value	Temporary Result
1	I	X	10	9
2	L	L	50	59
3	C	C	100	159
4	C	C	100	259
5	C	C	100	359

Examples: Conversion: Decimal to Roman Number

Example # 1: X = 36

Iteration #	Decimal Number (X)	Highest Decimal Value (V)	Highest Roman Numeral (N)	Temporary Result
1	36	10	X	X
2	26	10	X	XX
3	16	10	X	XXX
4	6	5	V	XXXV
5	1	1	I	XXXVI

Example # 2: X = 1996

Iteration #	Decimal Number (X)	Highest Decimal Value (V)	Highest Roman Numeral (N)	Temporary Result
1	1996	1000	M	M
2	996	900	CM	MCM
3	96	90	XC	MCMXC
4	6	5	V	MCMXCV
5	1	1	I	MCMXCVI

In order to test your logic, you may test your program using the following Roman numerals but don't limit yours to these test cases only:

MCXIV = 1114, CCCLIX = 359, MDCLXVI = 1666

IX

It's all talk until the code runs.

[... WARD CUNNINGHAM ...]

Objective:

- Resolving issues related to pointer as data member.
- Focusing on Object's initialization and resource Allocation/de-allocation issues and issues related to Object transition by value/reference.

Challenge – 1: *String ADT*

Complete the class functions given below. Your code will be marked based on the factors (Style Guidelines like .h/.cpp names, indentation, and **most importantly: how smart/clean your logic is?**)

```
class String
{
        char * data;
        int size;
    public:
```

`String ();`	Initializes data and size to 0.	
`String (const char c);`	Initializes data with char c	.2
`String(const char *);`	Initializes the data with received string by allocating memory on heap.	2
`String ( const String & );`		.4
`~String ();`	You know what to do.	.2
`char & at(int index);`	*Index:* Receives the index for string. *Return Value:* reference of array location represented by index	
`const char & at ( const int index) const;`		
`bool isEmpty ( ) const;`	Tells whether string is empty or not *Return Value*: return true if string empty otherwise false.	
`int getLength ( ) const;`	Returns length of the string	
`void display ( ) const;`	Prints the string on console	1
`void makeUpper ( );`	Change all the alphabets to uppercase	
`void makeLower ( );`	Change all the alphabets to lowercase	.4
`void trimLeft ( );`	Removes all the white space characters on the left of string	
`void trimRight ( );`	Removes all the white space characters on the right of string	
`void trim ( );`	Removes all the white space characters on both left and right sides of string	2
`void shrink ( );`	Resize/shrink the array equal to the length of string pointed by data.	1
`void reverse ( );`	It reverses the string stored in the calling object	1
`void reSize ( int );`	You know what to do.	1
`int compare ( String ) const;`	Compare the calling and receive object string and behave just like strcmp	.6
`String left ( int count ) ;`	*Count:* The number of characters to extract from calling object from left side *Return Value:* A String object that contains a copy of the specified range of characters	
`String right ( int count ) ;`		2
`void input ( );`	Takes input from console in calling object.	1
`int toInteger ( ) const;`		.4
`float toFloat ( ) const;`		1
`String concat ( String ) const ;`	It returns the concatenated result of received and calling object without changing calling object.	.6
`void concatEqual ( String );`	It concatenates the received object string with calling object.x	1
`int find ( String subStr, int start=0 ) const;`	Find the first occurrence of substring in the calling String object. By default, search starts from 0 index. If found then return the starting position of subStr found otherwise return -1.	2

`void insert ( int index, String subStr );`	Insert the substring at given index in calling object.	2
`void remove ( int index, int count=1 );`	Remove the characters (how many? Given in count) starting from index	2
`int replace ( String old, String newSubStr );`	Find all the occurrences of old substring and replace it with new substring. Return the count of occurrences found in calling object.	2
`void reverseWords ( );`	Reverse the words. Take care of punctuation marks, String: Hello! How are you? Me? Me fine? Output: olleH! woH era uoy? eM? eM enif?	2
`void changeToNewCharSet ( String cs );`	The function receives a String object containing 26 letter English alphabets. The character at index '0' in 'cs' represents/maps to 'a', character at index '1' maps to 'b' and so on. You need to change calling object string accordingly. Examples: cs: qwertyuiopasdfghjklzxcvbnm Ex-1: before call, calling object: "ok, what" after call, calling object: "ga, viqz" Ex-2: before call, calling object: "vgrt" after call, calling object: "cukz"	2
`String tokenzie( String );`	Returns a String object which contains the substring by extracting it from the calling object String depending upon the delimiter characters passed. See the following Sample Run given below to understand the functionality.	10

};

Sample Run

SR#		Console Output

1	`String s = "This, a ,sample, str-ing.";`
	`while(!s.isEmpty())`
	`{`
	`  String token = s.tokenize(",-");`
	`  token.display();`
	`  cout << '\n';`
	`}`
		Thi
		s
		a
		sample
		str
		ing.

2	For s = "- This, a sample string."; and delimiter = " ,.-"	This a sample string

Persistence and resilience only come from having been given the chance to work through difficult problems.

-- Gever Tulley --

Objective:

- Focus the use of const keyword with data member and functions.
- Targets the use/purpose of class level information.

Challenge – 1: *Toll Plaza*

I hope you know the functioning/concept of a Toll Plaza. In this task, we need to create a system for a toll plaza through which the user will be able to keep track of the vehicles pass through toll plaza and will also be keeping the count and type of vehicles passing through plaza, it will also keep record of the amount of money earned for the current day.

As an initial version, we want to develop an electronic system of toll plazas with basic features. Read the description below and develop the system accordingly.
Write a class TollPlaza for the system. Whenever a vehicle passes a toll plaza it is charged some amount depending on the vehicle type. Assume there are only 5 types of vehicles on the roads Car, Bike, Bus, HeavyDutyTruck, LightDutyTruck (pick-up trucks, wagons etc.). We assign a constant value to each of these categories. Toll fee charged to each of them are stored in variables _carToll, _bikeToll, _busToll, _heavyDutyTruckToll, _lightDutyTruckToll respectively. Since this information is same for every toll plaza, it is good to share this information rather than keeping a separate copy of all of them in every class instance.

For instance,
Car and _carToll can be declared in the manner:
static const int _CAR = 1; static int _carToll = 20; //20 Rs. Will be charged to vehicles of type Car.

Set Bike, Bus, HeavyDutyTruck, LightDutyTruck to const values 2, 3, 4 and 5 respectively. Tolls charged to each of them are 10, 50, 100 and 70 respectively.

Every toll plaza has a variable long int collectedToll representing the toll fee collected at each of them.

Moreover, each toll plaza has to collect a maximum amount of toll per day that is const long int MAX_TOLL. It is different for every toll plaza depending on whether the plaza is located on a busy road or not. Its value will be given whenever a TollPlaza instance is created. Once the collected collectedToll reaches the MAX_TOLL it shows the toll plaza was successful in attaining its today's goal amount. Also maintain an identifier variable int tollPlazaId and address (String) for every toll plaza.
[Have you noticed? how I have named class level identifiers.]

Member Functions for TollPlaza:

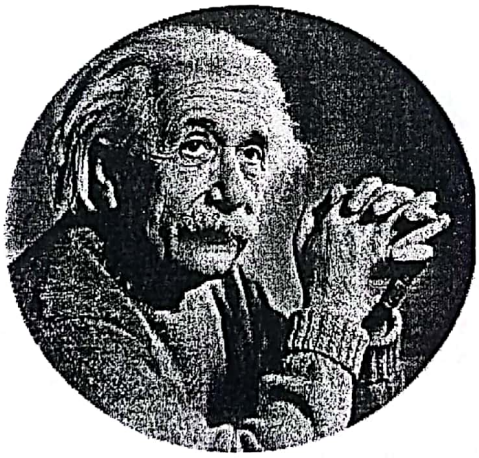
- `TollPlaza(int id, String, long int mt)`
  Receives plaza id, plaza address and plaza target max total
- `void resetTollPlaza()`
  Initializes data members for a new day.
- `void chargeVehicle(int vehicleType )`
  Charges the vehicle according to its type and adds to the collectedToll.
- `long int getCollectedTollToday() const`
  Returns the collected Toll.
- `long int getMaxToll() const`
  Returns the Target Toll to be collected.

When you will be done with the TollPlaza class, then you need to figure out the interface for the Toll Plaza application. For this Purpose, you need to create the following class named as 'TollPlazaSystem'

which contains a static function 'startApplication', the function 'startApplication' will be responsible for creating menu and management for a TollPlaza object.

```
class TollPlazaSystem
{
public:
    static void startApplication()
};
```

```
int main()
{
    TollPlazaSystem::startApplication();
    return 0;
}
```



IT'S NOT THAT I'M SO SMART;
IT'S JUST THAT
I STAY WITH PROBLEMS
LONGER.

[... ALBERT EINSTEIN ...]

Objective:

- Focusing on Array of objects.
- Understanding and implementing weak/strong Aggregation and dealing dynamic memory allocation for objects.

Challenge: *Relation*

Before we move to our actual task in today's lab, we shall have a quick understanding of what Relation is in Mathematics:

Set: Set is a collection of distinct elements.

```
A = {1, 2, 3, 4}  ,      B = {3, 2, 5, 6}
```

Relations: Set of ordered pairs

Let R be a binary relation from A to B is a subset of A*B
And relation R1, R2 are subset of A * B.

```
R1 = {(1, 3), (1, 2), (2, 6)}
R2 = {(1, 2), (1, 3), (2, 5)}
```

And relation R3 is subset of A * A.

```
R3 = {(1, 4), (1, 1), (2, 4)}
```

Formally, a relation R over a set X can be seen as a set of ordered pairs *(x, y)* of members of X. The relation R holds between x and y if (x, y) is a member of R. For example, the relation "is less than" on the natural numbers is an infinite set R_{less} of pairs of natural numbers that contains both *(1,3)* and *(3,4)*, but neither *(3,1)* nor *(4,4)*.

Some important ***properties*** that a relation R over a set A may have are:

- A binary relation R is called **reflexive** if and only if $\forall\, a \in A, aRa$ So, a relation R is reflexive if it relates every element of A to itself.
  - Example: The relation R = {(1,1), (1,2), (2,2), (3,3), (3,1)} on the set A = {1,2,3}
- A binary relation R on a set A is called **irreflexive** if aRa doesn't hold for any $a \in A$. This means that there is no element in R which is related to itself.
  - Example: The relation R = {(1,2), (2,1), (1,3), (2,3)} on the set A = {1,2,3}
- A binary relation R on a set A is called **symmetric** if $\forall\, a, b \in A$ it holds that if aRb then bRa.
  - Example: The relation R = {(1,2), (2,1), (1,3), (3,1)} on the set A = {1,2,3}
- A binary relation R on a set A is called **transitive** if $\forall\, a, b, c \in A$ it holds that if aRb and bRc then aRc.
  - Example: The relation R = {(1,2), (1,3), (2,2), (2,3), (3,3)} on the set A = {1,2,3}

Today's Coding Task:

Our objective is to make an ADT 'Relation' which implements the relation concept of Mathematics and test its properties.

Considering the above discussion, programmatically, a Relation can be considered as a set/array of ordered pairs.

Following classes/design we shall follow to implement the required features of Relation class.

```
class OrderedPair   // used to hold an ordered pair
{
    int a;
    int b;
public:
    OrderedPair(int n=0,int m=0)
    { a = n;   b = m;   }
```

```
    //Define yourself the standard getter/setter for a and b
};

class Relation  // used to represent a Relation over a set A*A
{
    Set setA;
        //Represents the set on which Relation is define. Since there is only one
        set. It is understood that the Relation is define on A*A.
        Set class interface is given at the end. Only the provided interface is
        allowed to be used in Relation implementation.

    OrderedPair * orderedPairList;
        //points to the array of objects of type OrderedPair (Relation elements).

    int capacity;

    int noOfOrderedPair;

    void resize();
        //resize the Relation (array of Ordered Pairs) as per your choice.
    void shrink();
        //shrink the Relation (array of Ordered Pairs) as per your choice.

public:
    Relation(Set);
        //Receives the set object on which Relation will be defined.
        Initializes capacity and noOfOrderedPair to zero.

    Relation( const Relation & );

    ~Relation();

    void insert (const OrderedPair & ref);
        //stores an ordered pair in the Relation. Must check the uniqueness and
        validity of ordered pair.

    void remove (const OrderedPair & ref);
        //removes the given ordered pair from the relation.

    int getCardinality() const;
        //returns the number of ordered-pair/elements in the relation.

    bool compare (const Relation &) const;
        //returns true if received and calling object are equal relations
        otherwise return false.

    bool isReflexive () const;
        //returns true if *this object is reflexive otherwise return false.

    bool isSymmetric () const;
        //returns true if *this object is symmetric otherwise return false.

    bool isIrreflexive () const;
        //returns true if *this object is irreflexive otherwise return false.

    bool isTransitive () const;
        //returns true if *this object is transitive otherwise return false.

    void print () const;
        //prints the relation on console.
        //Sample Output format: {(1,2), (1,3), (2,2), (2,3), (3,3)}
};
```

Relation

CC-112-L: Object Oriented Programming Lab
BSSE F22 – Semester: Spring 2023
Lab 07

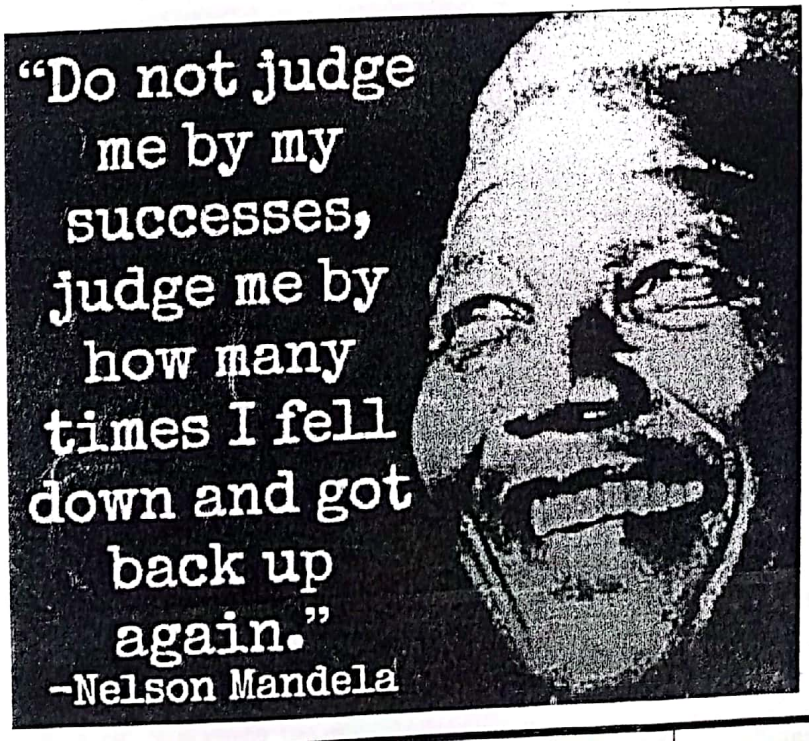
Issue Date/Time: 20-Oct-2023
Marks: 20.5

Set Class Interface

```
class Set
{
    Array data;
    int noOfElements;
public:
    Set( int cap = 0 ):data(cap)
    {  noOfElements=0;  }
    void insert ( int element );
    void remove ( int element );
    void print ( ) const;
    int getCardinality() const;
    bool isMember ( int val ) const;
    int isSubSet ( Set s2 ) const;
              //return 1 if *this is subset of s2.
              //return -1 if s2 is subset of *this.
              //return 0 if no one is subset.
              //return 2 if it is improper subset.
    void reSize ( int newcapacity );
    Set calcUnion ( const Set & s2 ) const;
    Set calcIntersection ( const Set & s2 ) const;
    Set calcDifference ( const Set & s2 ) const;
    Set calcSymmetricDifference ( const Set & s2 )
const;
    void displayPowerSet ( ) const;
};
```

```
class Array
{

    int size;
    int * data;
public:
    Array(int=0, ...);
    Array(const Array &);
    int & getSet(int);
    const int & getSet(int) const;
    void reSize(int);
    int getSize() const;
    ~Array();
};
```



Objective:

- Introducing flexibility/usability in the String class by introducing different operators for it.

It's a Close Book Lab

Challenge: *String*

An updated version of *String*, which will provide basic functionalities related to strings.

Note: You are not allowed to use any library functions related to strings.

```
class String
{
    char * data;
    int size;
public:
```

`String ();`	Initializes data to nullptr and size to 0.
`String (const char c);`	Initializes data with char c
`String (const char *);`	Initializes the data with received string by allocating memory on heap.
`String ( const String & );`	Copy Constructor
`String ( String && );`	Move Constructor
`String& operator = (const String &);`	Copy Assignment
`String& operator = (String &&);`	Move Assignment
`~String ();`	You know what to do.
`void input ();`	Takes input from console in *this object.
`void shrink();`	Resize/shrink the array equal to the length of string pointed by data.
`char & operator [] (const int index);`	Index: Receives the index for string. Return Value: reference of array location represented by index
`const char & operator [] ( const int index) const;`	
`bool operator ! () const;`	Tells whether string is empty or not? An empty string is one which has data==nullptr or data[0]=='\0'. Return Value: return true if string empty otherwise false.
`int getLength () const;`	Returns length of the string
`int getSize () const;`	Returns the value in the size attribute.
`void display () const;`	Prints the string on console
`int find (const String & subStr, const int start=0 ) const;`	Find the first occurrence of substring in the calling String object. By default, search starts from 0 index. If found then return the starting position of subStr found otherwise return -1.
`void insert (const int index, const String & subStr);`	Insert the substring at given index in calling object.
`void remove (const int index, const int count=1);`	Remove the characters (how many? Given in count) starting from index
`int replace (const String & old, const String & newSubStr );`	Find all the occurrences of old substring and replace it with new substring. Return the count of occurrences found in calling object.
`void trimLeft ();`	Removes all the white space characters on the left of string
`void trimRight ();`	Removes all the white space characters on the right of string
`void trim ();`	Removes all the white space characters on both left and right sides of string
`void makeUpper ();`	Change all the alphabets to uppercase
`void makeLower ();`	Change all the alphabets to lowercase
`void reverse ();`	It reverses the string stored in the calling object
`void resize (int);`	You know what to do.
`int operator == (const String & s2 ) const;`	Compare the calling and receive object string and behave just like strcmp.
`bool operator > (const String & s2 ) const;`	Returns true if calling object greater than received object otherwise false.

bool operator < (const String & s2) const;	Returns true if calling object less than received object otherwise false.
bool operator >= (const String & s2) const;	Returns true if calling object greater than or equal to received object otherwise false.
bool operator <= (const String & s2) const;	Returns true if calling object less than or equal to received object otherwise false.
bool operator != (const String & s2) const;	Returns true if calling object is not equal to received object otherwise false.
String left (const int count) ;	Count: The number of characters to extract from calling object from left side Return Value: A String object that contains a copy of the specified range of characters
String right (const int count) ;	
String operator + (const String & s2) const;	It returns the concatenated result of received and calling object without changing calling object.
void operator += (const String & s2);	It concatenates the received object string with calling object.

String & operator = (const long long int num);	Stores the received number as a string.
	String s;
	s = (long long int)-592950;
	s.display(); //prints -592950
	s = (long long int)489;
	s.display(); //prints 489

String & operator = (const double num);	Stores the received fractional value as a string.
	String s;
	s = 20.5;
	s.display(); //prints 20.5
	s = (double)489;
	s.display(); //prints 489

explicit operator long long int () const;	Converts the integral value stored in calling object to long long int and returns the integral value.
explicit operator double () const;	Converts the fractional value stored in calling object to double and returns it.

explicit operator bool () const;	Helps to use String in a condition like true/false. Return true if string is not empty. Returns false if it is empty.
	For Example:
	String s("Hello");
	if (s)
	cout<<"Success";//display success

String operator () (const String & delim);	Tokenize the string like strtok.

Sample Run

```
int main()
{
    String str(" This,Q--a sample string. nothing");
    String token;
    cout << "String = " << str << "\n";
    while ( str )
    {
        token = str(".-,");
        cout << "Token = " << token;
        cout << "\n";
    }
    cout << "String: " << str;
    return 0;
}
```

Console Output

```
String =  This,Q--a
sample string. nothing
Token =  This
Token = Q
Token =
Token = a sample string
Token =  nothing
String:
```

};

Global functions part of String.h and implementation in String.cpp

istream & operator >> (istream &, String &);	Console Input string
ostream & operator << (ostream &, const String &);	Console Output string

Can the above may also be used to store/retrieve data from hard disk?

They can, because they think they can.

Roll-No BSEF22M023

Objective:

- It will help you understand the benefits that we get through inheritance relationship.

Challenge – 1: *Counter* **(4)**

The class `'Counter'` is behaving as a counter which increment only but doesn't decrement the counter. Provide a new class named as `'FBCounter'`, which doesn't only increment but decrement the counter as well.

You must reuse the `'Counter'` class while implementing the new class by providing two version of `'FBCounter'` class:

- One using whole-part relationship
- Second using inheritance relationship

Note: You are not allowed to change anything in class `'Counter'`.

Counter Class	Sample Run	Sample Output

class Counter
{
int value;
public:
Counter(int i=0):value(i)
{}
void increment()
{ value++; }
void reset()
{ value=0; }
void startAt(int i)
{ value = i; }
int getCounterValue()
{ return value; }
};
	int main()
	{
	FBCounter c;
	c.increment();
	c.increment();
	c.decrement();
	c.increment();
	cout<<c.getCounterValue()<<"\n";
	return 0;
	}
		2

Challenge – 2: *Package Delivery* **(10)**

Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated. Create an inheritance hierarchy to represent various types of packages. Use *Package* as the base class of the hierarchy, then include classes *TwoDayPackage* and *OvernightPackage* that derive from *Package*. Base class *Package* should include data members representing the *name*, *address*, *city*, *state* and *ZIP code* for both the sender and the recipient of the package (idea: A *Person* class should have this information like *name*, *address*, *city*, *state*, *zip* and then compose it in *Package* class), in addition to data members that store the weight (in ounces) and cost per ounce to ship the package. *Package's* constructor should initialize these data members. Ensure that the weight and cost per ounce contain positive values. *Package* should provide a public member function *calculateCost* that returns a double indicating the cost associated with shipping the package. Package's *calculateCost* function should determine the cost by multiplying the weight by the cost per ounce. Derived class *TwoDayPackage* should inherit the functionality of base class Package, but also include a data member that represents a flat fee that the shipping company charges for two-day-delivery service. *TwoDayPackage's* constructor should receive a value to initialize this data member. *TwoDayPackage* should redefine member function *calculateCost* so that it computes the shipping cost by adding the flat fee to the weight-based cost calculated by base class *Package's calculateCost* function. Class *OvernightPackage* should inherit directly from class *Package* and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. *OvernightPackage* should redefine member function *calculateCost* so that it adds the additional fee per ounce to the standard cost per ounce before calculating the shipping cost.

Write a test program that creates objects of each type of Package and tests member function *calculateCost*.

12
5.6
72
67.2

Success is not the absence of Failure; it's the persistence through Failure.

Objective:

- Learning the basic advantage of using runtime polymorphism.

Challenge – X: *Account Hierarchy* **(10.5)**

Banks offer various types of accounts, such as savings, checking, certificate of deposits, and money market, to attract customers as well as meet with their specific needs. Two of the most commonly used accounts are savings and checking. Each of these accounts has various options. For example, you may have a savings account that requires no minimum balance but has a lower interest rate. Similarly, you may have a checking account that limits the number of checks you may write. Another type of account that is used to save money for the long term is certificate of deposit (CD).

In this challenge, you use abstract classes and pure virtual functions to design classes to manipulate various types of accounts. For simplicity, assume that the bank offers three types of accounts: savings, checking, and certificate of deposit, as described next.

Savings accounts: Suppose that the bank offers two types of savings accounts: one that has no minimum balance and a lower interest rate and another that requires a minimum balance and has a higher interest rate.

Checking accounts: Suppose that the bank offers three types of checking accounts: one with a monthly service charge, limited check writing, no minimum balance, and no interest; another with no monthly service charge, a minimum balance requirement, unlimited check writing and lower interest; and a third with no monthly service charge, a higher minimum requirement, a higher interest rate, and unlimited check writing.

Certificate of deposit (CD): In an account of this type, money is left for some time, and these accounts draw higher interest rates than savings or checking accounts. Suppose that you purchase a CD for six months. Then we say that the CD will mature in six months. Penalty for early withdrawal is stiff.

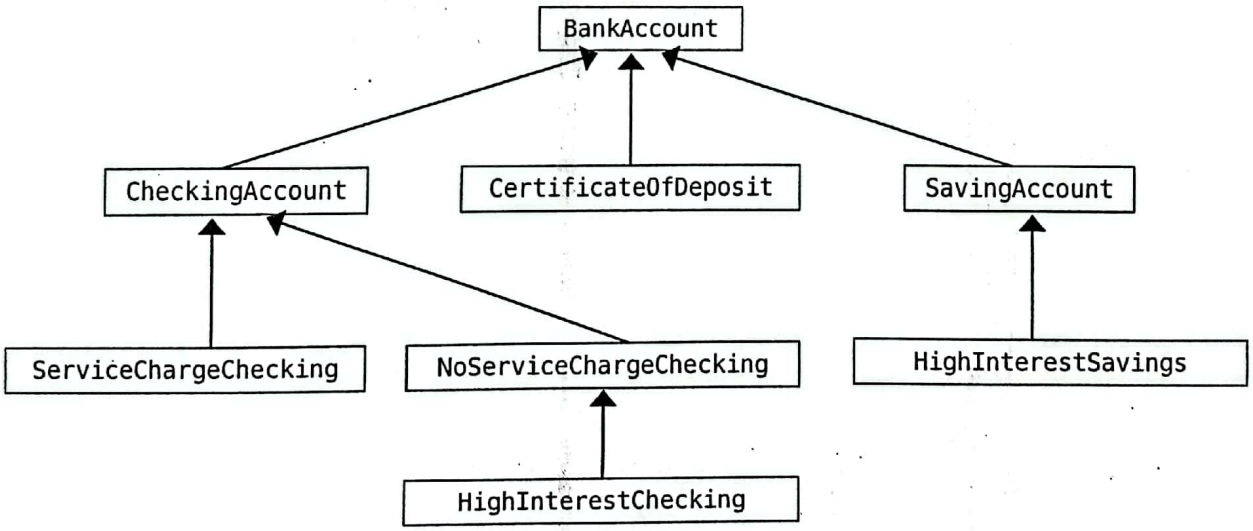


Figure: Inheritance Hierarchy

In the figure above, the classes `BankAccount` and `CheckingAccount` are abstract. That is, we cannot instantiate objects of these classes. The other classes in the diagram are not abstract.

`BankAccount`: Every bank account has an account number, the name of the owner, and a balance. Therefore, instance variables such as name, accountNumber, and balance should be declared in the abstract class bankAccount. Some attributes/operations common to all types of accounts are define here like: account owner's name, account number, and account balance; make deposits; withdraw money; and create monthly statement. So include functions to implement these operations. Some of these functions will be pure virtual.

`CheckingAccount`: A checking account is a bank account. Therefore, it inherits all the properties of a bank account. Because one of the objectives of a checking account is to be able to write checks, include the pure virtual function writeCheck to write a check.

ServiceChargeChecking: A service charge checking account is a checking account. Therefore, it inherits all the properties of a checking account. For simplicity, assume that this type of account does not pay any interest, allows the account holder to write a limited number of checks each month, and does not require any minimum balance. Include appropriate named constants, instance variables, and functions in this class.

NoServiceChargeChecking: A checking account with no monthly service charge is a checking account. Therefore, it inherits all the properties of a checking account. Furthermore, this type of account pays interest, allows the account holder to write checks, and requires a minimum balance.

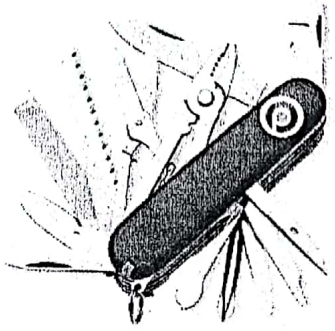
HighInterestChecking: A checking account with high interest is a checking account with no monthly service charge. Therefore, it inherits all the properties of a no service charge checking account. Furthermore, this type of account pays higher interest and requires a higher minimum balance than the no service charge checking account.

SavingsAccount: A savings account is a bank account. Therefore, it inherits all the properties of a bank account. Furthermore, a savings account also pays interest.

HighInterestSavings: A high-interest savings account is a savings account. Therefore, it inherits all the properties of a savings account. It also requires a minimum balance.

CertificateOfDeposit: A certificate of deposit account is a bank account. Therefore, it inherits all the properties of a bank account. In addition, it has instance variables to store the number of CD maturity months, interest rate, and the current CD month.

Write the definitions of the classes described in this challenge and a program to test your classes.



No to "Jack of all trades, master of none"

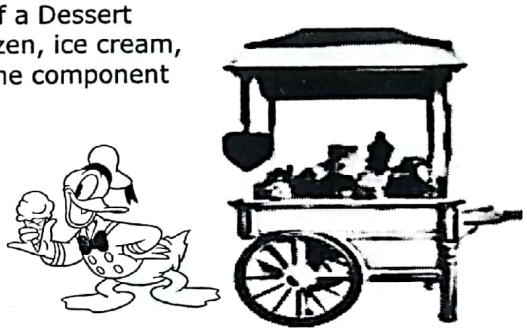
Yes to "Jack of all trades, master of one"

Objective:

- Learning the basic advantage of using runtime polymorphism.

Challenge – X: *Dessert Shoppe* **(53)**

In this Laboratory, you will be writing software in support of a Dessert Shoppe, which sells candy by the pound, cookies by the dozen, ice cream, and sundaes (ice cream with a topping). You will develop one component of this software i.e. checkout system. The interfacing will be dealt by another software development team so you don't have to worry about it.



To do this, you will implement an inheritance hierarchy of classes derived from a `DessertItem` abstract base class. The `Candy`, `Cookie`, and `IceCream` classes will be derived from the `DessertItem` class.
The `Sundae` class will be derived from the `IceCream` class.
You will also write a `Checkout` class which maintains a list (a resizable array) of `DessertItem's`.

The DessertItem Class

The `DessertItem` class is an *abstract base class* from which specific types of `DessertItems` can be derived. It contains only one data member, a name. It also defines a number of methods. All of the `DessertItem` class methods except the `getCost()` are defined in a generic way in the class, `DessertItem.cpp`. The `getCost()` method is an *abstract method* that is not defined in the `DessertItem` class because the method of determining the costs varies based on the type of item.

The DessertShoppe Class

It contains constants such as the tax rate as well the name of the store, the width used to display the costs of the items and item name width to be displayed on the receipt. Your code should use these constants wherever necessary! The `DessertShoppe` class also contains the `centsToDollars` method, which takes an integer number of cents and returns it as a String formatted in dollars and cents. For example, 105 cents would be returned as "1.05".

The Derived Classes

- The `Candy` class should be derived from the `DessertItem` class. A `Candy` item has a *weight* and a *price per pound*, which are used to determine its *cost*. For example, 2.30 lbs. of fudge @ .89 $/lb. = 205 cents.
- The `Cookie` class should be derived from the `DessertItem` class. A `Cookie` item has a *number* and a *price per dozen* which are used to determine its *cost*. For example, 4 cookies @ 399 cents /dz. = 133 cents.
- The `IceCream` class should be derived from the `DessertItem` class. An `IceCream` item simply has a *cost*.
- The `Sundae` class should be derived from the `IceCream` class. The *cost* of a Sundae is the *cost of the IceCream* plus the *cost of the topping*.

The Checkout Class

The `Checkout` class behaves as a point of sale, provides methods to enter dessert items into the cash register, clear the cash register, maintains the number of items, maintains the total cost of the items (before tax), maintains the total tax for the items, and maintains a string representing a receipt for the dessert items. The `Checkout` class should use an array to store the `DessertItem's` and their count of orders for a particular item.

Assumptions and General Guidelines

- C++ string class has been used in few places of code provided in this document, not good for you ☺. You must use String class.
- All type/classes of dessert items will store their price in cents but while printing the bill, the cost will be converted and displayed in dollars.

- All the item prices (including total tax and total bill) on receipt will be printed as rounded to nearest cent. But total bill and tax will be calculated on actual values to get the maximum possible accuracy. (see the sample output of receipt in this regard.)
- Other than what has been specified, you are free to decide about adding any private **data member(s)** or any **ctor/dtor** or any **member functions** needed for your classes but you got to justify your decision to the TA, failing to give some acceptable/reasonable justification will result in less or zero marks. Note: This liberty is not for `String` and `Array` class.
- You may assume that arguments passed to functions will be valid.
- You may assume that the items stored in the checkout will be unique.
- Just to save printing paper, I have defined function in header files but you must do and should already know, where to put those function definitions.
- You have mainly three tasks to implement in this lab:
  - Implement the hierarchy of classes for dessert items.
  - Checkout class and DessertShoppe class
  - Bill printing which is part of checkout class. You strictly got to follow the format as shown in the sample run.
  - You don't need to worry about the interface that how items will be scanned on point of sale and will be added to checkout class. This part is assigned to another development team which will use your classes. For your help, I have given a sample run which should produce the output desired, which will also help you understand the purpose of classes.

***** *Classes* *****

DessertItem.h

```
class DessertItem
{
    String name ;
public:
    DessertItem(String = "");
    String getName() const;
    virtual double getCost() const = 0;
    virtual double getTax() const;
};
```

DessertShoppe.h

```
class DessertShoppe
{
    static double _TAX_Rate;
    static const String _STORE_Name;
    static const int _MAX_ITEM_NAME_WIDTH;
public:
    static string centsToDollars(int cents)//for your help: may be a cattywampus
    {
        String s = "";

        if (cents < 0)
        {
            s += "-";
            cents *= -1;
        }
        int dollars = cents/100;
        cents = cents % 100;

        if (dollars > 0)
            s += dollars;

        s +=".";

        if (cents < 10)
            s += "0";

        s += cents;
        return s;
    }
```

```
};
```

Class Checkout

Maintains a list of DessertItem references. There is no limit to the number of DessertItem's in the list

```
class Checkout
{
    DessertItem * * list;  // list of different Dessert Items
    Array countPerItem;    //count of items at list[i] is stored at countPerItem[i]
    int itemsCount=0;
    int listCapacity;
public:
    Checkout();     //Initializes the object with some fix size of DessertItem array
    void clear();  //Clears the Checkout to begin checking out a new set of items i.e. the next
                   //customer in line.
    void enterItem(const DessertItem & item, int cnt=1);    //A deep copy of DessertItem is
                                                            //added to the end of the list of
                                                            //items with default count of 1.
    String toString();
                /*Returns a String representing a receipt for the current list of items. i.e. a
                String representing a receipt for the current list of DessertItem's with the
                name of the Dessert store, the items purchased, the tax, and the total cost. See
                the sample run output for further detail.*/
    double totalCost();    //Returns total cost of items
    double totalTax();     //Returns total tax on items
};
```

Sample Run

```
//Driver.cpp
int main()
{
    DessertShoppe::setTaxRate(17.5);
    Checkout checkout;
    checkout.enterItem (Candy(2.25, 399, "Fall 22: Special Platter of Candies : Pop Rocks
Candy+Nerds Candy+Swedish Fish Candy+Candy Corn"),3);
    checkout.enterItem (IceCream(105, "Vanilla Ice Cream"), 10);
    checkout.enterItem (Sundae(145, 50, "Chocolate Chip Ice Cream with Pineapple and Almonds"),
2);
    checkout.enterItem (Cookiee(399, 15, "Oatmeal Raisin Cookies"), 2);

    cout<<checkout.toString();

    checkout.clear();

    checkout.enterItem(IceCream(145, "Strawberry Ice Cream"));
    checkout.enterItem(Sundae(105, 50, "Caramel"));
    checkout.enterItem(Candy(1.33, 89, "Gummy Worms"));
    checkout.enterItem(Cookiee(44, 30, "Chocolate Chip Cookies"));
    checkout.enterItem(Candy(1.55, 209, "Salt Water Taffy"));
    checkout.enterItem(Candy(3.0, 109, "Candy Corn"));

    cout<<"\n\n"<<checkout.toString();

    return 0;
}
```

Sample Run Output

```
   Fall-22 : CS & SE : Dessert Shoppe
---------------------------------------
Fall 22: Special Platter.....26.94 = 3 x 8.98
 of Candies : Pop Rocks
Candy+Nerds Candy+Swedis
h Fish Candy+Candy Corn
Vanilla Ice Cream.................10.50 = 10 x 1.05
Chocolate Chip Ice Cream.....3.90 = 2 x 1.95
```

```
 with Pineapple and Almon
ds
Oatmeal Raisin Cookies.............9.98 = 2 x 4.99
Tax~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~8.98
Total Cost~~~~~~~~~~~~~~~~~~~~~~~~~~60.29

   Fall-18 : CS & SE : Dessert Shoppe
----------------------------------------
Strawberry Ice Cream...............1.45 = 1 x 1.45
Caramel............................1.55 = 1 x 1.55
Gummy Worms........................1.18 = 1 x 1.18
Chocolate Chip Cookies.............1.10 = 1 x 1.10
Salt Water Taffy...................3.24 = 1 x 3.24
Candy Corn.........................3.27 = 1 x 3.27
Tax~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~2.06
Total Cost~~~~~~~~~~~~~~~~~~~~~~~~~~13.86
```

"Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. What you eat before you step onto the scale determines how much you will weigh, and the software-development techniques you use determine how many errors testing will find."

-- Steve McConnell, Code Complete –

Objective:

- Exploring the power of C++ Templates.

Challenge – X: *Dictionary* **(10.5)**

In our daily life, we normally use the term dictionary as English/Urdu words and its meanings where a word is a key and its meaning is its value.
Actually, in general, anything, which maintains a key-value pair, can be treated as a dictionary.
Let say, we want to maintain a list of grocery items with their count as follows:

Dictionary	
Key	**Value**
Apples	15
Oranges	20
Bananas	30
Watermelon	10

In the dictionary given above key is the name of some fruit and value is the count of fruit.
So, in this dictionary, we are interested to find the count of any given fruit.

In this task, we are interested in developing an ADT, which can store *key-value* pair in it.
For this purpose, you need to develop the following classes:

DictionaryPair *object represents a key-value pair. Its implementation is already provided to you.*
GenericDictionary *object represents a collection (array of objects of type DictionaryPair) of key-value pairs.*

```
template <typename T, typename R>
class DictionaryPair
{
    T key;
    R value;
public:
    DictionaryPair () {}
    DictionaryPair (T k, R v) : key(k), value(v)
    { }
    void setKey (T k)
    { key = k; }
    void setValue (R v)
    { value = v ; }
    T getKey ()
    { return key; }
    R getValue ()
    { return value; }
};
```

```
template <typename T, typename R>
class GenericDictionary
{
    DictionaryPair <T, R> * data;
    int noOfItems;
    int capacity;
    void reSize ();
public:
    GenericDictionary ();
    void addPair (T k, R v);
    R getValue (T k);
    void print ();
    ~GenericDictionary ();
    bool isFull ();
    bool isEmpty ();
};
```

GenericDictionary class Description:

`data`	Points to an array of objects of type `DictionaryPair`.
`noOfItems`	Represents the number of key-value pairs of type `DictionaryPair` pointed by `data` pointer.
`capacity`	Represents the size of array pointed by `data` pointer. Initially the `capacity` is zero. Resizing will automatically be done by `addPair` function, which will automatically be doubled whenever the resize operation will be called. So, at start when `capacity` is 0, it will make it 1 and then will be doubled for non-zero capacities.
`GenericDictionary()`	Initializes the `capacity` to 0 and other data members accordingly.
`addPair()`	Adds the receive key-value in the dictionary. If the key is already available then it overwrites the existing key-value pair.
`getValue()`	Returns the value for the received key. If key is not found then it exits the application.
`Print()`	Prints all the key value pairs on console.
`~GenericDictionary()`	You should know what to do.
`isFull()`	Returns true if array pointed by `data` is full otherwise returns false.
`isEmpty()`	Returns true if there are no key-value pairs in the dictionary otherwise returns false.

Sample Run

Sample Code

```
int main()
{
    GenericDictionary<String, int> grocery;
    cout<<"Grocery List";
    cout<<"\n============\n";
    grocery.addPair("Oranges", 12);
    grocery.addPair("Apples", 18);
    grocery.addPair("Bananas", 78);
    grocery.addPair("Apricot", 316);
    grocery.print();
    cout<<"\nGrocery List After Modification";
    cout<<"\n==============================\n";
    grocery.addPair("Apples", 9);
    grocery.print();
    cout<<'\n'<<grocery.getValue("Apples")<<'\n';
    cout<<'\n'<<grocery.getValue("Mangoes")<<'\n';
    return 0;
}
```

Console Output

```
Grocery List
============
Oranges 12
Apples  18
Bananas 78
Apricot 316

Grocery List After Modification
==============================
Oranges 12
Apples  9
Bananas 78
Apricot 316

9
```

Objective:

- Handling file contents written in binary mode and exploring an application related to image processing.

Challenge – X: *Image Rotate* **(2, 5, 10)**

Give us an executable file named as rotate, which rotate a PGM (text or binary) image 90 degree clockwise.

Sample Run:

```
C:/> rotate oops.pgm
```

PGM image/file format

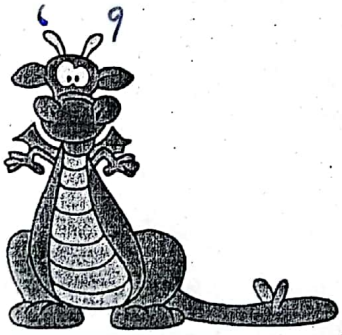
PGM stands for Portable Gray Map; it is a simple portable format for gray-tone images whose data is laid out in file in the following way:

1. ***Magic Number:*** The first line contains the Format i.e., P2 means PGM file (whose pixel values are written in text format) the other option P5, which means PGM file (whose pixel values are written in binary format)
2. ***Comment:*** The next line contains comment, which must start from symbol #. But it is optional, a file may or may not have comment written in it.
3. ***Number of Columns and Rows:*** The next line to comment contains two numbers separated by a white space. They represent the columns and rows of the image respectively.
4. ***Maximum Value:*** The next line to rows and columns contains maximum pixel value (intensity of grey scale (0~255)) that a pixel contains in this image.
5. ***Pixels:*** After the maximum value, there is a list of pixel values separated by at least one space (text mode have spaces not binary).

Consider the following file, which display OOPS, when opened in an image viewer which support pgm file format.

```
P2
# OOPS.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  3  0  0  7  0  0  7  0  0 11  0  0 11  0  0 15  0  0  0  0
0  3  0  0  3  0  0  7  0  0  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  3  0  0  7  0  0  7  0  0 11  0  0  0  0  0  0  0  0 15  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11  0  0  0  0  0 15 15 15 15  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Example



Imaged before rotation

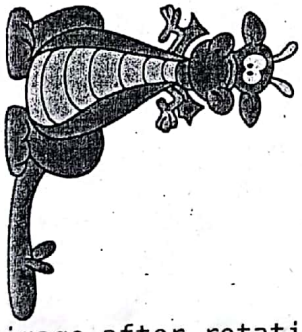


image after rotation

Coding is an art: craft it with passion.