Nauman

Objective:

- It will help practice solution to the issues related to initialization of objects and character arrays too ☺.
- Use of enumeration.

Task-1:

Add the constructor feature to the tasks given in Practice File 5.

Task-2:

It would be perfectly reasonable for the Time class to represent the time internally as the number of seconds since midnight rather than the three integer values: hour, minute and second. Clients could use the same public methods and get the same results. Modify the Time class to implement the time as the number of seconds since midnight and show that there is no visible change in functionality to the clients of the class. ***[Note: This exercise nicely demonstrates the virtues of Abstraction.]***

Task-3:

Write a program that allows two players to play a game of tic-tac-toe. Use a two-dimensional char array with three rows and three columns as the game board.
You all know what is Tic Tac Toe game, so here is the sample run which will guide you about game interface and at the end you will see a couple of possible classes used to implement the game.

Game Sample Run

```
Enter Player 1 Symbol: X
Enter Player 2 Symbol: Y
1    2    3
4    5    6
7    8    9

Player 1 Turn: Enter Cell #: 1
X    2    3
4    5    6
7    8    9

Player 2 Turn: Enter Cell #: 2
X    Y    3
4    5    6
7    8    9

Player 1 Turn: Enter Cell #: 9
X    Y    3
4    5    6
7    8    X

Player 2 Turn: Enter Cell #: 5
X    Y    3
4    Y    6
7    8    X

Player 1 Turn: Enter Cell #: 6
X    Y    3
4    Y    X
7    8    X

Player 2 Turn: Enter Cell #: 7
X    Y    3
4    Y    X
Y    8    X

Player 1 Turn: Enter Cell #: 3

Game Won by Player : 1
Program ended with exit code: 0
```

You are already familiar with this game in your PF course. Those who missed it can do it now and those who did it will do it in Object Oriented way and will be able to compare the advantages over procedural programming. following classes are suggested to be completed.

```
//GameBoard.h
enum PlayerTurn
{FIRST_PLAYER=1,SECOND_PLAYER=2};
enum GameStatus { DRAW, WIN, IN_PROGRESS};

class GameBoard
{
private:
    char data[3][3]; // Array used for board
    GameStatus gameStatus = IN_PROGRESS;
    int validMovesCount=0;

public:
    GameBoard();
    // initialize the board with following
values
    // 1 2 3
    // 4 5 6
    // 7 8 9

    void displayBoard();
    // display the board on console

    void markBoard(char pos, char
playerSymbol)
    // Mark the specified position in board
with symbol

    bool isValidPosition(char pos);
    // checks, whether the position is
within the range i.e 1 to 9

    bool isAlreadyMarked(char pos);
    // checks, whether the cell is already
marked or not

    int getValidMovesCount();
    // returns the count of valid moves on
board

};
```

```
//TicTacToe.h
class TicTacToe
{
public:
    void playGame();
        //This function provides interface
to play Tic Tac Toe Game
        // you have to create objects and
manage them in this function
};
```

```
//driver.cpp
int main()
{
    TicTacToe tic;
    tic.playGame();
    return 0;
}
```

You should look below only when you are done with yours Tic Tac Toe Game!

I am sharing my `TicTacToe::playGame` implementation with you, so you compare your implementation with the code flow given below in terms of reading and coding practices. It is to be noted, I may further divide the `playGame` function for functions like `symbolInput`, `switchPlayerTurn` etc. But I let it l this so you may get the flow in one go.

Note: You are not allowed to add/remove/update any data member in any of the classes. If you nee to define some functions to make them **cohesive** then you may define them in private block. In short change in public interface and data members until and unless specified otherwise. You got to follow this principle throughout the semester.

```
void TicTacToe::playGame()
{
    GameBoard board;
    PlayerTurn currentPlayer;

    char player1Symbol;
    char player2Symbol;
    bool validSymbol;
    do
```

Nouman

```
    {
        cout<<"Enter Player 1 Symbol: ";
        cin>>player1Symbol;
        validSymbol = isValidPlayerSymbol(player1Symbol);
        if (!validSymbol)
            cout<<"Not a Valid Symbol\n";
    }
    while(!validSymbol);

    do
    {
        cout<<"Enter Player 2 Symbol: ";
        cin>>player2Symbol;
        validSymbol = isValidPlayerSymbol(player2Symbol);
        if (!validSymbol || player2Symbol==player1Symbol)
            cout<<"Not a Valid Symbol\n";
    }
    while(!validSymbol || player2Symbol==player1Symbol);

    char cellNumber;
    currentPlayer = FIRST_PLAYER;
    char currentPlayerSymbol = player1Symbol;

    while(board.getGameStatus()==IN_PROGRESS)
    {
        board.displayBoard();
        bool validMove=false;
        do
        {
            cout<<"\nPlayer "<<currentPlayer<<" Turn: Enter Position: ";
            cin>>cellNumber;
            if (board.isValidPosition(cellNumber) && !board.isAlreadyMarked(cellNumber))
            {
                board.markBoard(cellNumber,currentPlayerSymbol);
                validMove = true;
            }
        }
        while (!validMove);

        if (board.getGameStatus()==WIN)
        {
            board.displayBoard();
            cout<<"\nGame Won by Player : "<<currentPlayer<<"\n";
        }
        else if (board.getGameStatus() == DRAW)
        {
            cout<<"\nGame DrawX\n";
        }
        else
        {
            currentPlayerSymbol = (currentPlayer==FIRST_PLAYER? player2Symbol: player1Symbol);
            currentPlayer = (currentPlayer==FIRST_PLAYER? SECOND_PLAYER: FIRST_PLAYER);
        }
    }
}

bool TicTacToe::isValidPlayerSymbol(char symbol)
{
    return !(symbol>='1' && symbol<='9');
}
```

We may come up with some design improvements after few weeks.

Issue Date: 7-Oct-2019

Nouman

Objective:

- Usage of constructor/destructor.
- Targets the object transition by reference.
- Character array manipulation.

Task-1: *Array ADT*

The Array ADT that we discussed today.

Private Members:

- `int *data;`
  pointer to an array of integers
- `int capcity;`
  capacity of array pointed by data
- `bool isValidIndex( int index )`
  return true if index is within bounds otherwise false.

Public Members:

The class 'Array' should support the following operations

- `Array( int cap = 0 );`
  Sets 'cap' to 'capacity' and initializes rest of the data members accordingly.
  If user sends any invalid value (-ve value) then sets the cap to zero.
- `~Array()`
  Free the dynamically allocated memory.
- `int & getSet(int index);`
  insert value at given index of array.
- `int getCapacity()`
  returns the size of array.
- `void reSize ( int newcapacity )`
  resize the array to new capacity. Make sure that elements in old array should be preserved in the new array if possible.

Task-2: *CString ADT*

A class which will provide basic functionalities related to strings.

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

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

`CString ();`	Initializes data and size to 0.
`CString (char c);`	Initializes data with char c
`CString(const char *);`	Initializes the data with received string by allocating memory on heap.
`~CString ();`	You know what to do.
`void input();` data	Takes input from console in calling object.
`char & at(int index);`	*Index:* Receives the index for string. *Return Value:* reference of array location represented by index

`bool isEmpty( );`	Tells whether string is empty or not *Return Value*: return true if string empty otherwise true.
`int getLenght();` pass count of characters	Returns length of the string
`void display();`	Prints the string on console
`int find(CString * subStr, int start=0 ) ;`	Find the substring in the calling CString object. By default, search starts from 0 index. Return the count of occurrences found in calling object.
`void insert(int index, CString * subStr);`	Insert the substring at given index in calling object.
`void remove(int index, int count=1);`	Remove the characters (how many? Given in count) starting from index
`int replace(CString * old, CString * 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 compare(CString & s2 );`	Compare the calling and receive object string. It should behave just like strcmp

};

We shall do many revisions on CString class. That's an initial and amateur version of CString class. You will soo receive an updated version of this class.

no

0 1 2 3 4 nouman

(5)

Nouman

Objective:

- Issue related to Object transition by value.
- Copy Constructor
- Something to learn different (Variable number of arguments) ☺

Task-1:

Add/Update/Modify the following in your CString class.

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

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

`CString ( const CString & );`	
`int find(CString subStr, int start=0 ) ;`	Find the substring in the calling CString object. By default, search starts from 0 index. Return the count of occurrences found in calling object.
`void insert(int index, CString subStr);`	Insert the substring at given index in calling object.
`int replace(CString old, CString newSubStr );`	Find all the occurrences of old substring and replace it with new substring. Return the count of occurrences found in calling object.
`int compare(CString s2 );`	Compare the calling and receive object string. It should behave just like strcmp
`void shrink();`	Resize/shrink the array equal to the length of string pointed by data.

```
};
```

It's not the final version, we shall keep updating this library over the next couple of weeks.☺

Task-2: Something to learn different

I have added a quite different kind of constructor in the Array class, which uses variable number of arguments feature. You are directed to Google (http://www.cprogramming.com/tutorial/c/lesson17.html) this feature and then explore the code given below.

```
class Array
{
    int * data;
    int  capacity;
    int isValidIndex( int index ) const
    {
        return index>=0 && index<capacity;
    }
public:
    ~Array()
    {
        if (data)
            delete [] data;
        data=nullptr;
        capacity=0;
    }
    int & getSet(int index)
    {
        if (isValidIndex(index))
```

```
    //Variable Number of Arguments
    Array(int argCount=0, ...)
    {
        if (argCount<=0)
        {
            capacity=0;
            data=0;
            return;
        }
        capacity = argCount;
        data = new int[capacity];
        va_list vl;
        va_start(vl, argCount);
        for ( int i=0; i<capacity; i++)
            data[i] = va_arg(vl, int);
va_end ( vl );
    }
};
void display(const Array & ref)
```

```
            return data[index];
        exit(0);
    }
    int getCapacity()
    {
        return capacity;
    }
    void reSize ( int newCap )
    {
        if (newCap<=0)
        {
            this->~Array();
            return;
        }
        int * ptr = new int[newCap];
        memcpy(ptr, data,
(newCap<capacity?newCap:capacity)*sizeof(int));

        this->~Array();

        capacity = newCap;
        data = ptr;
    }
    Array ( const Array & ref)
    {
        if (ref.data==0)
        {
            data=0;
            capacity=0;
            return;
        }
        capacity=ref.capacity;
        data = new int[capacity];
        memcpy(data, ref.data,
capacity*sizeof(int));
    }
{
    for ( int i=0; i<ref.getCapacity();
i++)
        cout<<ref.getSet(i)<<" ";
    cout<<endl;
}

int main()
{
    Array a(3,1,2,4);//first argument is
array size : rest are values
    a.getSet(1)=90;
    display(a);
    Array b;
    Array c(10);
    const Array d(5,10,20,30,40,50);
    //d.getSet(1)=110;//not allowed as d is
constant
    display(d);
    return 0;
}
```

Nouman

Objective:

- It will help you understand the idiom "Use const wherever possible" and "Principle of least privilege".

Task-1:

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

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

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

`CString ();`	Initializes data and size to 0.
`CString (const char c);`	Initializes data with char c
`CString(const char *);`	Initializes the data with received string by allocating memory on heap.
`CString ( const CString & );`	
`~CString ();`	You know what to do.
`void input();`	Takes input from console in calling object.
`void shrink();`	Resize/shrink the array equal to the length of string pointed by data.
`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
`int find(const CString & subStr, int start=0 ) const;`	Find the first occurrence of substring in the calling CString object. By default, search starts from 0 index. If found then return the starting position of subStr found otherwise return -1.
`void insert(int index, const CString & subStr);`	Insert the substring at given index in calling object.
`void remove(int index, int count=1);`	Remove the characters (how many? Given in count) starting from index
`int replace(const CString & old, const CString & 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.

2 of 2

`int compare(const CString & s2 ) const;`	Compare the calling and receive object string and behave just like strcmp
`CString left( int count ) ;`	*Count:* The number of characters to extract from calling object from left side *Return Value:* A CString object that contains a copy of the specified range of characters
`CString right( int count ) ;`	
`int toInteger() const;`	
`float toFloat() const;`	
`CString concat(const CString & s2 ) const ;`	It returns the concatenated result of received and calling object without changing calling object.
`void concatEqual(const CString & s2 );`	It concatenates the received object string with calling object.
`CString tokenzie(const CString & delim ) ;`	Returns a CString object which contains the substring by extracting it from the calling object CString depending upon the delimiter characters passed. See the following Sample Run to further understand the functionality:

```
int main()
{
    CString str(" This, --a sample string. nothing");
    CString token;
    cout<<"String = ";str.display();cout<<"\n";

    while(!str==false)
    {
        token = str.tokenize(".-,");
        cout<<"Token = ";token.display();cout<<"\n";
    }
    cout<<endl;
    return 0;
}
```

Console Output

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

Note:
The code given in main function will produce runtime error because of shallow copying of CString objects. i.e. token = str.tokenize (",.-");
The learning that we have done so far, we are able to copy such objects only at the time of declaration. We haven't been able to find its solution yet and still avoid such syntax with our objects. So, you got to think some other way/syntax of assigning CString objects until we rectify this issue on our CString class.

};

concatenate → کنکٹینیٹ

Nouman

Objective:

- Focus on the purpose/use of class level information (data/operations).

Task-1: discussed in class/lecture

```
class CMath
{
public:
    static float calcPower ( int base, int exponent );
    static int calcGCD ( int numerator, int denominator );
    static CString toCString ( long long int num );
    static long long int toInteger ( CString );
    //you may add other mathematical functions in the same way
};
```

Task-2:

There is still a possibility of creating more than one objects of the following class (discussed in lecture as well). Hunt that flaw but if you get exhausted then do discuss with me.

```
class Singleton
{
private:
    Singleton ( )
    { };
    ~Singleton ( )
    { }
    static Singleton * ptr;
public:
    static Singleton * createObject ( )
    {
        if ( ! ptr )
            ptr = new Singleton;
        return ptr;
    }
    static void freeObject ( )
    {
        if ( ptr )
        {
            delete ptr;
            ptr = nullptr;
        }
    }
};
Singleton * Singleton::ptr = nullptr;
```

Task-3:

Design a class called 'Date'. The class should store a date in three integers: month, day, and year. There should be member functions to print the date in the following forms:

Format-1: 12/25/2012
Format-2: December 25, 2012
Format-3: 25 December 2012

Demonstrate the class by writing a complete program implementing it.

Your setter functions should make sure following:

- A valid year is between 1900 and 2100
- A valid month is between 1-12
- A valid day can be between 1-31 (according to the respective month)

Make following daysInMonth array as class's private data member to easily know the number of days each month.

```
static const int daysInMonth[ 13 ] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

class Date
{
private:
    int day;
    int month;
    int year;
    static const int daysInMonth[ 13 ];
    bool isLeapYear () const;
public:
    Date ( );
    Date ( int, int, int );
    void setDate ( int, int, int );
    void setDay ( int );
    void setMonth ( int );
    void setYear ( int );
    int getDay ( ) const;
    int getMonth ( ) const;
    int getYear ( ) const;
    void printFormat1 ( ) const;
    void printFormat2 ( ) const;
    void printFormat3 ( ) const;
    void incDay ( int = 1 );
    void incMonth ( int = 1 );
    void incYear ( int = 1 );
    CString getDateInFormat1 ( ) const;
      //if *this object contains day=25, month=12 and year=2012 then it returns a CSting
      //object containing "12/25/2012"
    CString getDateInFormat2 ( ) const;
    CString getDateInFormat3 (e) const;
};
```

Note:

- You are not allowed to use C++ string functions but you are free to use CString function wherever needed.
- A leap year is a year which is either divisible by 4 yet not by 100, or it is divisible by 400.

Noumon

Objective:

- Understanding and implementing weak/strong Aggregation and dealing dynamic memory allocation for objects.
- Reusability through composition/aggregation.

~~compose~~ I ~~parent class decides~~ nt
parent class constructor decides which constructor of composed class should execute

Practice-8 Practice 2

Task 1: ***Set using Array***

You must be quite familiar with the class 'Array' implemented below. The other class i.e. 'Set' is also not new to you. Although the purpose of class 'Set' is same like the previous version but its data members are changed and 'Set' is implemented using class 'Array' object. I have defined a few functions in Set class, you are required to implement rest of the function given in Set class.
Remember: You must not change anything private/public in class Array. Wisely decide about the fate of destructor and copy constructor of class 'Set'

```
class Array
{  //Discussed in class  };

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;
};
```

for composed object of Array we initialize it in member initialization List

It call constructor

we do not need the copy constructor, because there is no heap, but for array member we have already it copy constructor

so we can write as
set e(a);

A{1,2,3,4,5}

B{1,2,4,6,10}

A - B = {3,5}

{1,2,3,4} {3,4,6,8}

A ∪ B = {1,2,3,4,6,8}

A ∩ B = {3,4}

Symetric = {1,2,6,8}

non common elements

Task 2: ***→ PF >= 65 : Bravo to others if they do it too.***

Complete the following class which will convert the floating-point value into 32b as per IEEE754. You got to use the standard/IEEE754 algorithm/method of conversion as discussed in lab/lecture. No short cuts allowed.

```
class IEEE754
{
    BitArray floatingBits;
    float value = 1;
public:
    IEEE754();
    void updateFloatingPointValue(float);
    void
updateIEEE754FloatingBits(BitArray);
    float getValue();
    BitArray getIEEE754FloatingBits();
};
```

```
int main()
{
    IEEE754 fp;
    fp.getValue();//display 1;
    fp.getIEEE754FloatingBits().dump();
//display 00111111 10000000 00000000 00000000
    fp.updateFloatingPointValue(12.375);
    fp.getIEEE754FloatingBits().dump();
//display 01000001 01000110 00000000 00000000
    return 0;
}
```

Nouman

Objective:

- Understanding and implementing weak/strong Aggregation and dealing dynamic memory allocation for objects.
- Reusability through composition/aggregation.
- Extension/Modification in ADT without changing the existing interface or code.
- Focusing on Array of objects.

1 2 3
4 5 6
7 8 9

Task-1: ***Manipulating Square Matrices***

As you know, that we implement Matrix class few weeks before in a quiz/lab. Suppose that the user wants to manipulate square matrices only and for this purpose he want to mention matrix order in single variable like N-Order matrix instead of separately giving rows and columns. For this purpose, what we can do? Well, considering the knowledge of OOP that we have explored so far, we can write a wrapper class (lets name it "SMatrix") and reuse the methods defined in Matrix class.
While implementing 'SMatrix', the idea is to give all the features of matrices without changing a bit in 'Matrix' class but reusing the functions defined in Matrix class instead of reinventing the wheel.
The basics has also been discussed in class/lecture, so I hope that you will be able to accomplish this solution.

```
class Matrix
{
    int rows,cols;
    double ** data;
public:
    Matrix();
    Matrix(int,int);
    Matrix(const Matrix &);
    ~Matrix();
    double & at( int, int);
    const double & at( int, int) const;
    int getRows() const;
    int getColums() const;
    void display() const;
    Matrix Transpose() const;
    Matrix add(const Matrix &) const;
    Matrix multiply(const Matrix &) const;
    Matrix reSize(int);
    //many other functions
};
```

Array * data
int row, col
1) Matrix
2) Matrix(int r, int c)
3) operator []
4) Array * operator () (int rNo)
5) ~Matrix();
6) operator =
using array class without copy constructor and assignment.

Task-2: ***Scheduler***

We need to design an application like the one you normally use in your mobiles about keeping record of tasks to be done for any particular date and time.
For this purpose, we need following classes:

data [ ] [ ]

Class Task: will be responsible for recording the message/task to be done in particular date and time. So, for this purpose the following members are needed for class Task:

```
class Task
{
    Date taskDate;
    Time taskTime;
    CString taskMsg;
};
```

You should be quite familiar with the class 'Date', class 'Time', and class 'CString'. So, we can say that a Task is composed of Date, Time, and CString objects (strong-aggregation/composition).

Class Scheduler: But as you know that a scheduler will record/save many tasks (more than one Task objects) in it. So we need another class, which will be responsible for keeping record of all tasks recorded/saved by user. Class 'Scheduler' will be responsible for this purpose, which is as follows:

```
class Scheduler
{
    Task * taskList;
    int noOfTasks;
    int capacity;
};
```

- taskList: will point to an array of objects of type Task.
- noOfTasks: keep record of the number of tasks saved by user in Scheduler object.
- capacity: size of the array pointed by 'taskList'

Class SchedulerApp: all the interface related things will come in it.

Class CString: Same old CString ☺.

Class Date and Time: Nothing new for you guys, but I have also provided few functions implementation of Date class.

You should decide yourself about placing copy constructor, and destructor in any class given below.

Also implement the function given in class 'Scheduler' and class 'SchedulerApp'

```
class CString
{
// same as given in previous practice file
};
class Time
{
// same as given in previous practice file
};
class Date
{
    static const int daysInMonth[ 13 ];
    int day;
    int month;
    int year;
    bool isLeapYear(int y) const;
    bool isValidDate(int d, int m, int y) const;

public:
    Date():day(14), month(10), year(2019)
    { }
    Date( int d, int m, int y):day(14), month(10), year(2019)
    {
        setDate(d,m,y);
    }
    void setDate( int d, int m, int y);
    void setDay(int d);
    void setMonth(int m);
    void setYear(int y);
    int getDay() const;
    int getMonth() const;
    int getYear() const;
    void printFormat1()const
    {
        cout.fill('0');
        cout<<setw(2)<<day<<"/"<<setw(2)<<month<<"/"<<setw(4)<<year;
    }
```

Nouman

```
    void printFormat2()const;
    void printFormat3()const;
    void incDay(int=1);
    void incMonth(int=1);
    void incYear(int=1);
    CString getDateInFormat1() const
    {
        CString date;
        date.reSize(11);//OR CString date("00/00/0000");
        date.concatEqual(to_string(day).c_str());
        //you can't use  to_string function ☺
        date.concatEqual("/");
        date.concatEqual(to_string(month).c_str());
        date.concatEqual("/");
        date.concatEqual(to_string(year).c_str());
        return date;
    }
    CString getDateInFormat2()const;
    CString getDateInFormat3()const;
};

const int Date::daysInMonth[ 13 ] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```
class Task
{
    Date taskDate;
    Time taskTime;
    CString taskMsg;
public:
    Task()
    {}
    Task(const Date & d, const Time & t, const CString & m):
taskDate(d),taskTime(t),taskMsg(m)
    {}
    void setTask(const Date & d, const Time & t, const CString & m)
    {
        taskDate = d;//no issue with shallow copy
        taskTime = t;//no issue with shallow copy
        taskMsg = m; // shallow copy will create issues
    }
    void updateDate(const Date & nd)
    {
        taskDate = nd;
    }
    void updateTime(const Time & nt)
    {
        taskTime = nt;
    }
    void updateMessage(const CString & m)
    {
        taskMsg = m;
    }
    Date getDate()
    { return taskDate; }
    Time getTime()
    { return taskTime; }
    CString getMessage()
    { return taskMsg; }
};
class Schedular
{
    Task * taskList;
    int noOfTasks;
```

```
    int capacity;
public:
    Schedular()
    {
        capacity=5;
        noOfTasks=0;
        taskList = new Task[capacity];
    }
    void addTask(const Task & t);
    void displayTask(const Date & d=Date(0,0,0))
    {
        //show todays tasks by default or according to the date received
    }
    void displayTodaysTasks();
    void reSize( int ); //resizes the array whenever it gets full.
};
class SchedulerApp
{
public:
    static void startApp()
    {
        //all the interface related things will come here
    }
};
```

Note:

- I have explicitly left couple of bugs in few places, which you got to identify/rectify yourself. And this statement may itself be a cattywampus ☺.
- To save space I have given definitions within class body. You got to follow all the coding conventions that we have discussed so far.

include

1) The constructor of base class decide which constructor of composed class is to be execute.

Objective:

- Introducing flexibility/usability in the CString class by adding defining different operators for it.

Task-1: ***Final Version of CString*** ☺

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

copy constructor

copy assignment operator

`CString ();`	Initializes data and size to 0.
`CString (const char c);`	Initializes data with char c
`CString(const char *);`	Initializes the data with received string by allocating memory on heap.
`CString ( const CString & );`	execute at declaration `CString b(a);`
`CString & operator = ( const CString & ref );`	`CString a("Nouman"); a = b;`
`~CString ();`	
`int getSize() const;`	Getter of the size data member
`void input();`	Takes input from console in calling 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 *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
`int find(const CString & subStr, int start=0 ) const;`	Find the first occurrence of substring in the calling CString object. By default, search starts from 0 index. If found then return the starting position of subStr found otherwise return -1.
`void insert(int index, const CString & subStr);`	Insert the substring at given index in calling object.
`void remove(int index, int count=1);`	Remove the characters (how many? Given in count) starting from index
`int replace(const CString & old, const CString & 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 CString & s2 ) const;`	Compare the calling and receive object string and behave just like strcmp
`CString left( int count ) ;`	*Count:* The number of characters to extract from calling object from left side *Return Value:* A CString object that contains a copy of the specified range of characters
`CString right( int count ) ;`	
`explicit operator long long int();`	Converts the integral value stored in calling object to long long int and returns
`explicit operator double();`	Converts the floating point value stored in calling object to double and returns
`CString operator + (const CString & s2 ) const;`	It returns the concatenated result of received and calling object without changing calling object.
`void operator += (const CString & s2 );`	It concatenates the received object string with calling object.
`CString tokenize(const CString & delim );`	Returns a CString object which contains the substring by extracting it from the calling object CString depending upon the delimiter characters passed. See the following Sample Run to further understand the functionality:
`CString operator () (const CString & delim );`	Different syntax to use tokenize.

```
int main()
{
    CString str(" This, --a sample string. nothing");
    CString token;
    cout<<"String = ";str.display();cout<<"\n";

    while(!str==false)
    {
        token = str.tokenize(".-,");
        cout<<"Token = ";token.display();cout<<"\n";
    }
    cout<<endl;
    return 0;
}
```

Console Output

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

Note:
No issue in sample run anymore.

};

Global functions part of CString.h	
`istream & operator >> (istream &, CString &);`	Console Input string
`ostream & operator << (ostream &, const CString &);`	Console Output string

CMP-245 Object Oriented Programming Lab
BS Fall 2018
Lab 09 X Practice file

Issue Date: 22-Nov-2019
Marks: 64.5

Objective:

- This lab will help in gripping the concept of object manipulation involved while object transition in case of aggregation/composition and array of objects.

Challenge: (4, 8, 9, 9, 10)

Shery is your best friend. Shery's father owns a snack bar but he is facing loss as he is maintaining all the orders on paper manually and its quite difficult and time consuming to maintain all the receipts/worksheets. Shery convinced his father to implement a snack bar management application to efficiently maintain all the orders. His father handed over this task to Shery (As he is also an IT student). Shery has analyzed the whole system and identified all classes, member functions, their data members and relationship between them, but his coding skills are weak. Luckily, Shery has many friends in CS/SE-F18. All the students of CS/SE-F18 are known for their exceptional programming skills. Shery approached you and request you to write code according to the description given to each class and convert his classes design into implementation model with basic console I/O interface.

The description of the system is as follows:
If we observe the snack bar we can easily notice following:

- Manager prepares a Menu containing food dishes/Menu Items each day.
- Customers give orders and each order has selected Menu Items in it.
- An order list is maintained to hold the records of orders and to calculate the revenues at the end of the day.

From above description, we have extracted the following nouns

Snack Bar
Order list
Menu
Menu Items
Orders

Now these nouns will be our expected classes and the relevant attributes will be the data members of the corresponding classes.
We can also represent the classes in UML notation as:

You can add the getter/setter and constructor/destructor functions in the design classes where you feel appropriate and may also add utility functions as and when needed.

According to the discussion: backside of this page contains the class diagram for our SnackBar software. Which you need to develop.

CMP-245 Object Oriented Programming Lab
BS Fall 2018
Lab 09

Issue Date: 22-Nov-2019
Marks: 64.5

Say No to "Jack of all trades, master of none"
Say Yes to "Jack of all trades, master of one"

Objective:

- It will help you understand the benefits, we get through inheritance relationship.
- It will also help in comparing inheritance and composition.

Task-1:

Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class *BasePlusCommissionEmployee* of the hierarchy discussed in 12.4 section of text-book-A to use composition rather than inheritance. After you do this, assess the relative merits of the two approaches for designing classes *CommissionEmployee* and *BasePlusCommissionEmployee*, as well as for object-oriented programs in general. Which approach is more natural? Why?

Task-2:

(Account *Inheritance Hierarchy)* Create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).

Create an inheritance hierarchy containing base class Account and derived classes SavingsAccount and CheckingAccount that inherit from class Account. Base class Account should include one data member of type double to represent the account balance. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it is greater than or equal to 0.0. If not, the balance should be set to 0.0 and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function credit should add an amount to the current balance. Member function debit should withdraw money from the Account and ensure that the debit amount does not exceed the Account's balance. If it does, the balance should be left unchanged and the function should print the message "Debit amount exceeded account balance." Member function getBalance should return the current balance.

Derived class SavingsAccount should inherit the functionality of an Account, but also include a data member of type double indicating the interest rate (percentage) assigned to the Account. SavingsAccount's constructor should receive the initial balance, as well as an initial value for the SavingsAccount's interest rate. SavingsAccount should provide a public member function calculateInterest that returns a double indicating the amount of interest earned by an account. Member function calculateInterest should determine this amount by multiplying the interest rate by the account balance. [*Note:* SavingsAccount should inherit member functions credit and debit as is without redefining them.]

Derived class CheckingAccount should inherit from base class Account and include an additional data member of type double that represents the fee charged per transaction. CheckingAccount's constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class CheckingAccount should redefine member functions credit and debit so that they subtract the fee from the account balance whenever either transaction is performed successfully. CheckingAccount's versions of these functions should invoke the base-class Account version to perform the updates to an account balance. CheckingAccount's debit function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [*Hint:* Define Account's debit function so that it returns a bool indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the SavingsAccount object by first invoking its calculateInterest function, then passing the returned interest amount to the object's credit function.

Task-3:
Case Study: Chapter 15: Inheritance

The Automobile, Car, Truck, and SUV classes

Suppose we are developing a program that a car dealership can use to manage its inven- tory of used cars. The dealership s inventory includes three types of automobiles: cars, pickup trucks, and sport-utility vehicles (SUVs). Regardless of the type, the dealership keeps the following data about each automobile:

- Make
- Year model
- Mileage
- Price

Each type of vehicle that is kept in inventory has these general characteristics, plus its own specialized characteristics. For cars, the dealership keeps the following additional data:

- Number of doors (2 or 4)

For pickup trucks, the dealership keeps the following additional data:

- Drive type (two-wheel drive or four-wheel drive)

And, for SUVs, the dealership keeps the following additional data:

- Passenger capacity

In designing this program, one approach would be to write the following three classes:

- A Car class with attributes for the make, year model, mileage, price, and number of doors.
- A Truck class with attributes for the make, year model, mileage, price, and drive type.
- An SUV class with attributes for the make, year model, mileage, price, and passen- ger capacity.

This would be an inefficient approach, however, because all three classes have a large number of common data attributes. As a result, the classes would contain a lot of duplicated code. In addition, if we discover later that we need to add more common attributes, we would have to modify all three classes.

A better approach would be to write an Automobile base class to hold all the general data about an automobile, and then write derived classes for each specific type of automobile. The following code shows the Automobile class.

Contents of Automobile.h

```
#ifndef AUTOMOBILE_H
#define AUTOMOBILE_H
class Automobile
{
private:
    CString make;
    int model;
    int mileage;
    double price;
public:
    Automobile();
    Automobile(CString autoMake, int autoModel, int autoMileage, double autoPrice);
```

```
    CString getMake() const;
    int getModel() const;
    int getMileage() const;
    double getPrice() const;
};
#endif
```

Contents of Automobile.cpp

```
#include "Automobile.h"
Automobile::Automobile():make("")
{
    model = 0;
    mileage = 0;
    price = 0.0;
}
Automobile::Automobile(CString autoMake, int autoModel, int autoMileage, double
autoPrice):make(autoMake)
{
    model = autoModel;
    mileage = autoMileage;
    price = autoPrice;
}
CString Automobile::getMake() const
{
    return make;
}
int Automobile::getModel() const
{
    return model;
}
int Automobile::getMileage() const
{
    return mileage;
}
double Automobile::getPrice() const
{
    return price;
}
```

The Automobile class is a complete class that we can create objects from. If we wish, we can write a program that creates instances of the Automobile class. However, the Automobile class holds only general data about an automobile. It does not hold any of the specific pieces of data that the dealership wants to keep about cars, pickup trucks, and SUVs. To hold data about those specific types of automobiles we will write derived classes that inherit from the Automobile class. The following shows the code for the Car class.

Contents of Car.h

```
#ifndef CAR_H
#define CAR_H
#include "Automobile.h"

class Car: public Automobile
{
private:
    int doors;
public:
    Car();
    Car(CString carMake, int carModel, int carMileage, double carPrice, int carDoors);
    int getDoors();
};
#endif
```

Contents of Car.cpp

```
#include "Car.h"
Car::Car() : Automobile()
{
    doors=0;
}
Car::Car(CString carMake, int carModel, int carMileage, double carPrice, int carDoors):
 Automobile(carMake, carModel, carMileage, carPrice)
{
    doors = carDoors;
}
int Car::getDoors()
{
    return doors;
}
```

Notice that the 'Car' default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The Car class also has an overloaded constructor, which accepts arguments for the car's make, model, mileage, price, and number of doors. This parameterized constructor also calls the Automobile class's constructor, passing the make, model, mileage, and price as arguments.

Now lets look at the Truck class, which also inherits from the Automobile class.

Contents of Truck.h

```
#ifndef TRUCK_H
#define TRUCK_H
#include "Automobile.h"
class Truck : public Automobile
{
private:
    CString driveType;
public:
    Truck();
    Truck(CString truckMake, int truckModel, int truckMileage, double truckPrice, CString
    truckDriveType);
    CString getDriveType();
};
#endif
```

Contents of Truck.cpp

```
#include "Truck.h"
Truck::Truck() : driverType(""),Automobile()
{
}
Truck::Truck(CString truckMake, int truckModel, int truckMileage, double truckPrice,
CString truckDriveType) : Automobile(truckMake, truckModel, truckMileage,
truckPrice),driveType(truckDriveType)
{
}
CString Truck::getDriveType()
{
    return driveType;
}
```

The Truck class defines a driveType attribute to hold a string describing the truck's drive type. The

class has a default constructor that sets the driveType attribute to an empty string. Notice that the Truck's default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The Truck class also has an overloaded constructor, that accepts arguments for the truck's make, model, mileage, price, and drive type, which calls the Automobile class's constructor, passing the make, model, mileage, and price as arguments.

Now lets look at the SUV class, which also inherits from the Automobile class.

Contents of SUV.h

```
#ifndef SUV_H
#define SUV_H
#include "Automobile.h"
class SUV : public Automobile
{
private:
    int passengers;
public:
    SUV();
    SUV(string SUVMake, int SUVModel, int SUVMileage,double SUVPrice, int SUVPassengers);
    int getPassengers();
};
#endif
```

Contents of SUV.cpp

```
#include "SUV.h"
SUV::SUV() : Automobile()
{
    passengers = 0;
}
SUV::SUV(string SUVMake, int SUVModel, int SUVMileage,double SUVPrice, int SUVPassengers)
: Automobile(SUVMake, SUVModel, SUVMileage, SUVPrice)
{
    passengers = SUVPassengers;
}
int SUV::getPassengers()
{
    return passengers;
}
```

for inheritance we call constructor of base class by using its class_name.

The SUV class defines a passenger's attribute to hold the number of passengers that the vehicle can accommodate. The class has a default constructor that sets the passengers attribute to 0. Notice hat the SUV's default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The SUV class also has an overloaded constructor that accepts arguments for the SUV s make, model, mileage, price, and number of passengers.

```
#include <iostream>
#include "Car.h"
#include "Truck.h"
#include "SUV.h"
using namespace std;
int main()
{
   // Create a Car object for a used 2007 BMW with
```

```
    // 50,000 miles, priced at $15,000, with 4 doors.

    Car car("BMW", 2007, 50000, 15000.0, 4);

    // Create a Truck object for a used 2006 Toyota
    // pickup with 40,000 miles, priced at $12,000,
    // with 4-wheel drive.

    Truck truck("Toyota", 2006, 40000, 12000.0, "4WD");

    // Create an SUV object for a used 2005 Volvo
    // with 30,000 miles, priced at $18,000, with
    // 5 passenger capacity.

    SUV suv("Volvo", 2005, 30000, 18000.0, 5);
    // Display the automobiles we have in inventory.
    cout << "We have the following car in inventory:\n"
         << car.getModel() << " " << car.getMake()
         << " with " << car.getDoors() << " doors and "
         << car.getMileage() << " miles.\nPrice: $"
         << car.getPrice() << endl << endl;
    cout << "We have the following truck in inventory:\n"
        << truck.getModel() << " " << truck.getMake()
        << " with " << truck.getDriveType()
        << " drive type and " << truck.getMileage()
        << " miles.\nPrice: $" << truck.getPrice()
        << endl << endl;

    cout << "We have the following SUV in inventory:\n"
        << suv.getModel() << " " << suv.getMake()
        << " with " << suv.getMileage() << " miles and "
        << suv.getPassengers() << " passenger capacity.\n"
        << "Price: $" << suv.getPrice() << endl;

    return 0;
}
```

Program Output

```
We have the following car in inventory:
2007 BMW with 4 doors and 50000 miles.
Price: $15000.00

We have the following truck in inventory:
2006 Toyota with 4WD drive type and 40000 miles.
Price: $12000.00

We have the following SUV in inventory:
2005 Volvo with 30000 miles and 5 passenger capacity.
Price: $18000.00
```

If we inherit a class and also makes object of different classes. Then for derive class object. First the constructor of base class execute and then the constructor of composed object execute in the order they are declare in the class (whether they write in any order in

Objective:

- To look into the technical and theoretical aspects of the Polymorphism.

Task 1:

Discussed in class also.

A student is assigned to make a program. The program is supposed to play the voice of animal selected. For this purpose, student comes up with two alternate designs. Which design do you think is better and WHY (give some logical reason(s))?

```
//Solution-A
class Animal
{
public:
    Animal();
    virtual void speak()
    { }
};
class Cat : public Animal
{
public:
    void speak()
    { cout<<"Meeeaaauuun"; }
};
class Dog : public Animal
{
public:
    void speak()
    { cout<<"Baao Baaaaaaoooo"; }
};
class Rat : public Animal
{
public:
    void speak()
    { cout<<"chi een eeeen"; }
};
```

```
//Solution-B
class Animal
{
public:
    enum EnumType { Cat, Dog, Rat };
    Animal( EnumType type );
    void speak();
private:
    EnumType animalType;
};
void Animal::speak()
{
    switch ( animalType )
    {
        case Cat:
           cout<<"Meeeaaauuun";
           break;
        case Dog:
           cout<<"Baao Baaaaaaoooo";
           break;
        case Rat:
           cout<<"chi een eeeen";
           break;
    }
};
```

Task 2: **just zip through it today**

Ship, CruiseShip, CargoShip and BattleShip Classes

1. Ship class

   Design a Ship class that has the following members:

   - A member variable for the name of the ship (a string)
   - A member variable for the year that the ship was built (a string)
   - A constructor and appropriate accessors and mutators
   - A virtual print function that displays the ship's name and the year it was built.

2. CruiseShip class

   Design a CruiseShip class that is derived from the Ship class. The CruiseShip class should have the following members:

   - A member variable for the maximum number of passengers (an int)
   - A constructor and appropriate accessors and mutators
   - A print function that overrides the print function in the base class. The CruiseShip class's print function should display the ship's name, year of built and the maximum number of passengers.

3. CargoShip class

   Design a CargoShip class that is derived from the Ship class. The CargoShip class should have the following

members:

- A member variable for the cargo capacity in tonnage (an int). total capacity of a ship (in tons)
- A constructor and appropriate accessors and mutators.
- A print function that overrides the print function in the base class. The CargoShip class's print function should display only the ship's name and the ship's cargo capacity.

4. BattleShip class جنگی جہاز

Design a BattleShip class that is derived from the Ship class. The BattleShip class should have the following members:

- A member variable for the total number of missiles (an int).
- A constructor and appropriate accessors and mutators.
- A print function that overrides the print function in the base class. The BattleShip class's print function should display only the ship's name and the missiles' capacity.

5. A Driver Program

Demonstrate the classes in a program that has an array of Ship pointers. The array elements should be initialized with the addresses of dynamically allocated Ship, CruiseShip, CargoShip and BattleShip objects. The program should then step through the array, calling each object's print function.

Task 3: do it before coming lab

A bank holds different types of accounts for its customers: **deposit accounts**, **loan accounts** and **mortgage accounts**.

- Customers could be **individuals** or **companies**.
- All accounts have **customer, balance** and **interest rate** (monthly based).
- **Deposit** accounts are allowed to deposit and withdraw money. **Loan** and **mortgage** accounts can only deposit money.
- All accounts can calculate their interest for a given period (in months) using the formula **A = money * (1 + interest rate * months)**
- **Loan** accounts have no interest for the first **3 months** if held by **individuals** and for the first **2 months** if held by a **company**.
- **Deposit** accounts have no interest if their balance is positive and less than 1000.
- **Mortgage** accounts have 1/2 interest for the first 12 months for **companies** and no interest for the first 6 months for **individuals**.

Write a program to model the bank system with classes. You should identify the classes and their relationship and implement the calculation of the interest functionality through overridden methods. Write a program to demonstrate that your classes work correctly.

Task 1:

Solution A: In this version, if I want to add one more animal, I just make class for it and inherit Animal class. I do not even touch the existing code

Solution B: But in this version if I want to add one more animal I have to change the existing code which is very diffi

Task 4: POLYMORPHISM

Thoroughly understand the contents till page 5 of this document before next lecture.

Polymorphism appears in C++ in two different forms

1. **Static/Compile Time/adhoc Polymorphism**
   1.1. Function/Operator Overloading
   1.2. Parameterized [Template] Polymorphism
2. **Dynamic/Runtime Polymorphism**

1.1 Function/Operator Overloading

In order to overload a function, a different list of parameters is used for each overloaded version. For example, a set of valid overloaded versions of a function named f() might look similar to the following:

```
void f(char c, int i);
void f();
void f(int i);
```

So, a function named as 'f' exists in more than one shapes (different parameter list). And which shape of 'f' to call? The decision is made on compile time.

Similarly, if we observe the following + operations:

```
int a; float b; double d; char t;
a+b;        b+a;        a+d;        t+a;
```

The same '+' operator is performing "additions operation" on different types of data. And this concept can be extended to user defined classes.
So which overloaded operator function to call? The decision is made at compile time.

1.2 Parameterized [Template] Polymorphism

A generic (template) function/class can be defined out of which actual function /class can be generated. The generation of actual function/class is dependent on the (in case of function: on function call parameter, in case of class: on object creation)
So, again the decision is made at compile time,

```
template<class T>
void f(T a)
{}
template<typename T>
class Test
{
      T data;
public:
      Test()
      { data=0; }
};
.....
f(12);              //compiler generates void f(int) considering the call parameter.
Test<float> fun; //Compiler generates class "Test<float>" replacing type T by float
                 //Wherever T is used in class template
Test<int> you; //this time: class Test<int> is generated.
```

2. Dynamic/Runtime Polymorphism

by method overriding.

Object interaction take place by sending messages to them.
When a message (member function call) is sent to an object, it will react by executing an appropriate method (function body).
Let's see an example:

```
class base
{
public:
   void f() {}
};
```

```
class derive: public base
{
public:
   void f() {}
};
```

```
void main()
{
   derived d;
   base *pb = & d;
   pb->f();
}
```

This just hide base class function f()

In the above example "**static type**" of "pb" is "base*" and "**dynamic type**" of "pb" is the type of pb's referent (derive*).

If we put the duty of "appropriate method invocation (binding)" to compiler it will do silly things. Because compiler has knowledge of only "static type" of an identifier, It will invoke base::f() rather than derive::f() due to the "static type" of "pb" is "base*". We have to find another mechanism that has access to "dynamic type" of an identifier.
Let's update the above example:

Updated example:

```
class base
{
public:
    virtual void f() {}
};
```

```
class derive: public base
{
public:
    void f() {}
};
```

```
void main()
{
    derived d;
    base *pb = & d;
    pb->f();
}
```

Now pb->f() binds to derive::f();
Miracle: because of overriding

Overriding:

1. The method in base class is declared as virtual"
2. The derive class method is exactly **"same type function"** as the base class method. That means the derive class method has the same return type, the same formal arguments, and it is a const member function if and only if the base member function is also const.

Overriding is precisely what is necessary for runtime polymorphism/binding
So, in short runtime **polymorphism** means:

Polymorphism:

"Objects of different classes related by inheritance respond differently to the same message."

Binding: means; which function body to invoke on a function call. If the decision is taken at compile time, it is called **compile/static binding** and if the decision is taken at runtime then it is called **runtime binding.**

```
class base
{
public:
    virtual void f() {}
};
```

```
class derive: public base
{
public:
    void f() {}
};
```

```
class Leaf: public derive
{
public:
    void f() {}
};
```

```
void main()
{
    derived d;
    Leaf f;
    base *pb = &d;
1.....pb->f(); //derive::f()
    pb = &f;
2.....pb->f(); //Leaf::f()
}
```

The same message "pb->f()" at line 1 & 2 but different responses(different method invocation/binding).

The following are synonyms:

- Early=Compile Time=Static binding
- Late=Dynamic=Runtime binding

Overloading vs Hiding vs Overriding:

- **Overloading**
  - Two functions that are overloaded must have the same scope but different signatures. Their formal argument lists are different or one of them is const and the other is not.
- **Hiding**
  - A function declared in derived class with the same as in base class hides the base class method and does not override it.
- **Overriding**
  - Already defined above....

Coding Example to differentiate among hiding, overriding and overloading.

```
class base
{
int a;
public:
    virtual void f(int)
    {}
    virtual void g(int)
    {}
    void h(float)
    {}
};
```

```
class derive: public base
{
int b;
public:
    void f(int) // overrides base::f(int)
    {}
    void f(double) //overloads derive::f(int) doesn't override
    {}             //base::f(int)

    char g(int) //illegal: return must mach for overriding
    {}          //virtual functions.

    char g(char) //Legal : but doesn't override base::g(int).
    {}           //signatures are different

    void h(float) //just hides base::h(float)
    {}

    void h(int) //Legal: since base::h is not virtual. Overloads
    {}          //derived::h(float)
};
```

Signature :

prototype without returntype is a signature

Runtime Polymorphism

This feature is known as covariant return type. This makes the virtual constructor possible.

Some points to ponder:

1. If the return type is a pointer or reference to the base class, then it may be a pointer or reference to the derived class in the derived class signature. Otherwise, the return type must match in any derived class. Not supported by the all the compilers.

This is the only exception that in which return type is different otherwise it 99% same in all cases.

```
class base
{
public:
    virtual base * f() {}
    virtual ~base() {}
};
```

```
class derive : public base
{
public:
    virtual derive * f() {}
    virtual ~derive() {}
};
```

override above

2. Virtual global friend is illegal, but virtual functions in one class may be designated as friend in another class.

→ but it only mark it Inline when we call it by its own object

3. Virtual function may be inline and have default arguments.

4. Changing the access level of a virtual function in a derived class is a poor idea. For example:

```
class base
{
public:
    virtual void fun() {}
};
```

```
class derive: public base
{
private:
    virtual void fun() {}
};
```

```
void main()
{
    base *pb=new derive;
    pb->fun(); → it execute
}    derive :: fun();
```

Note: The statement above "pb->fun()" is valid, because access specifiers are compile time check, so at compile time, compiler checks the type of pointer "pb" which is "base", so "pb" can access "base::fun()", But at runtime because of dynamic binding "pb->fun()" resolves to derive::fun(). vtable is independent of access specifiers

5. **Using** declaration help derived class to avoid hiding of base class functions.

Example 1:

```
class base
{
public:
      void print(int , int)      {}
};
class derive: public base
{
public:
      using base::print;
      void print(char)   {}       //bring into local scope
};                                //overloads with derive::print(char)
```

now we can access both function of derive class using derive class object

Example 2:

```
class base
{
public:
      virtual void fun(int , int)  {}
      virtual int fun(float)  {}
      virtual void fun(char *)  {}
      virtual void fun()  {}
};
class derive: public base
{
public:
      using base::fun;  //bring rest of three functions into local scope
```

```
    void fun(int , int)  {}
};
```

NOTE:
Without making use of "using declaration", derive::fun(int,int) hides the other versions of fun().

6. The virtual function mechanism works only with public derivation.

7. The virtual function mechanism doesn't work with private derivation.

```
class base
{
public:
    virtual void fun()
    {}
};
```

```
class derive: private base
{
public:
    void fun() {}
};
```

```
void main()
{
    base *pb = new derive; //illegal: conversion from 'class derive *' to
                           // 'class base *' does not exists
    base *pb=(base*) new derive;  //Legal: now pb->fun() resolves to
    pb->fun();                    //derive::fun()
}
```

now this is not allowed

now this execute derive::fun()

But what about this?

```
class base
{
public:
    virtual void fun()
    {}
};
```

```
class derive: private base
{
public:
    void fun()
    {
        base *pb = new derive;//Legal
        pb->fun();
    }
};
```

we can also do this for protected inheritance.

(we can only do it in function)

کیا func کے اندر اسی class کا object بن سکتا ہے۔ Yes

REASON:
The main function is just like a client which is using base and derive classes. So, no one knows that `class derive` has features of `class base` except the `class derive` itself. So, the idiom 'base class reference can refer to derive class object' works only on `class derive`.

8. The virtual function mechanism doesn't work with protected derivation.

```
class base
{
public:
    virtual void fun() {}
};
```

```
class derive: protected base
{
public:
    void fun() {}
};
```

```
void main()
{
    base *pb = new derive; // illegal: conversion from 'class derive *' to
                           // 'class base *' doesn't exists
    base *pb=(base*) new derive;
    pb->fun();now pb->fun() resolves to derive::fun()
}
```

9. But what about this?

```
class base
{
```

```
class derive: protected base
{
```

```
public:
    virtual void fun()
    {}
};
```

```
public:
    void fun()
    {
       base *pb = new derive;
       //Legal
       pb->fun();
    }
};
```

REASON:
The same reason as in point 8, the only difference is that the class derive inherits features of class base is only known by class derive himself or any down the hierarchy.

10. What happens when we call virtual function inside constructor or destructor:
Example

```
class base
{
public:
    int bdata;
    virtual void f()
    {
       wow();        //if class to f() is from constructor/destructor of
                     //base class then base::wow() Otherwise //derive::wow()
    }
    virtual void wow()     {}
    virtual void hi()      {}
    base()
    {
       hi();         //base::hi()
    }
    virtual ~base()
    {
       hi();         //base::hi()
    }
};
class derive: public base
{
public:
    int ddata;
    virtual void wow()     {}
    virtual void hi()      {}
    derive()
    {
            hi();  //derive::hi()
            f();   //base::f()
    }
    virtual ~derive()
    {
            hi();  //derive::hi()
            f();   //base::f()
    }
};
```

ctor

dtor

ctor

dtor

v table base class

base :: f()
base :: wow
base :: hi
base :: ~base

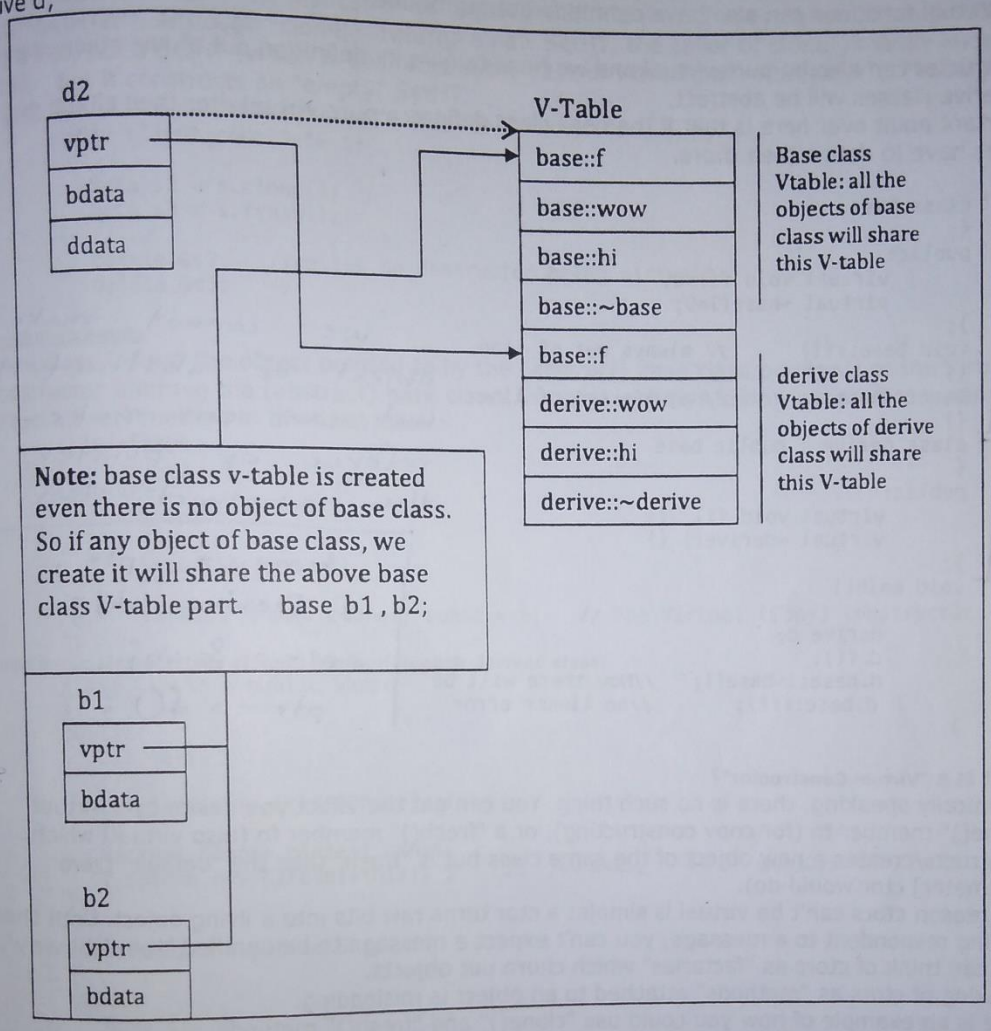
vtable derive class

base :: f
derive :: wow
derive :: hi
derive :: ~derive

NOTE: Strange? let's look at the layout of v-table construction

When the derive object d2 is created the following layout in memory was constructed (i.e; the base class v-table part along with derive class v-table part. So, the interesting thing about this figure is that the d2 vptr is pointing to two different location; Certainly not at a time. Actually, when d2 object was being constructed/initialized, in constructor/destructor of base class the d2 vptr was pointing to base class v-table part and other than this vptr value of d2 object will remain same: pointing to its own derive class v-table part.)

derive d;


d2
vptr
bdata
ddata
V-Table
base::f
base::wow
base::hi
base::~base
base::f
derive::wow
derive::hi
derive::~derive
Base class Vtable: all the objects of base class will share this V-table
derive class Vtable: all the objects of derive class will share this V-table
Note: base class v-table is created even there is no object of base class. So if any object of base class, we create it will share the above base class V-table part. base b1 , b2;
b1
vptr
bdata
b2
vptr
bdata

11. Pure Virtual functions can also have definition but out of line.

12. A destructor can also be pure virtual and we have to give its definition out of line otherwise all the derive classes will be abstract.
Important point over here is that if the base class defines a pure virtual dtor then all the derive classes have to define their dtors.

```
class base
{
public:
        virtual void f()=0;
        virtual ~base()=0;
};
void base::f()      // always out of line
{}
base::~base()       // always out of line
{}
class derive : public base
{
public:
        virtual void f()  {}
        virtual ~derive() {}
};
void main()
{
        derive d;
        d.f();
        d.base::~base();    //Now there will be
        d.base::f();        //no linker error
}
```

we cannot make object of abstract class but we can make refevce or pointer to the abstract class.

```
base * ptr;
Derive obj;
ptr = & obj
ptr -> f();
```

13. What is a "**Virtual Constructor**"?
Technically speaking, there is no such thing. You can get the effect you desire by a virtual "clone()" member fn (for copy constructing), or a "fresh()" member fn (also virtual) which constructs/creates a new object of the same class but is "fresh" (like the "default" [zero parameter] ctor would do).
The reason ctors can't be virtual is simple: a ctor turns raw bits into a living object. Until there's a living respondent to a message, you can't expect a message to be handled "the right way".
You can think of ctors as "factories" which churn out objects.
Thinking of ctors as "methods" attached to an object is misleading.
Here is an example of how you could use "clone()" and "fresh()" methods:

```
class Set
{
public:
        virtual void insert(int);      //Set of "int"
        virtual int  remove();
        //...
        virtual Set& clone() const = 0;  //pure virtual; Set is an ABC
        virtual Set& fresh() const = 0;
        virtual ~Set()
        { }
};
class SetHT: public Set
{
public:
        //...
        Set& clone() const
        {  return *new SetHT(*this);   }
        Set& fresh() const
        {  return *new SetHT();  }
};
```

"new SetHT(...)" returns a "SetHT*", so "*new" returns a SetHT&. A SetHT is-a Set, so the return value is correct. The invocation of "SetHT(*this)" is that of copy construction ("*this" has type "SetHT&"). Although "clone()" returns a new SetHT, the caller of clone() merely knows he has a Set, not a SetHT (which is desirable in the case of wanting a "virtual ctor"). "fresh()" is similar, but it constructs an "empty" SetHT.
Clients can use this as if they were "virtual constructors":

```
void client_code(Set& s)
{
  Set& s2 = s.clone();
  Set& s3 = s.fresh();
  //...
  delete &s2;   //relies on destructor being virtual!!
  delete &s3;   //
}
```

Another Example

If the class "owns" the object pointed to by the (abstract) base class pointer, use the Virtual Constructor Idiom in the (abstract) base class. As usual with this idiom, we declare a pure virtual clone() method in the base class:

```
class Shape
{
 public:
      Shape()
      {}
      Shape(Shape &ref)
      {}
      virtual Shape* clone() const = 0;   // The Virtual (Copy) Constructor
};
```

Then we implement this clone() method in each derived class:

```
class Circle : public Shape
{
public:
   Circle()
   {}
   Circle(Circle &ref):Shape(ref)
   {}
   virtual Shape* clone() const
   { return new Circle(*this); }
};

class Square : public Shape
{
public:
   Square()
   {}
   Square(Square &ref):Shape(ref)
   {}
   virtual Shape* clone() const
   { return new Square(*this); }
};
```

Now suppose that each Fred object "has-a" Shape object. Naturally the Fred object doesn't know whether the Shape is Circle or a Square or ... Fred's copy constructor and assignment operator will invoke Shape's clone() method to copy the object:

```
class Fred
{
public:
   Fred(Shape* p)
   { p_ = p; }
  ~Fred() { delete p_; }
   Fred(const Fred& f) : p_(f.p_->clone())
   {}
   Fred& operator= (const Fred& f)
   {
      if (this != &f)     // Check for self-assignment
```

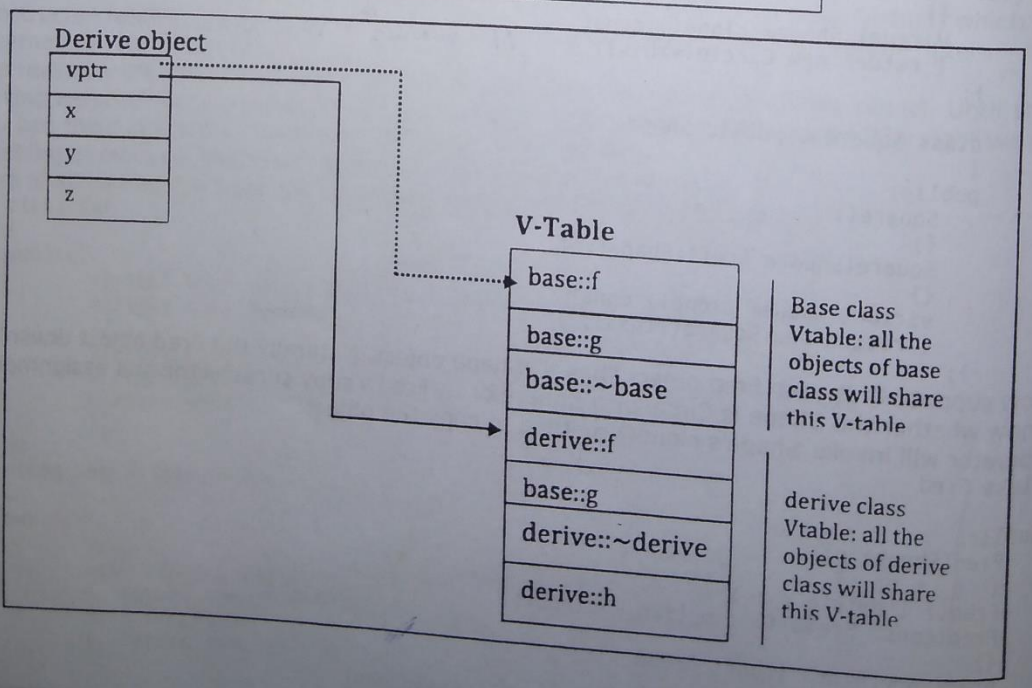
```
        {
                Shape* p2 = f.p_->clone();   // Create the new one FIRST...
                delete p_;                   // ...THEN delete the old one
                p_ = p2;
        }
        return *this;
    }
private:
    Shape* p_;
};
void main()
{
        Fred f(new Circle);     // as this makes in heap
        Fred f1(f);
        Fred f2(new Square);    // no destructor will call
        Fred f3(new Square);

        f1=f3;
        Fred f4=f;
}
```

14.V-Table Structure

Case : a.

```
class base
{
public:
   int x , y;
   virtual void f() {}
   virtual void g() {}
   virtual ~base() {}
};
```

```
class derive: public
base
{
 public:
    int z;
    virtual void f() {}
    virtual void h() {}
    virtual ~derive() {}
};
```

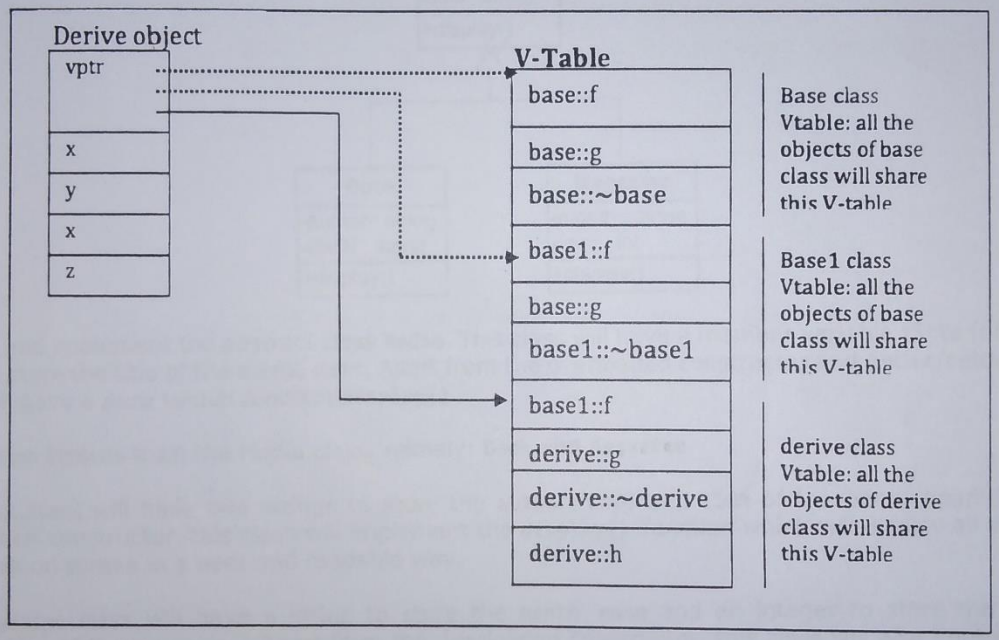


Case : b.

```
class base
{
public:
   int x, y;
   virtual void f() {}
   virtual void g() {}
   virtual ~base() {}
};
```

```
class base1:public base
{
public:
    int x;
    virtual void f() {}
    virtual ~base1() {}
};
```

```
class derive:public base1
{
public:
    int z;
    virtual void g() {}
    virtual void h() {}
    virtual ~derive() {}
};
```



base class destru

virtual destructor 13.9

Objective:

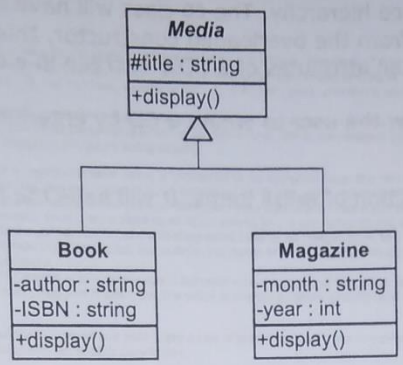
- A coding task to get your hands dirty ☺.

Task-1

[Taken: Robert Lafore book]

Task 1.1

In this task, you are required to implement the following inheritance hierarchy:



Declare and implement the *abstract* class **Media**. This class will have a member variable **title** (of CString type) to store the title of the media item. Apart from the overloaded constructor and getter/setter, Media class will have a *pure virtual function* **display()**.

Inherit two classes from the Media class, namely: **Book** and **Magazine**.

The **Book** class will have two strings to store the **author name** and **ISBN** of the book. Apart from the overloaded constructor, this class will implement the **display()** function which will display all attributes of a Book on screen in a neat and readable way.

The **Magazine** class will have a string to store the **month name** and an integer to store the **year of publication** of the magazine. Apart from the overloaded constructor, this class will also implement the **display()** function which will display all attributes of a Magazine on screen in a neat and readable way.

Now, implement a main function which should ask the user how many Media items the user wants to create, and store the value entered by the user in an integer variable **n**. Then, your program will dynamically allocate an array of **Media*** of size **n**.

After that, your program will ask the user to create **n** Media items. The user should be asked to enter 1 if he/she wants to create a Book and 2 if he/she wants to create a Magazine. Once the choice has been entered, your program should ask the user for all the attributes necessary for creating that item (Book or Magazine). Then, that item should be dynamically allocated and its address should be stored in the array of **Media***.

Once all Media items have been created, your program should traverse the array of **Media*** to display the details of each item on screen.

At the end, your program should properly deallocate all the dynamically allocated memory.

Task # 1.2

In order to accomplish this task, you will firstly need to implement a public member function **int getYear()** in the **Magazine** class.

Modify the program that you wrote in Task # 1.1 and implement a global function:

```
void searchByYear (Media**, int)
```

which takes the array of **Media*** and its **size** as parameters. This function should ask the user to enter a year, then it should search the array for all **magazines** of that year and display their details on screen.

Now, call the above function from the main function to search the array of **Media*** once all media items have been created.

Task # 1.3

Add another class **CD** to the inheritance hierarchy. The **CD** class will have an integer member variable to store the its **capacity** in MBs. Apart from the overloaded constructor, this class will also implement the **display()** function which will display all attributes of a CD on screen in a neat and readable way.

Also modify the main function to allow the user to create a CD by entering the option 3.

Task # 1.4

Add a **Shelf** class to store a list/collection of **Media** items. It will have the following declaration:

```
class Shelf
{
private:
        Media** items;
        int maxSize;
        int currSize;
public:
        Shelf (int);
        void insert (Media*);
        void displayContents ();
        ~Shelf();
};
```

The overloaded constructor will take an integer value as argument and initialize the **maxSize** to that value, and initialize **currSize** to 0. Constructor will also dynamically allocate an array of **Media*** through the member variable **items**.

The member function **insert(Media*)** will take a Media* as parameter, and store the received object into the next available index in the **items** array (use the member variable **currSize** to determine the next available index in the array). This function will also increment **currSize** to indicate the updated size of the array.

The member function **displayContents()** will display the details of all the items that are currently stored in the **items** array. This will be accomplished, simply, by calling the display member function on each Media item stored in the **items** array.

The destructor of the **Shelf** class will deallocate the dynamically allocated array which was allocated by the constructor.

Now, implement a main function, which should ask the user how many Media items the user wants to create, and declares a **Shelf** object to store those many items. After that the user should be asked to create those many Media items. After the creation of each Media item (Book, Magazine, or CD), it should be inserted into the Shelf. Once all items have been inserted, their details should be displayed by calling the **displayContents()** function.