Exercițiul 1

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 Test1
{
    int x;

public:
    void show() {}
};
class Test2
{
    int x;

public:
    virtual void show() {}
};
int main(void)
{
    cout << sizeof(Test1) << endl;
    cout << sizeof(Test2) << endl;
    return 0;
}
```

Rezolvare:

Programul compilează. Rezultatele afișate de acesta depinde de arhitectura folosită de calculator. Dacă procesorul este pe `32` de biți atunci va afișa: `4` și `8`, iar dacă acesta este pe `64` de biți atunci va afișa: `4` și `16`. Există o singură diferență între `Test1` și `Test2`. `show` este o metodă non-virtuală în `Test1`, ci virtuală în `Test2`. Când facem o metoda virtuală, compilatorul adaugă un pointer suplimentar numit `vptr`. Compilatorul face acest lucru pentru a atinge polimorfismul la executie. `vptr`-ul este un pointer suplimentar care se adaugă la dimensiunea clasei, de aceea obținem `8` ca dimensiune a clasei `Test2` dacă procesorul este pe `32` de biți și `16` dacă procesorul este pe `64` de biți (normal dimensiunea variabilei `x` este de `4` biți, iar cea a `vptr`-ului este de `8` biți pe o arhitectura de `64` de biți, dar se face o anumita "aliniere" de care credem ca s-a vorbit și la cursul de ASC). Pentru cei care sunt curioși să vadă cum se face alocarea în memorie în acest exercițiu puteți merge pe site-ul: https://gcc.godbolt.org/ . Puneți codul vostru și uitați-vă cum se realizează alocarea în asamblare pentru un procesor pe `64` de biți.

Exercițiul 2

Spuneți dacă programul de mai jos este corect. 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 P
{
public:
    virtual void show() = 0;
};
class Q : public P
{
    int x;
};
int main(void)
{
    Q q;
    return 0;
}
```

Rezolvare:

Programul nu compilează. Acesta nu trece de compilare deoarece metoda `show` din clasa `P` este declarată ca find pur virtuală. Pentru o metodă pur virtuală, toate clasele care moștenesc clasa ce conține metoda respectivă trebuie să o și implementeze.

O soluție simplă pentru a trece de compilare este sa declarăm funcția ca find non-virtuală:

```
#include <iostream>
using namespace std;
class P
{
public:
    void show();
};
class Q : public P
{
    int x;
};
int main(void)
{
    Q q;
    return 0;
}
```

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:
    virtual void fun() { cout << "A" << endl; }
};
class B : public A
{
public:
    virtual void fun() { cout << "B" << endl; }
};
class C : public B
{
public:
    virtual void fun() { cout << "C" << endl; }
};
int main()
{
    A *a = new C;
    A *b = new B;
    a->fun();
    b->fun();
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta afișează literele `C` și `B`. Observăm că clasa `A` conține o metodă virtuală `fun()`, metodă care e suprascrisă (se realizează procedeul de overriding) în clasele care o moștenesc (`B` și `C`). În funcția `main` se crează o instanță a clasei `C` și o instanță a clasei `B` cărora li se aplică procedeul de `upcasting`. Totuși, ținând cont că metoda este virtuală și suprascrisă în ambele clase, acesta va aplica comportamentele metodelor definite în clasele `B` și `C`.

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
{
protected:
    int x;

public:
    A(int i = -16) { x = i; }
    virtual A f(A a) { return x + a.x; }
```

```
    void afisare() { cout << x; }
};

class B : public A

{
public:
    B(int i = 3) : A(i) {}
    A f(A a) { return x + a.x + 1; }
};

int main()
{
    A *p1 = new B, *p2 = new A, *p3 = new A(p1->f(*p2));
    p3->afisare();
    return 0;
}
```

Rezolvare:

Programul nu compilează. Acesta nu trece de compilare din cauza instrucțiunii `x + a.x + 1;` în care se încearcă accesarea unui membru `protected` și anume `a.x`. Chiar dacă clasa `B` moștenește clasa `A` în mod `public`, iar câmpul `x` rămâne astfel disponibil în clasa `B`, acesta nu poate fi accesat în mod direct dintr-o instanță a clasei `A` așa cum se întâmplă în suprascrierea metodei `f` din clasa `B`.

O soluție posibilă care să nu schimbe comportamentul programului ar fi să schimbăm modificatorul de acces al lui `x` din clasa `A` din `protected` în `public`:

```
#include <iostream>
using namespace std;
class A
{
public:
    int x;

public:
    A(int i = -16) { x = i; }
    virtual A f(A a) { return x + a.x; }
    void afisare() { cout << x; }
};

class B : public A

{
public:
    B(int i = 3) : A(i) {}
    A f(A a) { return x + a.x + 1; }
};

int main()
{
```

```
    A *p1 = new B, *p2 = new A, *p3 = new A(p1->f(*p2));
    p3->afisare();
    return 0;
}
```

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>
#include <stdio.h>
using namespace std;
class Base
{
public:
    Base()
    {
        fun();
    }
    virtual void fun()
    {
        cout << "\nBase Function";
    }
};

class Derived : public Base
{
public:
    Derived() {}
    virtual void fun()
    {
        cout << "\nDerived Function";
    }
};

int main()
{
    Base *pBase = new Derived();
    delete pBase;
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta afișează: `\nBase Function`.

Când o metodă este apelată într-un constructor/destructor și obiectul din care apelul este făcut este obiectul construit/distrus, atunci metoda apelată va fi mereu metoda ce se află în acea clasă sau într-o clasă

din care obiectul derivă, niciodată în o clasa pentru care este baza, indiferent de tipul metodei.

Exercițiul 6

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 base
{
public:
    virtual void show() { cout << " In Base \n"; }
};

class derived : public base
{
    int x;

public:
    void show() { cout << "In derived \n"; }
    derived() { x = 10; }
    int getX() const { return x; }
};

int main()
{
    derived d;
    base *bp = &d;
    bp->show();
    cout << bp->getX();
    return 0;
}
```

Rezolvare:

Programul nu compilează. Deoarece pointer-ul `bp` este de tipul `base` și nu are declarată sau implementată metoda `getX()` aceasta nu poate apela metoda din clasa `derived`. Când un pointer al clasei de bază pointează spre o clasă derivată acesta va putea apela numai metodele clasei derivate ce sunt prezente și virtuale în clasa de bază.

O posibilă soluția ar putea fi să declarăm metoda `getX()` și în clasa `base` astfel:

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

class base
{
```

```
public:
    virtual void show() { cout << " In Base \n"; }
    int getX() const {};
};

class derived : public base
{
    int x;

public:
    void show() { cout << "In derived \n"; }
    derived() { x = 10; }
    int getX() const { return x; }
};

int main()
{
    derived d;
    base *bp = &d;
    bp->show();
    cout << bp->getX();
    return 0;
}
```

Exercițiul 7

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
{
protected:
    int x;

public:
    A(int i = -16)
    {
        x = i;
    }
    virtual A f(A a) { return x + a.x; }
    void afisare() { cout << x; }
};

class B : public A
{
public:
    B(int i = 3) : A(i) {}
```

```
    B f(B b)
    {
        return x + b.x + 1;
    }
};

int main()
{
    A *p1 = new B, *p2 = new A, *p3 = new A(p1->f(*p2));
    p3->afisare();
    return 0;
}
```

Rezolvare:

Programul compilează. Când am postat exercițiile am introdus o eroare din greșeală și anume la declarația pointer-ul `p1` noi l-am declarat cu numele `pl`. Acesta afișează valoarea `-13`. Pentru explicații vedeți exercțiul `12`. Este aproximativ la fel.

Exercițiul 8

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
{
protected:
    int x;

public:
    A(int i = -31) { x = i; }
    virtual A operator+(A a) { return x + a.x; }
};

class B : public A
{
public:
    B(int i = 12) { x = i; }
    B operator+(B b) { return x + b.x + 1; }
    void afisare() { cout << x; }
};

int main()
{
    A *p1 = new B, *p2 = new A;
    B *p3 = new A(p2->operator+(*p1));
    p3->afisare();
```

```
    return 0;
}
```

Rezolvare:

Programul nu compilează. Acesta nu trece de compilare din cauza instrucțiunii `B *p3 = new A(p2->operator+(*p1));` deoarece se încearcă un downcasting nepermis (de fapt conversia aceasta nu se poate realiza în mod implicit, dar se poate forța realizând-o în mod explicit `B *p3 = (B*) new A(p2->operator+(*p1));`).

Soluție:

```
#include <iostream>
using namespace std;
class A
{
protected:
    int x;

public:
    A(int i = -31) { x = i; }
    virtual A operator+(A a) { return x + a.x; }
};

class B : public A
{
public:
    B(int i = 12) { x = i; }
    B operator+(B b) { return x + b.x + 1; }
    void afisare() { cout << x; }
};

int main()
{
    A *p1 = new B, *p2 = new A;
    B *p3 = (B*) new A(p2->operator+(*p1));
    p3->afisare();
    return 0;
}
```

Exercițiul 9

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
```

```
{
    int i;

public:
    B() { i = 1; }
    virtual int get_i() { return i; }
};
class D : public B
{
    int j;

public:
    D() { j = 2; }
    int get_i() { return B::get_i() + j; }
};
int main()
{
    const int i = cin.get();
    if (i % 3)
    {
        D o;
    }
    else
    {
        B o;
    }
    cout << o.get_i();
    return 0;
}
```

Rezolvare:

Programul nu compilează. Acesta nu trece de compilare deoarece obiectul `o` este creat în alt `scop`, diferit de cel al funcției `main`, indiferent de valoarea citită de la tastură. Astfel obiectul `o` este dezalocat înaintea de execuția instrucțiunii `cout << o.get_i();`.

O posibilă soluție ar fi să declarăm pointer-ul în afara clasei și să aplicăm procedeul de upcasting:

```
#include <iostream>
using namespace std;
class B
{
    int i;

public:
    B() { i = 1; }
    virtual int get_i() { return i; }
};
class D : public B
{
    int j;
```

```
public:
    D() { j = 2; }
    int get_i() { return B::get_i() + j; }
};
int main()
{
    const int i = cin.get();
    B *o;
    if (i % 3)
    {
        o = new D();
    }
    else
    {
        o = new B();
    }
    cout << o->get_i();
    return 0;
}
```

Exercițiul 10

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
{
    int i;

public:
    B() { i = 1; }
    virtual int get_i() { return i; }
};
class D : virtual public B
{
    int j;

public:
    D() { j = 2; }
    int get_i() { return B::get_i() + j; }
};
class D2 : virtual public B
{
    int j2;

public:
    D2() { j2 = 3; }
```

```
    int get_i() { return B::get_i() + j2; }
};
class MM : public D, public D2
{
    int x;

public:
    MM() { x = D::get_i() + D2::get_i(); }
    int get_i() { return x; }
};
int main()
{
    B *o = new MM();
    cout << o->get_i() << "\n";
    MM *p = dynamic_cast<MM *>(o);
    if (p)
        cout << p->get_i() << "\n";
    D *p2 = dynamic_cast<D *>(o);
    if (p2)
        cout << p2->get_i() << "\n";
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta afișează valorile: 7, 7, 7. Să urmărim execuția pas cu pas. Mai întâi se crează un obiect de tipul clasei `MM` asupra căruia se aplică procedeul de upcasting, având un pointer de tipul clasei `B` care pointează către acest obiect. În timpul creări obiectului la care pointeză pointerul `o` este apelat constructorul clasei `B` care își inițializează câmpul `i` cu valoarea `1`, apoi constructorul clasei D care își inițializează câmpul `j` cu valoarea `2` și apoi constructorul clasei D2 care își inițializează câmpul `j2` cu valoarea `3`, iar în ultimul rând este apelat și constructorul clasei `MM` care observăm că apelează în mod explicit metodele `get_i()` din clasele `D` și `D2` și face suma rezultatelor, metode care și ele la rândul lor apelează metoda `get_i()` în mod explicit din clasa B și la care adună `j` și `j2`. Deci pentru `D1` avem 1 + 2 = 3 și pentru `D2` avem 1 + 3 = 4, iar suma lor este 7, de aici și primul 7. Apoi se execută instrucțiunea `cout << o->get_i() << "\n";`. Metoda `get_i()` este o metodă virtuală și este suprascrisă în toate clasele care o moștenesc, deci se va apela metoda `get_i()` din clasa MM și va întoarce valoarea 7. Apoi instrucțiunea `MM *p = dynamic_cast<MM *>(o);` realizează procedeul de downcasting cu succes și astfel se afișează din nou 7. De asemenea tot un procedeul de downcasting este realizat și de instrucțiunea `D *p2 = dynamic_cast<D *>(o);` care de asemenea se execută cu succes pentru că downcasting-ul se poate realiza pe întregul lanț ierarhic și se afișează din nou valoarea 7.

Exercițiul 11

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(int i = 28) { x = i; }
    virtual B f(B ob) { return x + ob.x + 1; }
    void afisare() { cout << x; }
};
class D : public B
{
public:
    D(int i = -32) : B(i) {}
    B f(B ob) { return x + ob.x - 1; }
};
int main()
{
    B *p1 = new D, *p2 = new B, *p3 = new B(p1->f(*p2));
    p3->afisare();
    return 0;
}
```

Rezolvare:

Programul nu compilează. Vezi rezolvarea de la exercițiul 4. (Este exact aceeași eroare).

O posibilă rezolvare presupune schimbarea modificatorului de acces din clasa `B` al membrului `x` din `protected` în `public`:

```
#include <iostream>
using namespace std;
class B
{
public:
    int x;

public:
    B(int i = 28) { x = i; }
    virtual B f(B ob) { return x + ob.x + 1; }
    void afisare() { cout << x; }
};
class D : public B
{
public:
    D(int i = -32) : B(i) {}
    B f(B ob) { return x + ob.x - 1; }
};
int main()
{
    B *p1 = new D, *p2 = new B, *p3 = new B(p1->f(*p2));
    p3->afisare();
```

```
    return 0;
}
```

Exercițiul 12

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(int i = 12) { x = i; }
    virtual B f(B ob) { return x + ob.x + 1; }
    void afisare() { cout << x; }
};
class D : public B
{
public:
    D(int i = -15) : B(i - 1) { x++; }
    B f(B ob) { return x - 2; }
};
int main()
{
    B *p1 = new D, *p2 = new B, *p3 = new B(p1->f(*p2));
    p3->afisare();
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta afișează valoarea `-17`. Pentru a înțelege de unde se afișează această valoarea să luăm programul să îl executăm pas cu pas. Instrucțiunea `B *p1 = new D` crează un obiect de tipul clasei `D` căruia i se aplică procedeul de `upcasting`. Clasa `D` moștenește în mod public clasa `B`. De asemenea observăm că atât clasa `D`, cât și clasa `B` au un constructor cu un parametru implicit. Astfel în timpul creări obiectului spre care pointează `p1` este apelat constructorul clasei `B` cu parametrul `-16`, iar câmpul de date `x` al clasei `B` primește valoarea `-16`, iar în constructorul clasei `D` acest câmp de date este incrementat cu `1`, deci valoarea lui `x` va rămâne `-15` pentru acest obiect. Următoarea instrucțiune `*p2 = new B` instanțiază un obiect de tipul clasei `B`, astfel este apelat constructorul cu parametrul implicit `12`, iar câmpul de date `x` al obiectului respectiv devine `12`. Ultima instrucțiune `*p3 = new B(p1->f(*p2))` crează de asemenea un obiect de tipul clasei `B`, dar este interesant parametrul furnizat constructorului. Analizând parametrul ne dăm seama că acesta reprezintă un apel al funcției `virtuale f` ce aparține obiectului la care pointează `p1`. Funcția find virtuală, iar obiectul find creat ca instanță a clasei `D`, se va apela

functia `f` ce aparține scopului clasei `D`, deci se va returna un obiect creat automat (pentru că după cum observați se întoarce un întreg, iar constructorul clasei `B` primește de asemenea un întreg) cu valoarea `-17` (-15 (x-ul lui p1) - 2). De aici valoarea `-17`.

Exercițiul 13

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
{
protected:
    static int x;

public:
    A(int i = 0) { x = i; }
    virtual A schimb() { return (7 - x); }
};
class B : public A
{
public:
    B(int i = 0) { x = i; }
    void afisare() { cout << x; }
};
int A::x = 5;
int main()
{
    A *p1 = new B(18);
    *p1 = p1->schimb();
    ((B *)p1)->afisare();
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta afișează valoarea `-11`. Instrucțiunea `A *p1 = new B(18)` crează un obiect de tipul clasei `B` aplicând procedeul de `upcasting`. Clasa `B` moștenește clasa `A` în mod public. Observăm că atât clasa `A`, cât și clasa `B` au câte un constructor cu parametru implicit. În instanțierea obiectului nostru vedem că constructorul clasei `B` este apelat cu un parametru explicit și anume `18`. În timpul creării obiectului este apelat mai întâi constructorul clasei `A` care schimbă valoarea câmpului static `x` (inițial `5`) în `0`. Apoi în corpul constructorului clasei `B` este schimbată din nou valoarea lui `x` în `18`. Următoarea instrucțiune `*p1 = p1->schimb()` apelează funcția `virtuală schimb` care crează un obiect de tipul `A` (atenție, funcția nu este suprascrisă în clasa derivată deci comportamentul rămâne cel din clasa de baza). Deoarece constructorul clasei `A` conține un parametru de tip întreg, iar funcția `A` returnează un întreg `(7 - x)`, de fapt se va crea un obiect de tipul clasei `A` în mod automat. Astfel, `p1` va pointa după execuția acestei instrucțiuni către un nou obiect, iar `x`-ul devine `-11`. De aici valoarea `-11`.

Exercițiul 14

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;
    B(int i = 10) { x = i; }

public:
    virtual B operator+(B ob)
    {
        B y(x + ob.x);
        return y;
    }
};
class D : public B
{
public:
    D(int i = 10) { x = i; }
    void operator=(B p) { x = p.x; }
    B operator+(B ob)
    {
        B y(x + ob.x + 1);
        return y;
    }
    void afisare() { cout << x; }
};

int main()
{
    D p1(-59), p2(32), *p3 = new D;
    *p3 = p1 + p2;
    p3->afisare();
    return 0;
}
```

Rezolvare:

Programul nu compilează. Una dintre erori apare la instrucțiunea `void operator=(B p) { x = p.x; }` care este aproximativ la fel cu cele menționate la exercițiile 4 și 11.

Exercițiul 15

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
{
public:
    int x;
    B(int i = 0) { x = i; }

    virtual B aduna(B ob) { return (x + ob.x); }
    B minus() { return (1 - x); }
    void afisare() { cout << x; }
};

class D : public B
{
public:
    D(int i = 0) { x = i; }

    B aduna(B ob) { return (x + ob.x + 1); }
};

int main()
{
    B *p1, *p2;
    p1 = new D(138);
    p2 = new B(-37);
    *p2 = p2->aduna(*p1);
    *p1 = p2->minus();
    p1->afisare();
    return 0;
}
```

Rezolvare:

Programul compilează. Acesta afișează valoarea `-100`. Observăm că clasa `D` moștenește clasa `B` în mod public și atât clasa `B` cât și clasa `D` au câte un constructor cu un parametru implicit și ambi realizează aceași acțiune; câmpului `x` îi este atribuită valoarea lui `i`. Instrucțiunea `p1 = new D(138);` crează un obiect de tipul clasei `D` căruia i se aplică procedeul de upcasting, iar `x`-ul obiectului respectiv devine `138`. Instrucțiunea `p2 = new B(-37);` crează un obiect de tipul clasei `B`, iar `x`-ul obiectului devine `-37`. Clasa `B` conține funcția virtuală `adună` care este suprascrisă în clasa `D`. Apelând funcția aduna prin intermediul pointer-ului `p2` care face referire la un obiect de tipul clasei B se crează un nou obiect de tipul clasei `B` al cărui `x` va fi `-37 + 138` deci `101`. De asemenea apelul funcției `minuns` folosind pointer-ul `p2` crează un obiect nou de tipul clasei `B`, al cărui `x` devine `1 - 101` deci `-100`. De aici valoarea `-100`.