Foundation for Advancement of Science & Technology

```
int main
{
int units, cost;
cin >> units
if (units < 100)
    cost = units * 5;
else if (units >= 100 && units <= 300)
    cost = units * 8 ;
else if (units > 300)
    cost = units * 10;
cout << cost ;
}

int main()
{
    int m, n, div, GCD; rem;
    cin >> m >> n;
    div = m;
        GCD = n;
    if (n < m)
        GCD = m;
        div = n;
    do
    {
        rem = GCD % div ;
        GCD = div ;
        div = rem ;
    }
    while (rem != 0);
```

```
int GCD (int m, int n)
{
    int rem;
    do
    {
        rem = m % n;
        m = n;
        n = rem;
    }
    while (rem != 0);
    return m;
}
```

Increase size of Dynamic Array

```
void
int* GrowArray (int *A, int oldsize, int newsize)
{
    int * newA = new int [newsize];
    for (int i = 0; i < oldsize ; i++)
        newA[i] = A[i];
    delete [] A;
    return newA;   A = newA;
}
int main ()
{
    int * arr = new int [10];
    arr = GrowArray (arr, 10, 20);
}
```

To Shrink an Array

```
void ShrinkArray (int* &A, int oldsize,
                  int newsize)
{
    int * newA = new int [newsize];
    for (int i = 0; i < newsize ; i++)
        newA[i] = A[i];
    delete [] A;
    A = newA;
}
```

Foundation for Advancement of Science & Technology

2D Char Array (names)

```
char ** names, temp[100];
int c-size;
cin >> c-size;
names = new char *[c-size];
for (int i=0 ; i< c-size ; i++){
    cin.getline(temp, 100);
    names[i] = new char [strlen(temp)+1];
    strcpy(names[i], temp);
}
for (int i=0 ; i< c-size ; i++)
    cout << names[i] << endl;
for (int i=0 ; i< c-size; i++){
    delete [] names[i];
    names[i] = nullptr;
}
delete [] names;
names = nullptr;
char ** f-name = new char *[c-size];
char ** l-name = new char *[c-size];

for (int i=0; i< c-size ; i++)
{
    int j=0;
    for (; names[i][j] != ' ' ; j++)
        temp[j] = names[i][j];
    temp[j] = '\0';
    int S = strlen(temp);
```

```
    f_name[i] = new char[s+1];
    strcpy(f_name[i], temp);
    for(; names[i][j] != '\0'; j++)
        temp[j-s] = names[i][j+1];
    temp[j] = '\0';
    l_name[i] = new char[strlen(temp)+1];
    strcpy(l_name[i], temp);
}

for(int i=0; i<c_size; i++)
{
    delete [] l_name[i];
    delete [] f_name[i];
    delete [] names[i];
    l_name[i] = nullptr;
    f_name[i] = nullptr;
    names[i] = nullptr;
}

delete [] names;
delete [] l_name;
delete [] f_name;
names = nullptr;
l_name = nullptr;
f_name = nullptr;
```

not necessary

Foundation for Advancement of Science & Technology

POINTERS

Pointers are special variables that store memory address of the specified data type.

Declaration:

datatype* variable_name;

e.g. int* ptr;

Initiallization:

datatype* variable_name = null var_name;

e.g. int* ptr = null ptr;

OR

int* ptr = &a;

- You cannot assign the value of an ordinary variable to a pointer.
- Pointers can only store the address of the specified datatype e.g. an integer pointer cannot store the address of a float variable.

Dereffference Operator (*):

*pointer_name means to use the pointer as the variable whose address was previously assigned to it.

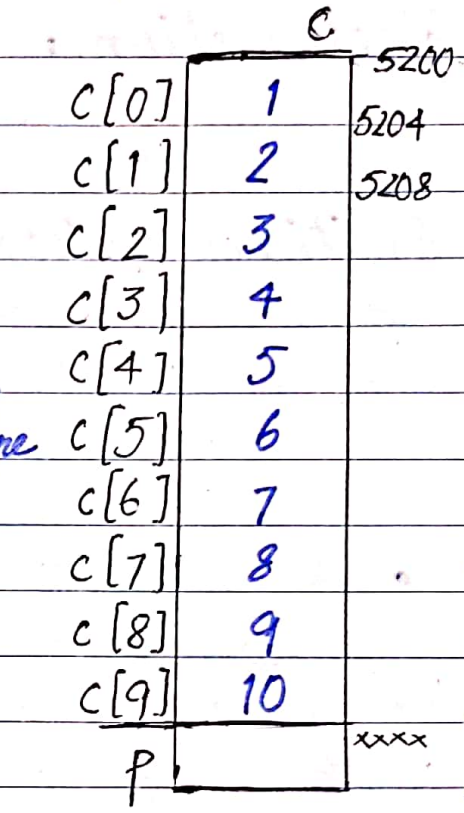
e.g.

```
int a = 10, b = 5, c = 15;
int *p = &a, *q = &b;
*p = 20;        // Now a = 20
cout << &a;     // 0x39740 (OR cout << p;)
cout << *p;     // 20
```

Foundation for Advancement of Science & Technology

One Dimensional Array:

```
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
int * p = c;  OR  int * p = &c[0];
```




```
cout << p;  // 5200
cout << &c[0];  // 5200
cout << c;  // 5200

for (int i=0; i<5; i++)
    cout << *(p+i); // p+i* size of datatype
// 1 2 3 4 5
  // cout << *p++;
```

[] subscript notation
() offset notation

Pointers in Functions:

Array pointers and arrays can be interchangably used in C++.
e.g. you can pass an array to a function that requires a pointer array and vice versa.

Foundation for Advancement of Science & Technology

Palindrome or not

```
int main()
{
    char c[90];
    cin >> c;
    isPalindrome(c);


}

bool isPalindrome(char c[])
{       bool flag = true;
    int size = strlen(c);
    char *p = c;
    for(int i=0 ; i<size/2 && flag==true ; i++)
    {
        if(*(p+i) != *(p+size-i-1))
            flag = false;
    }
    return flag;
}   OR
{
    char *p = c , *q = &c[strlen(c)-1];
    bool flag = true;
    for(int i=0; i<strlen(c)/2 ; i++)
        if(*(p+i) != *(q-i))
            flag = false;
    return flag;
}
```

```
void print (char * str)
{
    for(int i = 0 ; i< strlen (str); i++)
        cout << * (str +i) ;
}
int main()
{
    char s[] = "Hello World";
    print (s);
}
```

Swap Function

```
void swap (int * aptr , int * bptr)
{
        int temp = * aptr ;
        * aptr = * bptr ;
        * bptr = temp ;
}
int main ()
{
        int a, b;
        cin >> a >>b;          // e.g. a = 6 , b = 12
        swap (&a , &b);
        cout << a << b;        // a = 12 , b = 6
}
```

Foundation for Advancement of Science & Technology

Sorting an Array:

- Bubble sort (compares 2 elements at a time)
- Selection sort (selects largest from the whole array and places it at the end)

Constant Pointers:

- `const int * ptr = &a;`
  Above statement means that the value of the variable towards which the pointer is pointing (that is 'a' in this case) cannot be updated using the pointer. However, the value can be updated using the variable name itself. Also, the value of ptr can be updated.

- `int * const ptr = &a;`
  Above statement means that ptr can only point towards a i.e value of ptr cannot be updated. However, the value of the variable it is pointing towards can be updated.

- `const int * const ptr = &a;`
  (first point of above one and first two of the first one)

Sorting

```
int * findMin (int * arrptr , int size)
{   int min = *(arrptr), index =0
    for (int i= 1 ; i< size ; i++)
        if (*(arrptr+i) < min)
        {
            min = *( arrptr+i);
            index = i;
        }
    return index + arrptr;
}
void Sort (int a[], int size)
{
    for (int i=0; i< size ; i++)
    {
        int* ptr = findMin (a+i , size -i);
        swap (a[i], *(a+ptr));
    }
}
```

HOMEWORK

```
void Sort (int a[], int size)
{
    for (int i=0; i< size ; i++)
    {
        int idx = findMin (a+i , size -i);
        swap (a[i], a[idx +i];
    }
}
```

```
int * InputSet (int &s)
{
    cout << "Enter size of set: ";
    cin >> s;
    cout << "Enter elements of the set: ";
    int *set = new int [s];
    for (int i=0; i<s; i++)
        cin >> set [i];
    return set;
}
int *Intersection (int *A, int *B, int s1, int s2,
                   int &s3)
{   s3 = 0;
    for (int i= 0; i< s1; i++)
        for (int j=0; j< s2; j++)
            if (A[i] == B[j])
                s3++;
    int * intersection = new int [s3]; int idx=0;
    for (int i = 0; i< s1; i++)
        for (int j = 0; j< s2; j++)
            if (A[i] == B[j])
            {
                intersection [idx] = A[i];
                idx++;
            }
    return intersection;
}
```

Foundation for Advancement of Science & Technology

```
int main()
{
    int * set1, * set2, * intersection;
    int size1, size2, size3;
    set1 = InputSet(size1);
    set2 = InputSet(size2);
    intersection = Intersection(set1, set2, size1,
                                size2, size3);

}
```

Foundation for Advancement of Science & Technology

Dynamic Memory Allocation:

```
int n;
cin >> n;
int * arr = new int [n];
```

These above statements will create an array of size 'n' whose value can be assigned by the user i.e at runtime.

Deallocation:

```
delete [] arr; // to delete an array
delete mem; // to delete a single dynamic memory
```

A dangling pointer is the one that no longer points towards a memory location that is a part of our program's mem. In above case, arr and mem are dangling pointers. Since, dereferencing a dangling pointer can crash a program, we need to reallocate these pointers to nullptr.

i.e.
```
arr = nullptr;
mem = nullptr;
```

Consider the case:

```
int * p, * q;
p = new int [10];
q = p;
delete [] p; // or delete [] q, they are the same
```

In this case, we need to reallocate both p and q i.e.

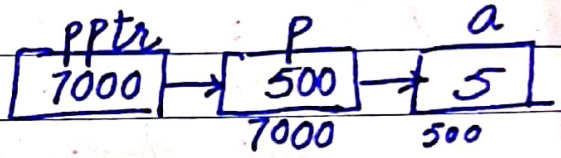
```
p = nullptr;
q = nullptr;
```

Pointer to Pointer:

It is used to store address of a pointer.

e.g.

```
int a, *p;
*p = &a;
int **pptr = &p;
cout << *pptr;  // 500
cout << p;      // 500
cout << pptr;   // 7000
cout << &p;     // 7000
cout << **pptr; // 5
```



- It is basically used to make 2D pointer array.

e.g.

```
int row, col;
cin >> row >> col;
int** m = new int *[row];
for (int i=0; i<row; i++)
    m[i] = new int [col];
```

This will allocate a 2D dynamic integer array

[m[i][j] = *(*(m+i)+j)]

- In dynamic 2D arrays, the size of each row i.e. number of columns in each row must not be the same.

Abstraction / Encapsulation

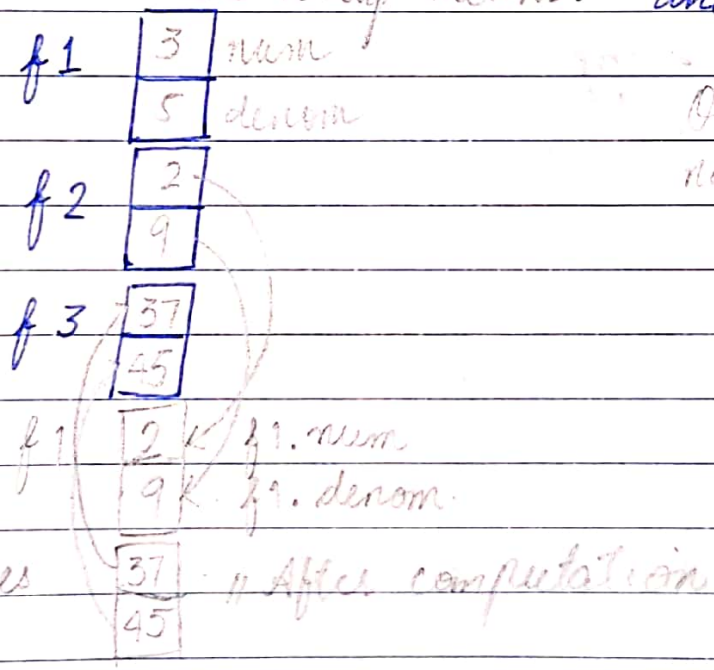
Example:

```
class fraction
{
    private:                    // access specifier
        int num;
        int denom;
    public:
        fraction (){
            num = 0;
            denom = 1;
        }
        void InputFrac ()
        {
            cin >> num;
            cin >> denom;
        }
        void OutputFrac ()
            cout << num << "/" << denom;
        fraction Add ( fraction f1)
        {
            fraction res; // res is object
```

```
        res.denom = f1.denom * denom;
        res.num = num * f1.denom + f1.num * denom;
        return res;
    }
};

int main()
{
    fraction f1;                    // objects, contains 2 values
    fraction f2, f3;                  size = 8 bytes.
caller object → f1.InputFrac();     // will input 2 variables i.e.
    f2.InputFrac();                   numerator & denominator
    f3 = f1.Add(f2);*               // This will assign f3 with
                                      the sum of fraction f1
                                      and f2 (value of res.)
```

* What this statement actually means:



Object is just a 'fancy name' for abstraction

Save a class as header file e.g here fraction.h

Foundation for Advancement of Science & Technology

Class (Set)

```
class Set
{
    int size;
    int * elements;

    Public:
        Set ();
        Set Union (Set);
        void Output ();
        void Input ();
        Set (int s);
        Set (int * set, int s);
        void remove ();
};
```

```
Set :: Set ()
{
    size = 0;
    elements = nullptr;
}
```

```
void Set :: Input ()
{
```

```
void Set :: remove ()
{
  if (size > 0)
  {
     delete [] elements;
     size = 0;
  }
  elements = nullptr;
}
```

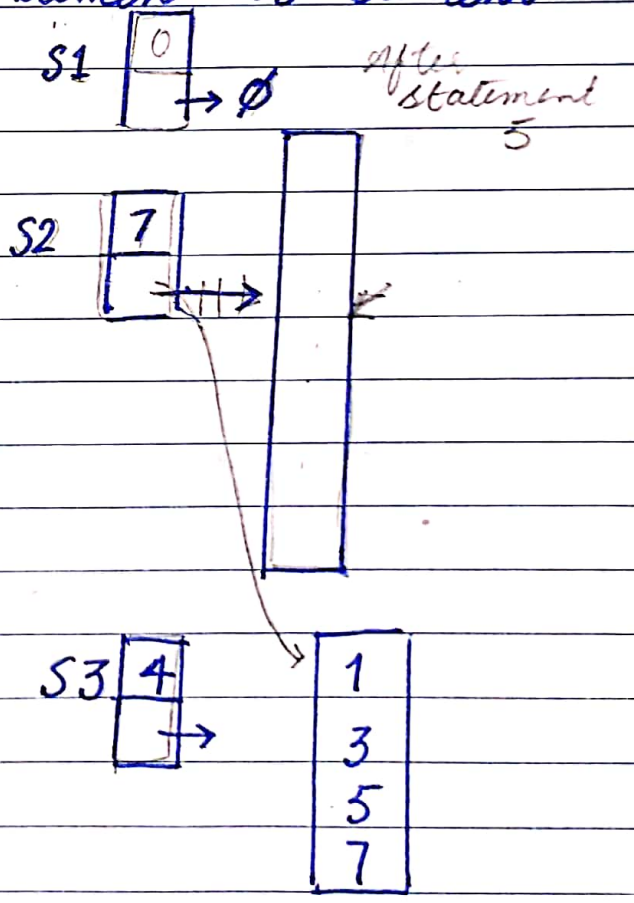
```
Set :: Set (int s)
{
    size = s;
    elements = new int [size];
}
Set :: Set (int *set, int s)
{
    size = s
    elements = new int [size];
    for (int i=0; i<size i++)
        elements [i] = set [i];
}
```

```
Set Set :: Union (Set S)
{
    Set U;
    U.size = size;
    for (int i = 0; i < S.size ; i++)
    {
        int flag = 0;
        for (int j = 0; j < size ; j++)
            if (elements[j] == S.elements[i])
                flag = 1;
        if (flag == 0)
            U.size ++;
    }
    U.elements = new int [U.size];
    int index = 0;
    for (int i = 0; i < S.size ; i++)
    {
        int flag = 0;
        for (int j = 0 ; j < size ; j++){
            if (elements[j] == S.elements[i])
                flag = 1;
            U.elements [index] = elements [j];
            index++
        }
        if (flag == 0){
            U.elements [index] = S.elements [i];
        } index++
}       }
```

Foundation for Advancement of Science & Technology

```
int main()
{
    Set S1, S3;          // This will allocate 2 sizes of value 0 and 2 element pointer arrays equal to nullptr.
    Set S2(7);           // This will allocate a set variable of size 7 and an array of 7 elements.
    int arr[] = {1, 3, 5, 7};
    Set S3(arr, 4);      // This will copy first 4 elements of arr in S3
    S2 = S3;             // S2.size = S3.size;
                         // S2.element = S3.element
```

(In this case, the value of memory location of 7 elements towards which S2 was pointing will be lost i.e memory loss)



Foundation for Advancement of Science & Technology

To define a function outside the class:

```
# include <fraction.h>
```

```
void OutputFrac();
fraction Add (fraction);
```
← statements in class

Fraction.cpp

```
void fraction :: Dulput Frac()
{
```
scope resolution operator
```
    cout << num << "/" << denom;
}

fraction fraction :: Add (fraction f)
{
    ....;
}
```

main.cpp

```
# include <fraction.h>

int main ()
{
    fraction f1, f2, f3;
    f1.InputFrac();
    f2.InputFrac();
    // f1.Add (f2); or f2.Add (f1); Same...
    f3 = f1.Add (f2);
```

```
void fraction :: simplify ()
{
    int GCD = 1;
    for (int i = 2 ; i <= num || i <= denom; i++)
        if (num % i == 0 && denom % i == 0)
            GCD = i;
    num /= GCD;
    denom /= GCD;
}
```

Constructors and Destructors

Constructor is used to initialize the data member of a class. Destructor is used to deallocate or delete members of a class.

Constructors:

```
class name ();
```

They donot have a return type. Constructors may accept parameters (amount of parameters depends on the amount of members of the class) e.g.

```
class name (datatype, datatype);
```

- There can be more than one constructor depending on the number of parameters passed but there is only one destructor in a class.

Foundation for Advancement of Science & Technology

Destructor:

~'class name ();

'~' differenciates a destructor from a constructor. A destructor never accepts any parameter nor does it have a return type.

```
class set {
{
    int size;
    int *ele;
public:
    set ();
    set (int);
    Set (int*, int);          } constructors
    void Inputset ();
    void Outputset ();
    ~ Set ();                 — // destructor.
    Set Intersection (Set)
    Set ( Set &);
};
Set :: Set ( int s)           // constructor
{
    size = s;
    ele = new int [size];
}
```

Foundation for Advancement of Science & Technology

```
Set :: ~Set()    // destructor never accepts a
{                          parameter
    if (size > 0)
        delete [] ele;
}
Set :: Set()          // constructor
{
    size = 0;
    ele  = nullptr;
}
Set :: InputSet()
{
    (cin >> size; ele = new int [size];) // optional
    for (int i=0; i< size; i++)
        cin >> ele[i];
}
Set :: OutputSet()
{
    for (int i=0; i< size; i++)
        cout << ele[i] << "\t";
    cout << endl;
}
```

(Issues due to destructor)

issue 1

```
Set Set :: Intersection (Set &s)   // in order to avoid deletion by destructor
{                                  // we pass parameters by reference.
    int count = 0;
    for (int i = 0; i < size; i++)
        for (int j = 0; j < s.size; j++)
            if (ele[i] == s.ele[j])
            {
                count++;
                break;
            }
    Set intersection(count);   count = 0
    for (int i = 0; i < size; i++)
        for (int j = 0; j < s.size; j++)
            if (ele[i] == s.ele[j])
            {
                intersection.ele[count] = ele[i];
                count++;
                break;
            }
    return intersection;   // At this point, destructor will be called and intersected set will be lost.
}
```

Set *intersection = new int [count]; Dynamic object

issue 2

For this, either pass the resultant set by reference as well and set return type to void or perform deep copy (i.e create copy of the array) instead of shallow copy (i.e referring to the same array using pointers)

Foundation for Advancement of Science & Technology

Copy Constructor:

To perform deep copy, copy constructor is used. Like all the constructors, it does not return any value i.e. does not have a return type.

Copy constructor syntax:

class name (class name *&) * as datatype.

Here, the class datatype passed as parameter differentiates copy constructor from default constructor.

e.g.

```
Set :: Set ( Set & copy)
{
    size = copy.size ;
    ele = new int [size];
    for (int i = 0 ; i< size ; i++)
        ele [i] = copy.ele [i];
}
```

When Constructors / Destructors are Called:

Constructor:

When an object is made, constructor is automaticdlly called. The type of constructor called depends on the numbes of parameters passed.

Foundation for Advancement of Science & Technology

Copy Constructor:

- It is called whenever an object is passed by value.
- It is also called when an object is returned by a function.
- When an object is initiallized at the time of declaration. e.g.

```
Set S3(S1);
```

This will copy contents of S1 to S3.

Destructor:

It is called when the lifetime of an object ends i.e. end of a function or main. Also, when delete operator is called for an object pointer, it is called.

Dynamic Objects:

Creating dynamic objects is another way of avoiding deletion by destructor since lifetime of dynamic objects is controlled by the programmer.

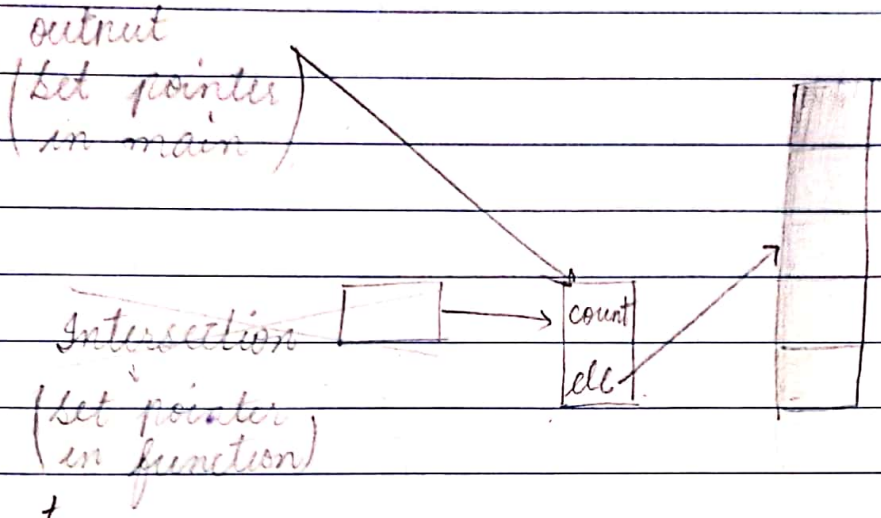
e.g. in case of intersection:

```
Set * intersection = new int [count];
```

Here, intersection is a pointer that contains address of the object. Also, Arrow operation (→) is used while accessing members of class instead of dot (.) operator.

Foundation for Advancement of Science & Technology

After Return statement



'This' Operator:

'this' is a default operator which is basically a pointer that points towards the caller object.

e.g. in class "Set"

```
Set & SetSize(int a)
{
    size = a;
    return * this;
}

Set & SetEle (int*p)
{
    ele = p;
    return * this;
}
```

```
int main()
{
    Set S1;
    int * ptr = new int [7];
    S1.SetSize(7).SetEle (ptr);
// This is known as cascading
here first, the size of set1
will be updated and
then the elements. After
this statement, size of
set1 will be 7 and ele
will be pointing where
initially ptr was pointing
}
```

Foundation for Advancement of Science & Technology

Operator Overloading:

An operator can perform different tasks depending upon the datatype e.g '+' can add two integers as well as float values or strings.

Operator overloading is defining a new task for a specific operator for a user difined datatype.

Operator overloading does not change:

- arity i.e '+' will remain binary
- operator precedence.
- associativity that is initially left to right.
- the definition of operator for built-in datatypes.

Example:

```
class fraction ()
{
    int num;
    int denum;
public:
    fraction ();
    fraction (int , int );
    fraction operator+ ( fraction f);
    fraction operator + (int )
    void input ();
    void output
    fraction operator+= ( fraction f);
```

```
    fraction operator++(int);   // post increment
    fraction operator++();      // pre increment
};

fraction :: fraction()
{
    num = 0;
    denum = 1;
}

fraction :: fraction(int a, int b)
{
    num = a;
    denum = b;
}

fraction fraction :: operator+(fraction f)
{
    fraction res;
    res.denum = denum * f.denum;
    res.num = num * f.denum +
              denum * f.num;
    return res;
}
```

```
fraction fraction :: operator+(int a)
{
    fraction f(a, 1);
    return (*this) + f;
    // OR return this->operator1(f);
}

fraction fraction :: operator+=(fraction f)
{
    return (*this) + f;
}
// Defining '+' does not mean '+=' is a
   valid operation for user defined
   datatypes. It must be separately defined

void fraction :: input()
{
    cin >> num;
    cin >> denum;
}

void fraction :: output()
{
    cout << num << "/" << denum;
}
```

```
fraction fraction :: operator ++(int) // post inc.
{
    fraction temp = * this ;  // to return org fraction
    * this = * this + 1;      // to increment the
    return temp ;                caller fraction/object.
}
fraction & fraction :: operator ++() // pre inc.
{
    * this = * this + 1 ;     // to increment the
    return * this ;              caller object.
}
```

You can use all public functions in other functions outside the class just as they are used in main.

```
fraction operator + (int a, fraction f)
{
    return f + a; // This will call operator+
                     function from the class
                     and compute the output
}
```

```
int main()
{
    fraction f1, f2(1,2), f3(2,3);
    f1 = f2 + f3;      // f1 = f2.operator+(f3);
                          for compiler
    f1 = f2 + 5;       // f1 = f2.operator+(5);
                          i.e. first fraction will
                          be the caller object
    f3 = (f1++) + f2;  // compute f1+f2 and
                          store in f3, then
                          increment f1.
// Left hand side operand is alway the caller object. Hence, the statement such as: int a=5;
        a += f2;
will generate an error since the caller object does not belong to the class fraction
```

friend: * so that private members can be accessed outside class.

The keyword 'friend' is used to build a connection between two classes or functions*

e.g. in class 'fraction', we can declare a friend function as:

```
friend ostream& operator << (ostream & cout,
                             fraction f);
```

The friend function can be both private or public.

Foundation for Advancement of Science & Technology

For friend function definition outside the class:

```
ostream& operator<<(ostream& out, fraction f)
{
    out << f.num << "/" << f.denum << endl;
    return cout;
}
```

- If a class A is a friend of class B, it means that you can access members of B in A but it does not mean that B can access members of A

* Now, in main, the statement:

```
cout << f1;
```

means: operator<<(cout, f1);

Returning cout in function enables cascading, hence:

```
cout << f1 << f2 << endl;
```

is a valid statement.

```
[cout << f1] << f2 << endl;
operator<<(cout, f1); → returns cout
→ [cout << f2] << endl
operator<<(cout, f2); → returns cout
→ cout << endl;
```

Foundation for Advancement of Science & Technology

```
class set
{
    int size;
    int * ele;
    const int a;
public:
    Set & operator= (Set & right);
    Set ();
    Set (int, int);
    void out () const; // to declare constant function
};

int main ()
{
    Set S1, S2(3), S3(7);
    S1 = S2;          // S1.operator=(S2);
    S3 = S3;          // In this case, we do not want deep copy hence we applied self assignment check in the function.
}
```

```
Set & Set :: operator=(Set & right)
{
    if (this != & right) // to avoid self assignment
    {
        if (size >0)      // to delete already
            delete [] ele;    allocated memory
        size = right.size;
        ele = new int [size];
        for (int i=0; i< size ; i++)
            ele[i] = right.ele[i];
    }
    return * this;    // to enable cascading
}

Set :: Set (): a (0) ← member initializer list
// you initialize constant data member in
   member initializer list and not in the
   function definition. Other members may
   also be initialized here.
{
    ele = nullptr;
    size = 0;
}

Set :: Set (int s, int a1): a (a1)
{
    size = s; ele = new int [size];
}
```

Usually you need to make a non-member function when the left hand side operand is not an object of the class such as in case of 'cout' or 'cin'. It is preferred not to make non-member functions when member function can be made.

Static Variables:

Keyword 'static' is used to declare static variables. In classes, when we need to make a variable that is shared by all the objects i.e. that variable has only one instance.

e.g. it can be used to count the total number of objects of a class.

* can be made public
* not initialized in constructors since it is common for all objects
* it exists even when there is no object of the class.

Relationship

- Composition / Aggregation

⇒ whole / part relationship

⇒ composition : jis mai class ka part , whole k life time k sath khatam ho jaye

⇒ aggregation : jis mai zaroori nai k whole k khatam honay k sath part bhi khatam ho jaye

- Association

⇒ ik class dosri class k kch functions ko use karay

- Inheritance

⇒ 2, 3 [a more] common functions ki separate class banana and then each class wo class inherit / use karay gi

- Polymorphism

⇒ functions common hon lekin har class k liye diff kaam ke rahay hon

part :

```
class Point
{   friend class circle;
    int x_co;
    int y_co;
Public:
    Point()
    {
        cout << "point default constructor";
        x_co = y_co = 0;
    }
    Point(int x, int y)
    {
        cout << "point overloaded constructor";
        x_co = x;
        y_co = y;
    }
    void printpoint()
    { cout "(" << x_co << "," << y_co <<
};                                  ")";}
```

whole :

```
class Circle
{
    Point center;
    float radius;
Public:
    void printinfo()
{
    cout << "radius" << radius;
    center.printpoint();
}
    circle() {
    radius = 0;
} cout << " circle default constructor";
```


Scanned with CamScanner

Composition / Aggregation

Composition:

```
class point{
{   friend class circle;  // Now circle can access
    int x_co;             //  private members of
    int y_co;             //  class point.
public:
    point()
    {
        x_co = y_co = 0;
    }
    point(int x, int y)
    {
        x_co = x;
        y_co = y;
    }
    void printpoint()
    {
        cout << "(" << x_co << ", " << y_co << ")";
    }
};
```

↑ this statement may be placed anywhere ins the definition of this class.

Foundation for Advancement of Science & Technology

```
class circle
{
    point center;
    float radius;
public:
    void printinfo()
    {
        cout << "Radius: " << radius;
        // cout << "(" << center.x_co << ", " <<
        center.printpoint();
    }
    circle()
    {
        radius = 0;
    // We do not have to initialize attributes
    of center to zero since the default
    constructor has already been called
        of class point
    }
    circle(int x, int y, float r): center(x, y)
    {
        radius = r;
    }
};
```

Foundation for Advancement of Science & Technology

Dynamic Object

March 31, 2022. Thursday

```
Set* Set :: Intersection (Set S)
{   int count = 0;                    // Set * this;
    for (int i=0; i<size; i++)
      for(int j=0; j<size; j++)
        if(ele[i] == S.ele[j])        // this -> ele[i]
          { count++;
            break;
          }
Set * sptr;                           // constructor will be automatically called
sptr = new Set(count);
```

if we'd written

```
sptr = new Set;
```

default constructor will be called.

```
  int k=0;
  for(int i=0; i<size; i++)
    for(int j=0; j<size; j++)
      if(ek[i] == S.ele[j])
        { sptr -> ele[k++] = ele[i];
          break;
        }
  return Set;
}
```

-> is called when we have address but not value and we cannot call any function.

```
int main()
{
   Set * sp;
   sp = S2.Intersection(S3);          // & S2
   sp -> output Set();
   delete sp;                         // destructor will be automatically called
}
```

-> no use of [] because its a class not a int.

```
Set :: Set(int size)
{  this -> size = size;
}

Set :: Set(Set & copy)
{
   size = copy.size
   ele = new int [size];
   for(int i=0; i<size; i++)
      ele[i] = copy.ele[i];
}
```

or *(this)=copy ⇒ assignment operator

```
Set & Setsize(int a)
{  size = a;
   return *this;                      // has callee object ka address
}

Set & Setele(int *p)
{  ele = p;
   return * this;
}
```


Scanned with CamScanner

```
int main()
{
    set s1;
    int * ptr;
    s1.setsize(7).setele(ptr).output();
```

cascading

* cascading cannot go further after a void function.

Scanned with CamScanner

FAST
Foundation for Advancement of Science & Technology

Inheritance: (reuseability)

The concept of inheritance is basically used for the purpose of reuseability. It reduces redundancies in a program.

e.g.

```
class salaried_Emp
{
    char * name;
    char * designation;
    Date DOB;
    int salary;
}
```

```
class Commissioned_Emp
{
    char * name;
    char * designation;
    Date DOB;
    float commission;
    int sales;
}
```

```
class Hourly_Emp
{
    char* name;
    char* designation;
    Date DOB;
    int hours_worked;
    int per_hour_rate;
}
```


Scanned with CamScanner

```
class Emp → (base / parent class)
{
    char* name;
    char* designation;
    Date DOJ;
}
// Derived / Child classes:  → type of inheritance
class Salaried_Emp: Public Emp
{
    int Sakary;   // Salaried_Emp will have a
}                    total of 4 attributes

Class Hourly_Emp: Public Emp
{
    int hours_worked;
    int per-hour_rate;
}

Class Commissioned_Emp: Public Emp
{
    float commission;
    int Sales;
}
```

- Public members are public in derived class.
- Private members of base class are hidden members of derived class hence won't be accessible.

Scanned with CamScanner

Foundation for Advancement of Science & Technology

- The "protected" members of base class are also protected in derived class. The keyword protective means that attributes will be easily accessible in derived class but will be private for all other functions and classes But using keyword 'protected' is not considered good programming practice.

- Following functions cannot be inherited
  - constructors
  - destructor
  - assignment operator

- There can be multiple levels of heirarchy in inheritance. e.g. there is a derived class of salaried_Emp called Commissioned_salaried_Emp. Here this derived class will have access to both salaried_Emp and its base class Emp.

- Parent and child class both can have a function with same name and attributes this is called function overriding e.g there is a function in salaried_Emp:

```
void print() const{
        Emp :: print(); // to call print() of Emp class
        cout << "Salary: " << salary;
}
```


Scanned with CamScanner

- Member initialization will be done in the same way as it was done in composition e.g. for overloaded constructor of Salaried_Emp

```
Salaried_Emp(char*n, char*d, Date date, int s):
Emp (n, d, date) // overloaded constructor of emp
{
    Salary = s;
}
```

- There can be three types of inheritance
  - public
  - protected
  - private

Type of Inheritance	(Base class) Data Members		
	public	protected	private
public *	public	protected	hidden
protected	protected	protected	hidden
private	private	private	hidden

* public is the most used type of inheritance

Scanned with CamScanner

Foundation for Advancement of Science & Technology

POLYMORPHISM (many forms)

A way of programming through which we can perform 'generic' programming. Through polymorphism same function or object behaves differently depending upon the condition. Polymorphism is applied to classes that are related to each other by inheritance. Through polymorphism, you can call a member function that will execute a specific function depending on the caller object.

```
class Emp
{
    char * name;
    char * des;      // designation
    Date DOJ;        // date of joining
public:
  * void print()
    {
        cout << "Name: " << name;
        cout << "Designation: " << des;
        cout << "Date of Joining: " << DOJ;
    }
  * virtual
```


Scanned with CamScanner

```
class Salaried_Emp : Public Emp
{
    float monthly_salary;

  public:
    void print ()
    {
        cout << "Salaried_Employee \n";
        Emp :: print ();
        cout << "Monthly Salary : " << monthly_salary;
    }
    float getSalary()
    {
        return monthly_salary;
    }
}
int main ()
{
    Emp E;
    Salaried_Emp S;
    Emp* eptr = & S;      // valid statement....
    eptr → print () ;     // will call print() of class
                          Emp but if keyword 'vir-
                          tual' is used print () of
                          Salaried_Emp will be called
    eptr → getSalary(); X ← not allowed / invalid
}
```


Scanned with CamScanner

EARLY/LATE BINDING

important concept

- If there is a relation of parent/child (base/derived) between two classes. We can assign address of child class to a pointer of parent class without using explicit type casting e.g. Salaried_Emp a;
  Emp * ptr = &a; ← valid statement
  Child class may also be indirect child class.
- In above case, if we call a function that is in both the classes through the pointer, it will call the function of parent class.
- Through pointer of parent class pointing to an object of child class, you cannot access members of child class other than those which are 'redefined' in child class.
- * To call the 'redefined' functions of child class we must add a keyword "virtual" to the function of parent class. e.g. in case of parent class Emp, we shall add 'virtual' in the print function:
  virtual void print();
- The keyword 'virtual' leads to late binding i.e. the decision of which print function will be called is delayed till runtime rather than at compile time.

Scanned with CamScanner

```
class Dlv_Emp : public Emp
{
    float * Dsalary;

public:
    Dlv_emp(int d)
    {
        Dsalary = new int[d];
    }
    ~Dlv_emp()
    {
        delete [] Dsalary;
    }
virtual void print()
    {
        .....
    }
```

```
class Emp
{
    char* name;
    char* des;
    Date DOJ;

public:
virtual void print()
{
    ....
}
virtual ~Emp()
{
    delete name;
    delete [] des;
}
virtual float compute_
Salary() = 0;
// pure virtual function
```


Scanned with CamScanner

```
int main ()
{
    Emp * employees [5];
    employee [0] = new S_emp (....);
    employee [1] = new H_emp (....);
    employee [2] = new Dlv_emp (....);
    employee [3] = new S_emp (....);
    employee [4] = new H_emp (....);
    for (int i = 0; i < 5; i++)
        employee [i] → print ();
/* The above line will call print of each
child class respectively since we have used
keyword 'virtual' before print function of
parent class */
    for (int i = 0; i < 5 ; i++)
        delete employee [i];
/* This again will call destructors of all
respective classes */

    Emp e1; → //invalid statement / compile
                time error.
    Emp * e1 = new emp (....); // invalid

}. * Here we shall have to add virtual to the
destructor of both parent and child class
since destructor is not inherited.
```


Scanned with CamScanner

Foundation for Advancement of Science & Technology

Pure Virtual Function:

A pure virtual function is the one that is present in parent class but has no definition of itself. It is applied to avoid redundancy and apply concepts of polymorphism.

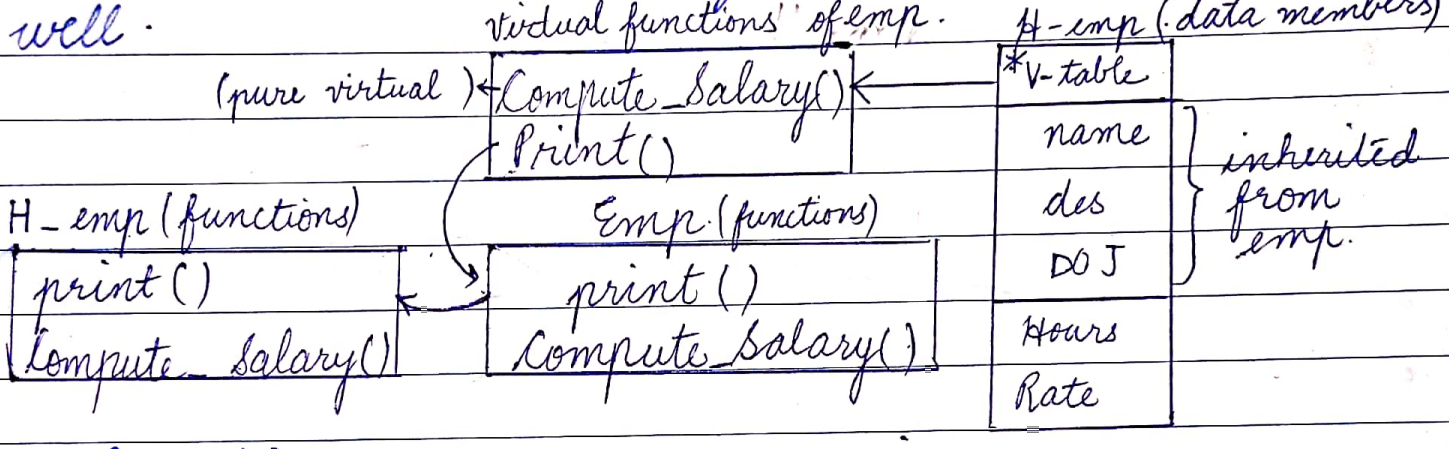
- Once a pure virtual function is made in a class, the class becomes 'Abstract' i.e. there can be no object of this class though pointers can exist.
- If you do not overwrite the pure virtual function in child class, the child class also becomes an 'abstract class'

★ It is a good programming practice to use keyword 'virtual' before every function

Scanned with CamScanner

Virtual Table:

There exists a virtual table (V-table) for every base class that has a virtual function in it. This V-table contains pointers of all the virtual functions. This V-table is assigned to each of the objects of that class. V-table is inherited by the derived class as well.



Explanation:

employee[1] -> print();

- The above statement will move the execution to the V-table of hourly-emp.
- After derefering the V-table, the execution will move to pointer of print function in the V-table.
- Now, this pointer will be derefered and the corresponding print function will be executed depending on where the pointer is pointing.
- Initially, this pointer points to the virtual

function of base class.
- If function is overridden in derived class, the pointer in V_table of derived class points to the over-rided function in derived class.

Template:

We can add templates in programming that consist of a few tasks that are required to perform a particular type of tasks.

Two types of templates:
- Function templates
- class templates

Templates are used to write a generic code for all datatypes i.e. this code is compatable with all datatypes.

- For defining a template you must add the keyword 'template' before the function or the class:

template (typename < variable_name>)

OR

template (class < variable_name>)

e.g.

```
template (typename T)
void sort (T arr[], int size)
{
    for (int i=0; i< size ; i++)
        for(int j= 0; j<size-i-1; j++)
            if (arr[j] > arr[j+1])
                swap (arr[j], arr[j+1]);
}

int main()
{
    fraction f1, f2(3,5), f3(3,9);
    fraction f_arr[3]= { f1, f2, f3};
    sort (f_arr, 3); → valid
// This will first check if there is a sort function in the class fraction. If there isn't any, it will look for template sort function and execute accordingly.
    int arr[3] = {5, 7, 2};
    sort ( arr, 3); → valid
}
```

* if '<' operator is not defined in fraction, then using this sort function may generate an error.

Foundation for Advancement of Science & Technology

Example class: // Multiple common types/temp types will be comma separated

```
template < typename T>
class Set
{
    T* ele;
    int size;
public:
    Set ()
    {
        size = 0;
        ele = nullptr;
    }
    Set (int s)
    {
        size = s;
        ele = new T[size];
    }
    Set (T arr[], int s)
    {
        size = s;
        ele = new T[size];
        for (int i=0; i< size ; i++)
            ele[i] = arr[i];
    }
    Set<T>* intersection (Set<T>& A)
    {
        int count = 0;
```

{1, 2, 3}

C.ele[count] = ele[i]; {2, 3, 6}

```
for ( int i=0; i< size ; i++)
    for ( int j = 0 ; j < A.size ; j++)
        if ( ele[i] == A.ele[j])
            count++;
Set <T>* intersection = new Set<T> ( count );
                                    count = 0;
for ( int i = 0 ; i < size ; i++)
    for ( int j = 0; j < A.size ; j++)
        if ( ele[i] == A.ele[j])
        {
            intersection.ele[count] = ele[i];
            count++;
        }
return intersection;
}

main ()
{
    int a[5] = {1, 2, 3, 4, 5};
    Set <int> S1(a, 5); // This will make a set
                           of integers.
    float f[3] = {0.6, 1.5, 3.2};
    Set <float> S2(f,3); // This will make a set
                            of float values.
}
```

Foundation for Advancement of Science & Technology

Exception Handling:

It is a method used to stop the program from terminating abnormally. A program terminates abnormally when there is a runtime error e.g. when a number is divided by zero or square root of a negative number or when we try to access a memory location that is not part of the program.

Try & Catch Blocks:

Try & catch block is the method to handle exceptions. Its syntax is:

```
try{

}
catch(<datatype><variable>){

}
```

- There can be multiple catch blocks for a single try.
- It is better to add default catch block to each try block so that there is no chance of abnormal termination even if relevant catch block does not exist.

```
int main(){
{   string s = "Invalid Input";
    int dividend, divisor;
    float res;
    try←(keyword)
    {
        cout << "Enter dividend:";
        cin >> dividend;
        cout << "Enter divisor:";
        cin >> divisor;★
        if (divisor == 0)          // 'throw' keyword is
            throw divisor;         // to move exception
        res = (float) dividend/    // to the relative
              divisor;             // catch block i.e with
        cout << "Answer:" << res; // 'int' as a parameter
    }
    catch (int x)
    {
        cout << "Divided by:" << x;
    }
    catch (...)      // default catch block
    {
        cout << "some error has occured.";
    }
    // Here, any datatype except for 'int' will
    // call default catch block.
}
```

cin is in failed state ↑

```
★ if (!cin)
      throw s;
  // the cin statement
  // fails if the input
  // is not of excepted
  // datatype
```

used.

Foundation for Advancement of Science & Technology

Default catch Block:

Default catch block is added to avoid any chance of abnormal termination. The default catch block is executed when the datatype thrown is not in the parameter of any block.

- Default catch block is written AFTER all other catch blocks i.e. at the end.

stdexcept:

It contains 'predefined errors'. By adding this library, we not have to 'throw' all the exceptions by ourself. e.g. in case of 'divided by zero' exception we can write the catch function as: -> is not in the library.

```
catch (divide_by_zero e)
{
    cout << "divided by" << e";
}
```

Foundation for Advancement of Science & Technology

```
int main()
{
    try
    {
        int * arr [100];
        for (int i = 0; i < 100 ; i++)
            arr[i] = new int [10000000];
    }
    catch (bad_alloc bae)
    {
        cout << "Exception occured";
        cout << bae.what();
// what() is a function defined in stdexcept
// that contains the error message.
    }
}
```

User-defined Exceptions

We can define new classes to handle exceptions. In this case, the exception shall not be automatically created, we shall have to use 'throw' keyword.

```
class div_by_zero
{
    string msg;
public
    div_by_zero()
    {
        msg = "divide by zero";
    }
    div_by_zero (string s)
    {
        msg = s;
    }
    string what ()    // what () should be defined
    {                 // so there is uniformity
        return msg;   // among all exceptions since
    }                 // all predefined exceptions
};                    // have this class.

int main()
{
    try
    {
        float res;
        int dividend, divisor;
        cin >> divident >> divisor;
        if (divisor == 0)
            throw div_by_zero;
```

```
        res = (float) divident / divisor;
        cout << res;
    }
    catch ( div_by_zero e)
    {
        cout << "Exception occured";
        cout << e.what();
    }
```

Stack Unwinding

Stack contains the record of functions and where they are called. For example there are 3 functions. main function calls function 1, function 1 has a statement that calls function 2 and function 2 calls the function 3. Consider, an exception occurs in function 3. If there is a catch block in function 3, the exception will be handled otherwise, it will be transferred to function 2 (caller function). Now, if function 2 has a catch block, the exception will be handled otherwise it will be transferred to function 1 (caller function) and so on.

Multiple Inheritance:

The type of inheritance in which a class can be inherited from two or more classes

e.g.

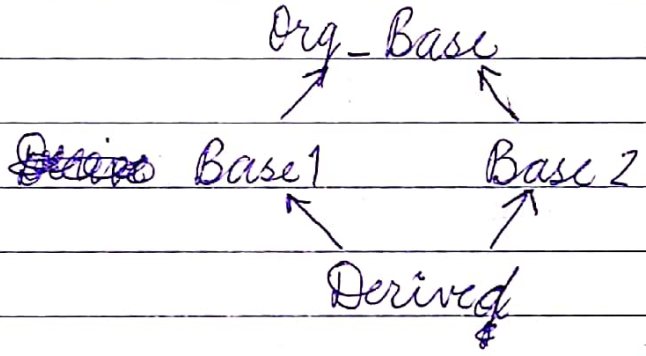
```
class Base1
{
    int i;
public:
    void PrintData()
    {
        cout << i;
    }
};
```

```
class Base2
{
    char c;
public:
    void PrintData()
    {
        cout << c;
    }
};
```

```
class Derived : public Base1, public Base2
{
    double d;
public:
    void print()
    {
        * PrintData();    // error.
        * Base1:: PrintData();    // correct.
    }
```

Issues in multiple inheritance:

- Since the base classes are independent of each other, they may contain same functions i.e. with same name and parameter list. For this, we shall have to use scope resolution operator to tell the compiler which function is being called.
- **Diamond Problem:** If base classes have also been derived from a class, then there will be issues in the derived class. There will be duplication of members of the class from which the base classes have been derived.



Base 1 and Base 2 both have inherited the members of Org-Base and thus, there will be duplication in derived. One method is to use scope resolution operator to access those members.

Foundation for Advancement of Science & Technology

- Another method to resolve diamond problem is use the method of virtual inheritance