Probleme

1.

```
#include <iostream>
using namespace std;
class B {
protected:
    int i;
public:
    B(int j = 5) {
        cout << " cb ";
        i = j - 2;
    }
    B(const B &ob) { cout << "CC B"; }
    ~B() { cout << " db "; }
    int get_i() { return i; }
};

class D1 : public B {
    int j;
public:
    D1(int k = 10) {
        cout << " cd1 ";
        j = i - k + 3;
    }
    D1(const D1 &ob) { cout << "CC D1"; }
    ~D1() { cout << " dd1 "; }
};

class D2 : public D1 {
    int k;
public:
    D2(int l = 15) {
        cout << " cd2 ";
        k = i - l + 3;
    }
    D2(const D2 &ob) { cout << "CC D2"; }
    ~D2() { cout << " dd2 "; }
};

D1 f(D1 d1, D2 d2) { return d1.get_i() + d2.get_i(); }

int main() {
    D1 ob1(3);
    D2 ob2;
    cout << f(ob1, ob2).get_i() + ob2.get_i();
    return 0;
}
```

Linia `D1 ob1(3);`

Ne ducem la nivelul clasei D1 și observăm că moștenește din clasa B. Astfel, apelăm în ordinea asta:

- `B(j = 5)` => se afișează `cb` și **ob1.i = 3**.
- `D1(k = 3)` => se afișează `cd1` și **ob1.j = 3**.

Linia `D2 ob2;`

Ne ducem la nivelul clasei D2 și observăm că moștenește din clasa D1 care moștenește din B. Astfel, apelăm în ordinea asta:

- `B(j = 5)` => se afișează `cb` și **ob2.i = 3**.
- `D1(k = 10)` => se afișează `cd1` și **ob2.j = -4**.
- `D2(l = 15)` => se afișează `cd2` și **ob2.k = -9**.

Linia `cout << f(ob1, ob2).get_i() + ob2.get_i();`

- `f(ob1, ob2)` => apelăm constructorul de copiere pentru `ob1` și pentru `ob2` și, automat, constructorii parametrizați din clasele de bază => **cb cd1 CC D2 cb CC D1**.
- Când returnăm obiectul de tip D1, se mai apelează constructorul parametrizat pentru D1 => **cb cd1**.

La finalul execuției se apelează destructorii în ordine inversă constructorilor.

Rezultatul întors de funcție este `ob1.i + ob2.i = 6`

Output:

```
cb cd1 cb cd1 cd2 cb cd1 CC D2 CC cb CC D1 cb cd1
6
dd1 db dd1 db dd2 dd1 db dd2 dd1 db db1 db
```

Outputul va fi tot pe aceeași linie, dar l-am separat așa ca să vedem mai clar cum se apelează destructorii fix în ordine inversă.

2

```
#include <iostream>
using namespace std;

class cls
{
    int *v, nr;
    public:
        cls(int i = 0)
        {
            nr = i;
            v = new int[i];
            for (int j = 0; j < size(); j++)
```

```
                v[j] = 3 * j;
        }
        ~cls() { delete[] v; }
        int size() { return nr; }
        int &operator[](int i) { return v[i]; }
        cls operator+(cls);
};

cls cls::operator+(cls y)
{
    cls x(size());
    for (int i = 0; i < size(); i++)
        x[i] = v[i] + y[i];
    return x;
}

int main()
{
    cls x(10), y = x, z;
    x[3] = y[6] = -15;
    z = x + y;
    for (int i = 0; i < x.size(); i++)
        cout << z[i] << " ";
    return 0;
}
```

Rezolvare: Programul compilează. Instrucțiunea `cls x(10)` creează un obiect de tipul clasei `cls`. În momentul instanțierii obiectului, câmpul `nr` al obiectului x primește valoarea `10`, iar pointerul `v` face referire către un tablou unidimensional de `10` elemente ce va fi inițializat cu valorile: `0 3 6 9 12 15 18 21 24 27`.

Apoi observăm că se execută instrucțiunea `y = x`, deci se apelează constructorul de copiere default al clasei `cls`, ceea ce înseamnă că se va crea o copie "shallow" a vectorului `v`. Cu alte cuvinte `x` si `y` vor face referire către acelasi vector `v` (către aceeasi zonă de memorie).

La executarea instrucțiunii `(cls) z` se creează un obiect de tipul clasei `z` cu `nr = 0` și `v` un vector cu nici un element. Ținând cont că data `v` din cadrul obiectelor `x` si `y` fac referire la aceeasi zona de memorie după executarea instrucțiunii `x[3]=y[6]=-15` vectorul `v` va avea valorile: `0 3 6 -15 12 15 -15 21 24 27`.

Problema apare la executarea instrucțiunii `z = x + y`. Observam ca instrucțiunea `cls x(size())` creează un obiect pe stiva. Atunci când se face return dintr-o funcție, destructorii sunt ultimele metode care se apelează. Destructorul pentru variabila `x`, este apelat într-adevar după ce valoarea s-a copiat în valoarea returnată de funcție, dar cum constructorul de copiere/operatorul de atribuire nu a fost rescris se menține o copie "shallow" către vectorul `v` al obiectului `x` care este distrus.

Astfel, programul are un comportament nedefinit și nu putem cunoaște exact ce se va afișa la zona de memorie respectivă.

3

```
#include<iostream>
using namespace std;

class A
{
    int *v;
    int x;
public:
    A(int i=0):v(new int[i]),x(i)
    {
        for(int j=0;j<i;j++)
            v[j] = j;
    }

    int get_x()const{return x;}
    //int& set_x(int i){x=i;}

    A& operator=(A& a)
    {
        x=a.x;
        v=new int[x];
        for(int j=0;j<x;j++)
            v[j]=a[j];
        return a;
    }
     int& operator[](int i)const{return v[i];}
};

ostream& operator<<(ostream& o, const A& a)
{
    for(int i=0;i<a.get_x();i++)
        cout<<a[i]<<" ";
    return o;
}

int main()
{
    A a(12), b;
    cout<<(a=b=a);
    return 0;
}
```

Rezolvare: Programul compilează. Acesta are un comportament nedefinit la afișare, deoarece operatorul de atribuire este scris eronat. Pentru a înțelege ce se întâmplă să luăm execuția programului pas cu pas, mai exact la linia `cout<<(a=b=a)`.

În momentul în care `b = a`, observăm că pointer-ul `v` al obiectului `b` va face referire către pointer-ul `v` al obiectului `a`, pentru că de fapt se realizează tot o copie "shallow" din moment ce întoarcem valoarea lui `a`.

Deci `b` va fi de fapt o referință către `a`. În momentul în care se realizează instrucțiunea `a = b` o puteți gândi ca și cum s-ar executa `a = a`, iar dacă ne uităm în suprascrierea operatorului de atribuire atunci observăm

că se execută instrucțiunea `v = new int[x]`, deci zona de memorie a lui `v` din `a` este realocată. De aici și comportamentul nedefinit.

3

```
#include <iostream>
using namespace std;

template<class T, class U>
T f(T x, U y) { return x + y; }

int f(int x, int y) { return x - y; }

int main() {
    int* a = new int(3), b(2);
    cout << *f(a, b);
    return 0;
}
```

Rezolvare:

Programul compilează. Programul are un rezultat nedefinit, deoarece se afișeazp valoarea de la o zonă de memorie neinițializată.

În momentul executării instrucțiunii `cout << *f(a, b);` se va duce pe template și va aduna o adresă de memorie, la care se adaugă 2 octeți (aritmetica pointerilor).

4

```
class B
{ /* instructiuni */
public:
    B() {cout<<"B"<<" ";}
};
class D1 : virtual B
{ /* instructiuni */
public:
    D1() {cout<<"D1"<<" ";}
};
class D2 : virtual B
{ /* instructiuni */
public:
    D2() {cout<<"D2"<<" ";}
};
class D3 : B
{ /* instructiuni */
public:
    D3() {cout<<"D3"<<" ";}
};
class D4 : private B
```

```
{ /* instructiuni */
public:
    D4() {cout<<"D4"<<" ";}
};
class D5 :virtual public B
{ /* instructiuni */
public:
    D5() {cout<<"D5"<<" ";}
};
class M1 : D1, public D2, D3, private D4,  D5
{ /* instructiuni */
public:
    M1() {cout<<"M1"<<" ";}
};
class M2 :D1,D2,virtual D3, virtual D4, virtual D5
{ /* instructiuni */
public:
    M2() {cout<<"M2"<<" ";}
};
int main(){
    M1 h;
    cout<<endl;
    M2 a;
}
```

Rezolvare (pentru `M2 a`):

În cazul moștenirilor combinate (virtuale și nevirtuale) sunt câteva reguli ce se aplică în stabilirea ordinii apelării:

- Clasele de bază moștenite virtual se construiesc primele;
- Clasele virtuale ce nu sunt de bază se construiesc după;
- În final toate clasele nevirtuale sunt construite.

În codul de mai sus următorii constructori se apelează, în paranteză o să scriu clasa ce apelează fiecare tip de constructor

```
virtual B(D1,D2,D5)      B(D3)    B(D4)

D1(M2)     D2(M2)     virtual D3(M2)    virtual D4(M2)   virtual D5(M2)

                          M2
```

Conform primei reguli prima dată se apelează constructorul virtual al clasei de bază `virtual B`, o să se afișeze `"B"`. Apoi, se vor apela constructorii celorlalte clase virtuale `(D3,D4,D5)` și o să fie construite în mod clasic.

La final o să se apeleze constructorii pentru clasele moștenite nevirtual `(D1 si D2)`, pentru acestea constructorul lui `B` nu o să se mai apeleze, deoarece a fost deja apelat la început (el este constructorul de bază virtual), deci, în final o sa se afiseze:

```
B (constructor de bază virtual)

B (constructorul lui B din D3) D3 (C din D3)

B D4 (C din B din D4 și C lui D4)

D5 (C lui D5, C lui B nu mai este apelat deoarece este moștenit virtual)

D1 D2 (în final constructorii claselor nevirtuale sunt apelați, B nu mai
este apelat deoarece B este moștenit virtual)
```