1(a) Functions that are declared within classes are automatic inline functions. Inline keyword isn't needed here but can be added, so there are basically two types of inline functions, automatic and manual. If inside class defined function is a prototype then we have to make it inline manually otherwise it's auto inline.

Advantages: Parameterized macros aren't safe compared to inline functions as that macro can be used wrongly by mistakenly putting them in false areas, but as inline functions are functions, so. when that keyword is used, it will give compile error and inline functions also don't have space overheads so, it won't delay or won't give memory limit exceed while inline functions are small.

1(b) If an object is passed as a regular function parameter, it will bitwise copy itself and when functions work is done the parameter object will call it's destructor and it will free the memory of p but here is the problem, p this p is a pointer so, earlier bitwise copy was same pointer. So, when program ends that object will call it's destructor but that was already freed so, the program will behave undefinely here.

# copy constructor

```
myclass(myclass &obj) {
    p = (int *) malloc(sizeof(int));
    *p = obj.get();
}
```

1(c). Friend functions in a class is ~~used~~ not a member function of that class but it can access all members of that class. Friend functions can't be called by objects of that class.

friend functions are mostly used when operator overloading is needed; when operators overloading is done, we sometimes need other classes data on (int + class) when these cases arise, friend functions are the only option. By doing so, we can access those classes private members outside of that class and do our overload the way we want. But '=' operator can't be overloaded with friend functions.

forward declaration:

class myclass;

1(d). Reference is the address of something (variable or class). When we write a variable as reference it becomes l-value and it's value when changed, it's change by address.

Scenario - 1: when copy constructor is not there

" - 2: when we want to use the return value as L-Value.

" - 3: for swapping without pointers

Reference use dot operator because it accesses the class as objects and not pointers.

we know that when class is accessed as pointers, (->) operator is needed but reference access it as objects and thus uses dot operator.

2(a). Default argument can overload constructor because in different cases, it behaves differently like, when default argument is added to a constructor. Like below,

myclass(int a, int b=0, int c=0){.....}

we can call this class's constructor like,

myclass(2), myclass(2,3), myclass(2,3,4)

So, here myclass constructor is overloaded. If we didn't use default constructor here we would've needed three constructors so, this constructor is overloaded.

This function prototype is incorrect as we know that, if one parameter is made default, pass that parameter, all of them should be default. So, correct way is,

double area(double width=0, double length=0);

2(b) (i)

```
coord operator + (int i)
{
    coord temp (x+i, y+i);

    return temp;
}
```

(ii) Inside coord,

```
friend coord operator + (int, i, coord &ob)
{
    coord temp;

    temp.x = i + ob.x;

    temp.y = i + ob.y;

    return temp;
}
```

(iii) void setx(int i=0) { x=i; }

int getx() { return x; }

void sety(int i=0) { y=i; }

int gety() { return y; }

(iv) class coond {

coond(int i=0, int j=0) { x=i; y=j; }

void getxy(int &i, int &j) { i=x; j=y; }

coond operator+(int i);

friend coond operator+(int i, coond &ob);

void setx(int i=0); void sety(int j=0);

int getx(); int gety();

};

2(c): Protected access specifier is needed because when we want to access a class in this mode public members in that class is made to protected So, one can access it in derived class without exposing it to the outer world. The protected access specifier is very useful in terms of inheritence and when a function or member is protected, it stays hidden in outer world but can be used in derived class.

3(a).

```
cout << setw(10) << 5 << endl;
cout << setfill('%') << setw(10) << 5 << endl;
cout << Left << setfill('%') << setw(10) << 5 << endl;
cout << setprecision(7) << 13 << endl;
cout << setprecision(4) << Left << setfill('%')
     << setw(10) << endl;
```

3(b).

```
ostream& setup(ostream &os, int w, int prec)
{
    os.width(10);
    os.precision(4);
    os.fill('*');
    return os;
}
```

3(c).

```
class queue : public list {

void store(int i)
{
    list* node = new list;
    node->num = i;
    node->next = null;

    if(head == null) head = node;
    else
    {
        tail->next = node;
        tail = node;
    }
    delete node;
}
```

```
int retrieve()
{
    if (head = null) return 0;
    else
    {
        int tmp = head->num;
        head = head->next;
        return tmp;
    }
```

4(a). Early Binding

(i) It is done in compile time.

(ii) It's called Static binding.

(iii) When Base pointer access derived class, it always take base class function.

example: class Base{
    void show(){cout<<"Base";}

Late Binding

(i) It is done in run time.

(ii) dynamic binding.

(iii) By using virtual, it can access derived class function

(iv) Here we just write Virtual infront of show function.

```
class derived : public Base
{   void show(){cout<<"der.."; }
};
int main() { Base * ptr = new derived;}
```

A(b). Template <class T>

```
class Input {
 T data;
 Input ( String s, T min, T max)
 {
     cout << s << endl;

     cin >> data;

     while ( data > max || data < min)
     {
         cout << s << endl;

          cin >> data;

     }

}
```

4(c). ~~assuming the map is public~~

```
Template < class K, class V >
V function ( vector < pair < K, V >> &map, K key
{
    for(auto it = map.begin(); it != map.end();
        it++)
    {
        if ( it.first == key)
        {
            return it.second
        }
    }
return V();
}
```