Exercițiul 1

Fiind date următoarele tipuri de clase spuneţi de câte ori este moştenită clasa B în clasa M1. Dar în clasa M2 ?

```
class B
{ /* instructiuni */
};
class D1 : virtual B
{ /* instructiuni */
};
class D2 : virtual B
{ /* instructiuni */
};
class D3 : B
{ /* instructiuni */
};
class D4 : private B
{ /* instructiuni */
};
class D5 : virtual public B
{ /* instructiuni */
};
class M1 : D1, public D2, D3, private D4, virtual D5
{ /* instructiuni */
};
class M2 : D1, D2, virtual D3, virtual D4, virtual D5
{ /* instructiuni */
};
```

Rezolvare:

Pentru `M1`:

Clasa `B` este moștenită de 3 ori de către clasa `M1`. Observăm că clasele `D1`, `D2` și `D5` moștenesc în mod virtual clasa `B`, astfel ea este moștenită în clasa `M1` o singură dată, iar clasele `D4` și `D5` realizează o moștenire simplă pentru clasa `B`, astfel ea este moștenită de două ori în clasa `M1`.

Pentru `M2`:

Clasa `B` este moștenită de 3 ori de către clasa `M2`. Explicația este aceeași ca cea de mai sus.

Exercițiul 2

Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează, în caz negativ spuneți de ce nu este corect și realizați o modificare astfel încât acesta să compileze fără a-i schimba funcționalitatea.

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

class B
{
protected:
    int x;

public:
    B() { x = 78; }
};

class D1 : virtual public B
{
public:
    D1() { x = 15; }
};
class D2 : virtual public B
{
public:
    D2() { x = 37; }
};
class C : public D2, public D1
{
public:
    int get_x() { return x; }
};
int main()
{
    C ob;
    cout << ob.get_x();
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta afișează valoarea `15`.

Explicație:

Instrucțiunea `C ob;` instanțiază un obiect de tipul clasei `C`. Clasa `C` moștenește clasele `D2`, `D1` (atenție la ordine) în mod `public`. La rândul lor, clasele `D2` și `D1` moștenesc în mod `public virtual` clasa `B`, astfel clasa `C` moștenește o singură dată clasa `B`. Ordinea de execuție a constructorilor este următoarea:

```
B
D2
D1
C
```

Pentru că ultima modificare asupra lui `x` o realizează constructorul clasei `D1` valoarea acestuia devenind `15`.

Exercițiul 3

Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează, în caz negativ spuneți de ce nu este corect și realizați o modificare astfel încât acesta să compileze fără a-i schimba funcționalitatea.

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

class A
{
public:
    int x;
    A(int i = 0) { x = i; }
    virtual A minus() { return (1 - x); }
};

class B : public A
{
public:
    B(int i = 0) { x = i; }
    void afisare() { cout << x; }
};

int main()
{
    A *p1 = new B(18);
    *p1 = p1->minus();
    p1->afisare();
    return 0;
}
```

Rezolvare:

Programul nu compilează. Problema apare din cauza downcasting-ului realizat la crearea obiectului `A *p1 = new B(18);`. Clasa `A` nu conține metoda `afisare` de aceea nu poate fi accesata de pointer-ul `p1` (acesta are vizibilitate doar asupra membrilor din clasa `A`).

O metodă posibilă de rezolvare ar fi să declarăm metoda `afisare` în clasa A ca metodă virtuală:

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

class A
{
public:
    int x;
    A(int i = 0) { x = i; }
    virtual void afisare() {}
    virtual A minus() { return (1 - x); }
```

```
};

class B : public A
{
public:
    B(int i = 0) { x = i; }
    void afisare() { cout << x; }
};

int main()
{
    A *p1 = new B(18);
    *p1 = p1->minus();
    p1->afisare();
    return 0;
}
```

Exercițiul 4

Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează, în caz negativ spuneți de ce nu este corect și realizați o modificare astfel încât acesta să compileze fără a-i schimba funcționalitatea.

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

class A
{
public:
    int x;
    A(int i = 0) { x = i; }
    virtual A minus() { cout << x; }
};

class B : public A
{
public:
    B(int i = 0) { x = i; }
    void afisare() { cout << x; }
};
int main()

{
    A *p1 = new A(18);
    *p1 = p1->minus();
    dynamic_cast<B *>(p1)->afisare();
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta nu afișează nimic pentru că va da eroare la rulare:

```
anpopescu@ANPOPESCU-L:~/Documents/Tutoriate$ g++ -std=c++17 main.cpp -o
main.out
anpopescu@ANPOPESCU-L:~/Documents/Tutoriate$ ./main.out
Segmentation fault (core dumped)
```

Explicație:

Operatorul `dynamic_cast` întoarce `NULL` dacă conversia nu se poate realiza cu succes. Downcasting-ul care se încearcă a se face în program nu se realizează cu succes pentru ca pointer-ul `p1` face referire la un obiect de tipul clasei `A`. Astfel se încearcă a se realiza un apel de forma: `NULL -> afisare()`. De aici și segmentation fault.

(La genul acesta de probleme nu trebuie să faceți programul să funcționeze. Este suficient doar să compileze.)

Exercițiul 5

Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează, în caz negativ spuneți de ce nu este corect și realizați o modificare astfel încât acesta să compileze fără a-i schimba funcționalitatea.

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

class A
{
public:
    int x;
    A(int i = 13) { x = i; }
};

class B : virtual public A
{
public:
    B(int i = 15) { x = i; }
};

class C : virtual public A
{
public:
    C(int i = 17) { x = i; }
};

class D : public A
{
public:
    D(int i = 19)
    {
```

```
        x = i;
    }
};

class E : public B, public C, public D
{
public:
    int y;
    E(int i, int j) : D(i), B(j)
    {
        y = x + i + j;
    }
    E(E &e) { y = -e.y; }
};

int main()
{
    E e1(-9.3), e2 = e1;
    cout << e2.y;
    return 0;
}
```

Rezolvare:

Programul nu compilează. Programul nu compilează deoarece clasa `E` nu conține un constructor cu un singur parametru de tip întreg: `E(int i, int j)`. Chiar dacă adaugăm un parametru cu valoare implicită în constructorul clasei `E`, programul tot nu va compila deoarece clasa `D` moștenește în mod nevirtual clasa `A` și astfel apare o problemă de ambiguitate (Problema diamantului).

O posibilă rezolvare ar fi să adaugăm un parametru implicit constructorului clasei `E` și să schimbăm tipul de moștenire al clasei `D`:

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

class A
{
public:
    int x;
    A(int i = 13) { x = i; }
};

class B : virtual public A
{
public:
    B(int i = 15) { x = i; }
};

class C : virtual public A
{
```

```
public:
    C(int i = 17) { x = i; }
};

class D : virtual public A
{
public:
    D(int i = 19)
    {

        x = i;
    }
};

class E : public B, public C, public D
{
public:
    int y;
    E(int i, int j = 0) : D(i), B(j)
    {
        y = x + i + j;
    }
    E(E &e) { y = -e.y; }
};

int main()
{
    E e1(-9.3), e2 = e1;
    cout << e2.y;
    return 0;
}
```