# Varianta 1

1 Specificați si testați funcția: **(1.5p)**

```
int f(int x) {
	if (x <= 0)
		throw std::exception("Invalid argument!");

	int rez = 0;
	while (x)
	{
		rez = rez * 10 + x % 10;
		x /= 10;
	}
	return rez;
}
```

2 Indicați rezultatul execuției pentru următoarele programe c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

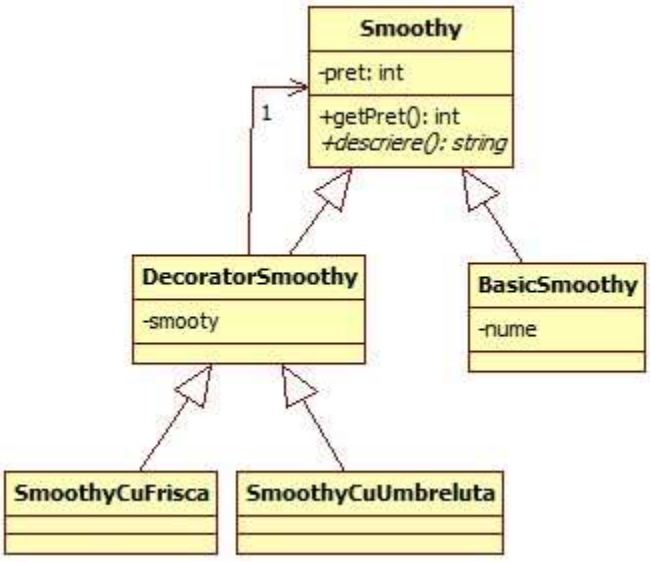
```
//2 a  (1p)
#include <iostream>
using namespace std;
int except(bool thrEx) {
	if (thrEx) {
		throw 2;
	}
	return 3;
}

int main() {
	try {
		cout << except(1 < 1);
		cout << except(true);
		cout << except(false);
	} catch (int ex) {
		cout << ex;
	}
	cout << 4;
	return 0;
}
```

```
//2 b  (0.5p)
#include <iostream>
using namespace std;
class A {
public:
  A() {cout << "A" << endl;}
  ~A() {cout << "~A" << endl;}
  void print() {
     cout << "print" << endl;
  }
};

void f() {
	A a[2];
	a[1].print();
}
int main() {
	f();
	return 0;
}
```

3 Scrieți codul C++ ce corespunde diagramei de clase UML. **(4p)**




- Clasa abstracta **Smoothy** are o metodă pur virtuală descriere(). **DecoratorSmoothy** conține un smoothy, metodele descriere() si getPret() returnează descrierea si pretul smoothy-ului agregat.
- Clasele **SmoothyCuFrisca** si **SmoothyCuUmbreluta** adaugă textul "cu frisca" respectiv "cu umbreluta" la descrierea smoothy-ului conținut. Prețul unui smoothy care are frișca creste cu 2 Ron, cel cu umbreluța costa in plus 3 RON.
- Clasa **BasicSmoothy** reprezintă un smoothy fără frișcă si fără umbreluța, metoda descriere() returnează denumirea smothy-ului.

Se cere:

1 Codul C++ **doar pentru clasele**: **Smoothy**,**DecoratorSmoothy**, **SmoothyCuFrisca (0.75)**

2 Scrieți o funcție C++ care returnează o lista de smothy-uri: un smoothy de kivi cu frișcă si umbreluță, un smoothy de căpșuni cu frișcă si un smoothy simplu de kivi. **(0.5p)**

3 Programul principal apelează funcția descrisa mai sus, apoi tipărește descrierea si prețul pentru fiecare smoothy in ordine alfabetica după descriere. **(0.25p)**

- Creați doar metode si atribute care rezultă din diagrama UML (adăugați doar lucruri specifice C++ ex: constructori). Nu adăugați câmpuri, metode, nu schimbați vizibilitatea, nu folosiți friend. Folosiți STL unde exista posibilitatea.
- Detalii barem: **1.5p** Polimorfism, **1p** Gestiunea memoriei, **1.5p** Restul(defalcat mai sus)

**4** Definiți clasa Geanta astfel încât următoarea secvență C++ sa fie corecta sintactic si sa efectueze ceea ce indica comentariile. **(2p)**

```
void calatorie() {
	Geanta<string> ganta{ "Ion" };//creaza geanta pentru Ion
	ganta = ganta + string{ "haine" }; //adauga obiect in ganta
	ganta + string{ "pahar" };
	for (auto o : ganta) {//itereaza obiectele din geanta
		cout << o << "\n";
	}
}
```

1)

```
#include <iostream>
#include <assert.h>

using namespace std;

// Rolul acestei functii este de a calcula rasturnatul unui numar
// preconditii x <=0, altfel programul o sa arunce exceptii
// param de intrare: x - intreg
// param de iesire: rez - intreg
// postconditii: rez sa fie inversul lui x
int f(int x) {
if (x <= 0)
throw std::exception("Invalid argument!");
int rez = 0;
while (x)
{
rez = rez * 10 + x % 10;
x /= 10;
}
return rez;
}

void teste() {

assert(f(12) == 21);
assert(f(123) == 321);
try {
f(0);
```

```
assert(false);
}
catch (exception) {
assert(true);
}
try {
f(-2);
assert(false);
}
catch (exception) {
assert(true);
}
}

int main() {

teste();
return 0;
}
```

2)

```
/*
#include <iostream>
using namespace std;
int except(bool thrEx) {
if (thrEx) {
throw 2;
}
return 3;
```

```
}
int main() {
try {
cout << except(1 < 1);
cout << except(true);
cout << except(false);
}
catch (int ex) {
cout << ex;
}
cout << 4;
return 0;
}
// Rezultatul este 324, deoarece al 2 lea except arunca exceptie si rezulta valoarea 2
// iar partea de mai jos nu se executa intrand pe catch
*/
/*
#include <iostream>
using namespace std;
class A {
public:
A() { cout << "A" << endl; }
~A() { cout << "~A" << endl; }
void print() {
cout << "print" << endl;
}
};
void f() {
A a[2];
```

```
a[1].print();
}
int main() {
f();
return 0;
}*/
// rezultatul este
//          A
//          A
//          print
// ~A
// ~A
```

3)

```
#include <iostream>
#include <vector>
#include <string>
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>

using namespace std;

class Smoothy {

private:

int pret;
```

```
public:

Smoothy(int p) : pret{ p } {};

virtual int getPret() {

return this->pret;
}

virtual string descriere() = 0;

virtual ~Smoothy() = default;
};

class BasicSmoothy : public Smoothy {

private:

string nume;

public:

BasicSmoothy(int pret, string n) : Smoothy(pret), nume{ n }{};

string descriere() override {

return nume;
}
};
```

```
class DecoratorSmoothy : public Smoothy {

private:

Smoothy* s;

public:

DecoratorSmoothy(Smoothy* s0) : Smoothy(s0->getPret()), s{ s0 } {};

virtual string descriere() override{

return s->descriere();
};

virtual int getPret() override{

return s->getPret();
}

virtual ~DecoratorSmoothy() {

delete s;
}
};

class SmoothyCuFrisca : public DecoratorSmoothy {
```

```
public:

SmoothyCuFrisca(Smoothy* s) : DecoratorSmoothy(s) {};

string descriere() override {

return DecoratorSmoothy::descriere() + " cu frisca";
};

int getPret() override {

return DecoratorSmoothy::getPret() + 2;
}

};

class SmoothyCuUmbreluta : public DecoratorSmoothy {

public:

SmoothyCuUmbreluta(Smoothy* s) : DecoratorSmoothy(s) {};

string descriere() override {

return DecoratorSmoothy::descriere() + " cu umbreluta";
};

int getPret() override {
```

```
return DecoratorSmoothy::getPret() + 3;
}
};

vector<Smoothy*> functie2() {

vector<Smoothy*> rez;

rez.push_back(new SmoothyCuUmbreluta(new SmoothyCuFrisca(new BasicSmoothy{ 1, "kivi" })));

rez.push_back(new SmoothyCuFrisca(new BasicSmoothy{ 2, "capsuni" }));

rez.push_back(new BasicSmoothy{ 3, "kivi" });

return rez;
}

int main() {

{
vector<Smoothy*> rez = functie2();

for (int i = 0; i < rez.size() - 1; i++)

for (int j = i + 1; j < rez.size(); j++)

if (rez[i]->descriere() > rez[j]->descriere())

swap(rez[i], rez[j]);
```

```
for (auto s : rez) {

cout << s->descriere() << " " << s->getPret() << '\n';
delete s;
}
}

_CrtDumpMemoryLeaks();

return 0;
}
```

4)

```
/*
#include <iostream>
#include <string>
#include <vector>

using std::string;
using std::vector;
using std::cout;

template<typename TElem>
class Geanta {

private:
```

```
string owner;
vector<TElem> obiecte;

public:

Geanta(string nume) : owner{ nume } {};

Geanta& operator+ (const TElem& obj) {

obiecte.push_back(obj);
return *this;
};

typename vector<TElem>::iterator begin() {

return obiecte.begin();
}

typename vector<TElem>::iterator end() {

return obiecte.end();
}

};

void calatorie() {
Geanta<string> ganta{ "Ion" };//creaza geanta pentru Ion
ganta = ganta + string{ "haine" }; //adauga obiect in ganta
ganta + string{ "pahar" };
```

```
for (auto o : ganta) {//itereaza obiectele din geanta
cout << o << "\n";
}
}

int main() {

calatorie();
return 0;
}*/
```

# Varianta 2

**1** Specificați si testați funcția: (**1.5p**)

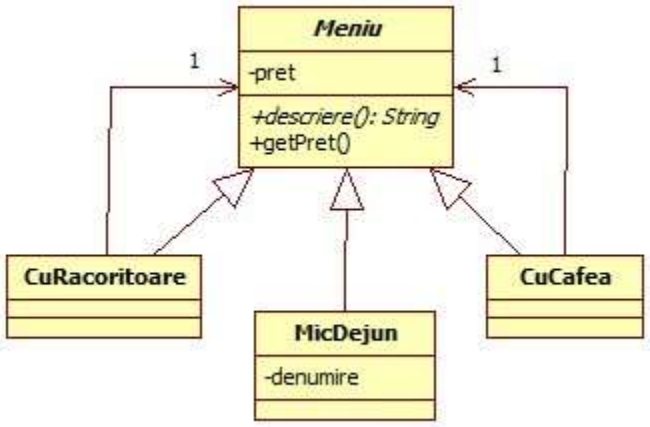
```
std::pair<int, int> f(std::vector<int> l) {
	if (l.size()<2) throw std::exception{};
	std::pair<int, int> rez{-1,-1};
	for (auto el:l) {
		if (el < rez.second) continue;
		if (rez.first < el) {
			rez.second = rez.first;
			rez.first = el;
		}else{
			rez.second=el;
		}
	}
	return rez;
}
```

**2** Indicați rezultatul execuției pentru următoarele programe c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

```
//2 a  (1p)
#include <iostream>
#include <vector>
struct A {
	A() {std::cout << "A";}
	virtual void print() {
		std::cout << "A";
	}
};
struct B : public A{
	B() { std::cout << "B"; }
	void print() override {
		std::cout << "B";
	}
};
int main() {
	std::vector<A> v;
	v.push_back(A{});
	v.push_back(B{});
	for (auto& el : v) el.print();
	return 0;
}
```

```
//2 b  (0.5p)
#include <iostream>
using namespace std;
class A {
	int x;
public:
	A(int x) : x{ x } {}
	 void print(){cout<< x <<endl;}
};
A f(A a) {
	a.print();
	a = A{ 10 };
	a.print();
	return a;
}
int main() {
	A a{ 4 };
	a.print();
	f(a);
	a.print();
}
```

**3** Scrieți codul C++ ce corespunde diagramei de clase UML. (**4p**)




- Clasa abstracta **Meniu** are o metoda pur virtuala descriere()
- **CuRacoritoare** si **CuCafea** conțin un meniu si metoda descriere() adaugă textul "cu racoritoare" respectiv "cu cafea" la descrierea meniului conținut. Prețul unui meniu care conține răcoritoare creste cu 4 Ron, cel cu cafea costa in plus 5 RON.
- Clasa **MicDejun** reprezintă un meniu fără răcoritoare si fără cafea, metoda descriere() returnează denumirea meniului. In restaurant pizzerie exista 2 feluri de mic dejun: Ochiuri si Omleta, la prețul de 10 respectiv 15 RON.

Se cere:

1 Codul C++ **doar pentru clasele**: **Meniu**, **CuCafea (0.75)**
2 Scrieți o funcție C++ care returnează o lista de meniuri: un meniu cu Omleta cu răcoritoare si cafea, un meniu cu Ochiuri si cafea, un meniu cu Omleta. **(0.5p)**
3 In programul principal se creează o comanda (folosind funcția descrisa mai sus), apoi se tipărește descrierea si prețul pentru fiecare pizza in ordinea descrescătoare a preturilor. **(0.25p)**

- Creați doar metode si atribute care rezultă din diagrama UML (adăugați doar lucruri specifice C++ ex: constructori). Nu adăugați câmpuri, metode, nu schimbați vizibilitatea, nu folosiți friend. Folosiți STL unde există posibilitatea.

Detalii barem: **1.5p** Polimorfism, **1p** Gestiunea memoriei, **1.5p** Restul(defalcat mai sus)

**4** Definiți clasa Measurement astfel încât următoarea secvență C++ sa fie corecta sintactic si sa efectueze ceea ce indica comentariile. (**2p**)

```
int main() {
      //creaza un vector de masuratori cu valorile (10,2,3)
      std::vector<Measurement<int>> v{ 10,2,3 };
      v[2] + 3 + 2; //aduna la masuratoarea 3 valuarea 5
      std::sort(v.begin(), v.end());  //sorteaza masuratorile
      //tipareste masuratorile (in acest caz: 2,8,10)
      for (const auto& m : v) std::cout << m.value() << ",";
      return 0;
}
```

1)

```
/*
#include <iostream>
#include <utility>
#include <vector>
#include <assert.h>

// Rolul acestei functii este de a returna cele mai mari
// doua elemente din vectorul trimis ca si parametru
// date de intrare: l - vector de intregi
// date de iesire: retunreaza o pereche de intregi, cele 2 numere
// fiind cele mai mari 2 numere din vector
// @ functia poate arunca exceptii in cazul in care vectorul are
// lungimea mai mica decat 2
// Daca vectorul trimis ca si parametru contine doar elemente negative
// aceasta functie o sa returneze -1, -1
std::pair<int, int> f(std::vector<int> l) {
if (l.size() < 2) throw std::exception{};
std::pair<int, int> rez{ -1,-1 };
for (auto el : l) {
if (el < rez.second) continue;
if (rez.first < el) {
rez.second = rez.first;
rez.first = el;
}
else {
rez.second = el;
}
}
```

```
return rez;
}

void teste() {

std::vector<int> li;

try {
f(li);
assert(false);
}
catch (std::exception) {
assert(true);
}

std::vector<int> l{ 1, 3, 5, 7, 9 };

std::pair<int, int> p = f(l);

assert(p.first == 9);

assert(p.second == 7);
}

int main() {

teste();
return 0;
}*/
```

2)

```
/*
#include <iostream>
#include <vector>
struct A {
A() { std::cout << "A"; }
virtual void print() {
std::cout << "A";
}
};
struct B : public A {
B() { std::cout << "B"; }
void print() override {
std::cout << "B";
}
};
int main() {
std::vector<A> v;
v.push_back(A{});
v.push_back(B{});
for (auto& el : v) el.print();
return 0;
}*/
// Rezultatul este: AABAA
```

//2 b (0.5p)

```
/*
#include <iostream>
using namespace std;
class A {
int x;
public:
A(int x) : x{ x } {}
void print() { cout << x << endl; }
};
A f(A a) {
a.print();
a = A{ 10 };
a.print();
return a;
}
int main() {
A a{ 4 };
a.print();
f(a);
a.print();
}*/
// Rezultatul este:
//          4
//          4
// 10
//  4
```

3)

```
/*
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

class Meniu {

private:

int pret;

public:

Meniu(int p) : pret{ p } {};

virtual string descriere() = 0;

virtual int getPret() {

return pret;
}

virtual ~Meniu() = default;
};

class MicDejun : public Meniu{
```

```
private:

string denumire;

public:

MicDejun(string d, int p) : Meniu{ p }, denumire{ d }{};

string descriere() override {

return denumire;
}

};

class CuRacoritoare : public Meniu {

private:

Meniu* m;

public:

CuRacoritoare(Meniu* m0) : Meniu{ m0->getPret() }, m{ m0 }{};

string descriere() override {

return m->descriere() + " cu racoritoare";
```

```
}

int getPret() override {

return m->getPret() + 4;
}

~CuRacoritoare() {

delete m;
}
};

class CuCafea : public Meniu {

private:

Meniu* m;

public:

CuCafea(Meniu* m0) : Meniu{ m0->getPret() }, m{ m0 }{};

string descriere() override {

return m->descriere() + " cu cafea";
}

int getPret() override {
```

```
return m->getPret() + 5;
}

~CuCafea() {

delete m;
}
};

vector<Meniu*> functie2() {

vector<Meniu*> rez;

rez.push_back(new CuCafea(new CuRacoritoare(new MicDejun("Omleta", 15))));

rez.push_back(new CuCafea(new MicDejun("Ochiuri", 10)));

rez.push_back(new MicDejun("Omleta", 15));

return rez;
}

int main() {

vector<Meniu*> rezultat = functie2();

sort(rezultat.begin(), rezultat.end(), [](Meniu* o1, Meniu* o2) {
```

```
return o1->getPret() > o2->getPret();
});

for (auto m : rezultat) {

cout << m->descriere() << " " << m->getPret() << '\n';
delete m;
}

return 0;
}*/
```

4)

```
/*
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template<typename TElem>
class Measurement {

private:

TElem val;

public:
```

```
Measurement(TElem v) : val{ v } {};

TElem value() const{

return val;
}

Measurement<TElem>& operator+(const Measurement<TElem>& e) {

val = val + e.val;
return *this;
}

friend bool operator< (const Measurement<TElem>& e1, const Measurement<TElem>& e2) {

return e1.val < e2.val;
}
};

int main() {
//creaza un vector de masuratori cu valorile (10,2,3)
std::vector<Measurement<int>> v{ 10,2,3 };
v[2] + 3 + 2; //aduna la masuratoarea 3 valuarea 5
std::sort(v.begin(), v.end()); //sorteaza masuratorile
//tipareste masuratorile (in acest caz: 2,8,10)
for (const auto& m : v) std::cout << m.value() << ",";
return 0;
}*/
```

# Varianta 3

1 Specificați si testați funcția: (1.5p)

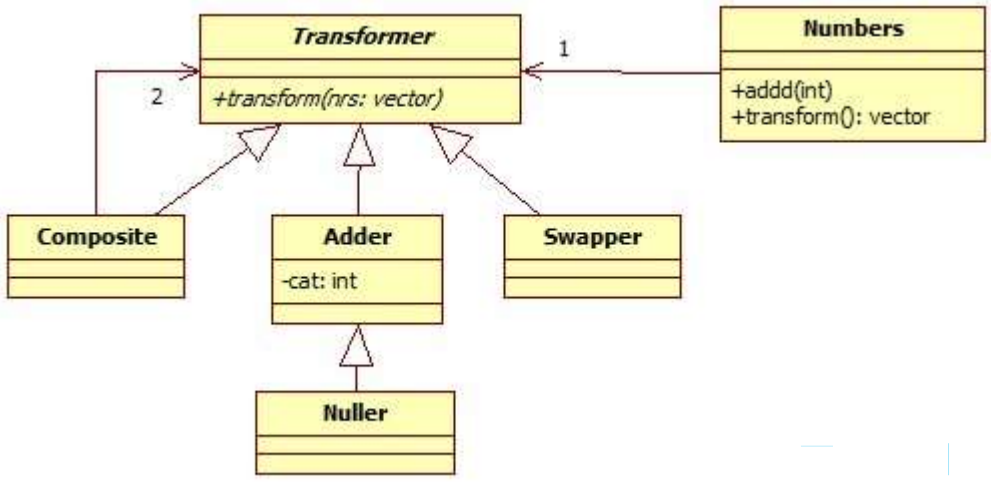
```
bool f(int a) {
     if (a <= 0)
          throw std::exception("Illegal argument");
     int d = 2;
     while (d<a && a % d>0) d++;
     return d>=a;
}
```

2 Indicați rezultatul execuției pentru următoarele programe c++. Dacă sunt erori indicați locul unde apare eroarea si motivul.

```
//2 a  (1p)
#include <iostream>
using namespace std;
class A {
public:
 A(){cout << "A()" << endl;}
 void print() {cout << "printA" <<
endl;}
};
class B: public A {
public:
 B(){cout << "B()" << endl;}
 void print() {cout << "printB" <<
endl;}
};
int main() {
     A* o1 = new A();
     B* o2 = new B();
     o1->print();
     o2->print();
     delete o1;delete o2;
   return 0;
}
```

```
//2 b  (0.5p)
#include <iostream>
#include <vector>
using namespace std;
int main() {
     vector<int> v;
     v.push_back(5);
     v.push_back(7);
     v[0] = 6;
     v.push_back(8);
     auto it = v.begin();
     it = it + 1;
     while (it != v.end()) {
          cout << *it << endl;
          it++;
     }
   return 0;
}
```

3 Scrieți codul C++ ce corespunde diagramei de clase UML. **(4p)**




- Clasa abstracta **Transformer** are o metoda pur virtuala transform(nrs)
- Metoda transform() din clasa **Adder** adaugă la fiecare număr un număr dat (cat) , metoda transform din **Swapper** interschimbă numere consecutive (poziția 0 cu poziția 1, poziția 2 cu 3, etc) iar transform() din clasa **Nuller** înlocuiește numărul cu 0 daca in urma aplicări adunării numărul este > 10 sau lasă numărul ce rezulta in urma adunării. Clasa **Composite** in metoda transform() aplica succesiv cele doua transformări folosind **Transfromer**-ele agregate.
- Metoda transform() din clasa **Numbers** ordonează descrescător numerele adăugate cu add si apelează metoda transform(nrs) din Transformer-ul continut.

Se cere:

1 Codul C++ **doar pentru clasele**: **Transformer**,**Composite**, **Nuller (0.75p)**
2 Scrieți o funcție fiecare creează si returnează un obiect **Numbers** care compune un Nuller (cat=9) cu un Swapper compus cu un Adder (cat=3). **(0.5p)**
3 In funcția main apelați funcțiile de mai sus, adăugați cate 5 numere in cele doua obiecte **Numbers**. apoi apelați funcția transform pentru ambele. **(0.25p)**

- Creați doar metode si atribute care rezulta din diagrama UML (adăugați doar lucruri specifice C++ ex: constructori). Nu adăugați câmpuri, metode, nu schimbați vizibilitatea, nu folosiți friend. **Barem**: **1.5p** Polimorfism, **1p** Gestiunea memoriei, **1.5p** Defalcat mai sus

4 Definiți clasele ToDo si Examen general astfel încât următoarea secvență C++ sa fie corecta sintactic si sa efectueze ceea ce indică comentariile. (2p)

```
void todolist() {
	ToDo<Examen> todo;
	Examen oop{ "oop scris","8:00" };
	todo << oop << Examen{"oop practic", "11:00"};	//Adauga 2 examene la todo
	std::cout << oop.getDescriere();//tipareste la consola: oop scris ora 8:00
	//itereaza elementele adaugate si tipareste la consola lista de activitati
	//in acest caz tipareste: De facut:oop scris ora 8:00;oop practic ora 11:00
	todo.printToDoList(std::cout);
}
```

1)

```
/*
#include <iostream>
#include <exception>
#include <cassert>
#include <string>

using namespace std;

// Rolul acestei functii este de a verifica daca numarul a, ce este
// trimis ca si parametru este prim sau nu. Aceasta functie are un
// caz special si anume cazul in care a = 1, pentru care functia
// returneaza true, dar 1 totusi nu este numar prim.
// parametrii de intrare: a - intreg
// parametrii de iesire: true - daca a este prim sau 1, altfel false
// @ Functia arunca o exceptie in cazul in care a este mai mic sau
// egal decat 0
bool f(int a) {
if (a <= 0)
throw std::exception("Illegal argument");
int d = 2;
while (d < a && a % d>0) d++;
return d >= a;
}

void teste() {

try {
f(0);
```

```
assert(false);
}
catch (const exception& err) {
assert(string(err.what()) == "Illegal argument");
}

assert(f(1) == true);
assert(f(4) == false);
assert(f(5) == true);
}

int main() {

teste();
return 0;
}*/

2)

//2 a (1p)
/*
#include <iostream>
using namespace std;
class A {
public:
A() { cout << "A()" << endl; }
void print() {
cout << "printA" <<
endl;
```

```
}
};
class B : public A {
public:
B() { cout << "B()" << endl; }
void print() {
cout << "printB" <<
endl;
}
};
int main() {
A* o1 = new A();
B* o2 = new B();
o1->print();
o2->print();
delete o1; delete o2;
return 0;
}*/
// Rezultatul este:
//          A()
//          A()
//          B()
//          printA
//          printB

//2 b (0.5p)
/*
#include <iostream>
#include <vector>
```

```
using namespace std;
int main() {
vector<int> v;
v.push_back(5);
v.push_back(7);
v[0] = 6;
v.push_back(8);
auto it = v.begin();
it = it + 1;
while (it != v.end()) {
cout << *it << endl;
it++;
}
return 0;
}*/
// Rezultatul este:
//         7
//         8
```

3)

```
/*
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
```

```
using namespace std;

class Transformer {

public:

virtual void transform(vector<int>& nrs) = 0;

virtual ~Transformer() = default;
};

class Adder : public Transformer{

private:

int cat;

public:

Adder(int c) : cat{ c } {};

virtual void transform(vector<int>& nrs) {

for (int i = 0; i < nrs.size(); i++)

nrs[i] += this->cat;
}
};
```

```
class Swapper : public Transformer {

void transform(vector<int>& nrs) {

for (int i = 0; i <= nrs.size() - 2; i += 2) {

int aux = nrs[i];
nrs[i] = nrs[i + 1];
nrs[i + 1] = aux;
}
}
};

class Nuller : public Adder {

public:

Nuller(int cat) : Adder{ cat } {};

void transform(vector<int>& nrs) {

Adder::transform(nrs);

for (int i = 0; i < nrs.size(); i++) {

if (nrs.at(i) > 10)

nrs[i] = 0;
```

```
}
}
};

class Composite : public Transformer {

private:

Transformer* t1;
Transformer* t2;

public:

Composite(Transformer* t01, Transformer* t02) : t1{ t01 }, t2{ t02 }{};

void transform(vector<int>& nrs) {

t1->transform(nrs);
t2->transform(nrs);
}

~Composite() {

delete t1;
delete t2;
}
};

class Numbers{
```

```
private:

Transformer* t;

vector<int> v;

public:

Numbers(Transformer* t0) : t{ t0 } {};

Numbers(Numbers&& ot) noexcept{      // move constructor

this->t = ot.t;
this->v = ot.v;
ot.t = nullptr;
}

void addd(int elem) {

v.push_back(elem);
}

vector<int>& transform() {

sort(v.begin(), v.end(), [](int a, int b) {

return a > b;
});
```

```
this->t->transform(v);

return v;
}

~Numbers() {

if(t != nullptr)

delete t;
}
};

Numbers functie2() {

Numbers n{ new Composite{ new Nuller{9}, new Composite{new Swapper, new Adder{3} } } };

return n;
}

int main() {

{
Numbers n = functie2();

n.addd(1);
n.addd(1);
n.addd(1);
```

```
n.addd(1);
n.addd(1);

vector<int> v = n.transform();

for (auto nr : v)

cout << nr << " ";
}
_CrtDumpMemoryLeaks();
return 0;
}*/
```

4)

```
/*
#include <iostream>
#include <vector>
#include <string>

using namespace std;

class Examen {

private:

string nume;

public:
```

```
Examen(const string& nume1, const string& ora) {

this->nume = nume1 + " ora " + ora;
}

const string& getDescriere() const{

return this->nume;
}
};

template<typename TElem>
class ToDo {

private:

vector<TElem> lista;

public:

ToDo& operator<<(const TElem& elem) {

lista.push_back(elem);
return *this;
}

void printToDoList(ostream& out) {
```

```
out << "De facut: ";

int nr = lista.size();

for (int i = 0; i < nr; i++) {

out << lista.at(i).getDescriere();

if (i != nr - 1)

out << ';';

}
}
};

void todolist() {
ToDo<Examen> todo;
Examen oop{ "oop scris","8:00" };
todo << oop << Examen{ "oop practic", "11:00" }; //Adauga 2 examene la todo
std::cout << oop.getDescriere(); //tipareste la consola: oop scris ora 8:00
//itereaza elementele adaugate si tipareste la consola lista de activitati
//in acest caz tipareste: De facut:oop scris ora 8:00;oop practic ora 11:00
todo.printToDoList(std::cout);
}

int main() {

todolist();
```

```
return 0;
}*/
```

# Varianta 4

1 Specificați si testați funcția: **(1.5p)**

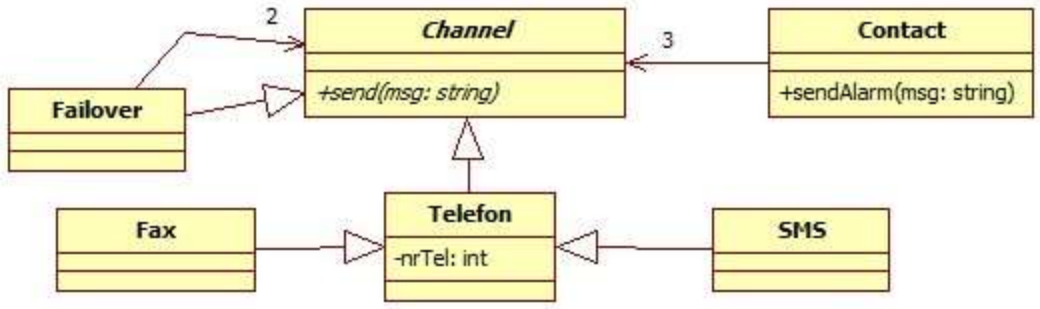
```
bool f(int a) {
	if (a <= 1)
		throw "Illegal argument";
	int aux = 0;
	for (int i = 2; i < a; i++) {
		if (a % i == 0) {
			aux++;
		}
	}
	return aux == 0;
}
```

**2** Indicați rezultatul execuției pentru următoarele programe c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

```
//2 a  (1p)
#include <vector>
#include <iostream>
using namespace std;
class A {
public:
	virtual void f() = 0;
};
class B:public A{
public:
	void f() override {
		cout << "f din B";
	}
};
class C :public B {
public:
	void f() override {
		cout << "f din C";
	}
};
int main() {
	vector<A> v;
	B b;
	v.push_back(b);
	C c;
	v.push_back(c);
	for (auto e : v) { e.f(); }
	return 0;
}
```

```
//2 b  (0.5p)
#include <iostream>
using namespace std;
class A {
public:
 A() {cout << "A" << endl;}
 ~A() {cout << "~A" << endl;    }
 void print() {cout << "print" <<
endl;}
};
void f() {
	A a[2];
	a[0].print();
}
int main() {
	f();
	return 0;
}
```

**3** Scrieți codul C++ ce corespunde diagramei de clase UML. **(4p)**




- Clasa abstractă **Channel** are o metoda pur virtuala *send*
- Metoda *send* din clasa **Telefon** tipărește mesajul "dail:" si numărul de telefon conținut, dar din când in când (in funcție de un număr aleator generat) aruncă excepție std::exception indicând ca linia este ocupată.
- Clasa **Fax** si **SMS** încearcă sa apeleze numărul de telefon si in caz de succes tipărește "sending fax" respectiv "sending sms". Clasa **Failover** încearcă sa trimită mesajul pe primul canal, dacă trimiterea eșuează (este ocupat) atunci încearcă trimiterea pe canalul secundar.
- Metoda *sendAlarm* din clasa **Contact**, încearcă sa trimită repetat mesajul pe cele 3 canale conținute pe rând până reușește trimiterea (găsește o linie care nu este ocupat).

Se cere:

**1** Codul C++ **doar pentru clasele**: **Channel**, **Failover**, **Fax, Contact(0.75p)**

**2** Scrieți o funcție C++ care creează si returnează un obiect **Contact** cu următoarele canale (alegeți voi numere de telefon): 1 Telefon; 2 Fax "daca este ocupat încearcă" Sms ; 3 Telefon "daca este ocupat încearcă" Fax "daca este ocupat încearcă" SMS. **(0.5p)**

**3** In funcția main apelați funcția de mai sus si trimiteți 3 mesaje. **(0.25p)**

- Creați doar metode si atribute care rezulta din diagrama UML (adăugați doar lucruri specifice C++ ex: constructori). Nu adăugați câmpuri, metode, nu schimbați vizibilitatea, nu folosiți friend. Folosiți STL unde exista posibilitatea.

Detalii barem: **1.5p** Polimorfism, **1p** Gestiunea memoriei, **1.5p** Restul(Defalcat mai sus)

**4** Definiți clasa Expresie generală astfel încât următoarea secvență C++ sa fie corecta sintactic si să efectueze ceea ce indică comentariile. **(2p)**

```
void operatii() {
	Expresie<int> exp{ 3 };//construim o expresie,pornim cu operandul 3
	//se extinde expresia in dreapta cu operator (+ sau -)  si operand
	exp = exp + 7 + 3;
	exp = exp - 8;
	//tipareste valoarea expresiei (in acest caz:5 rezultat din 3+7+3-8)
	cout << exp.valoare()<<"\n";
	exp.undo();  //reface ultima operatie efectuata
	//tipareste valoarea expresiei (in acest caz:13 rezultat din 3+7+3)
	cout << exp.valoare() << "\n";
	exp.undo().undo();
	cout << exp.valoare() << "\n";  //tipareste valoarea expresiei (in acest caz:3)
}
```

1)

```
/*
#include <iostream>
#include <cassert>
#include <string>

using namespace std;

// Aceasta functie are scopul de a verifica daca numarul transmis
// ca parametru este sau nu prim, in caz afirmativ returneaza true,
// altfel false
// @ Functia arunca exceptie daca numarul este mai mic sau egal decat 1
// param de intrare: a - intreg
// param de iesire: true - daca a este prim, altfel false
bool f(int a) {
if (a <= 1)
throw "Illegal argument";
int aux = 0;
for (int i = 2; i < a; i++) {
if (a % i == 0) {
aux++;
}
}
return aux == 0;
}

void teste() {

try {
```

```
f(1);
assert(false);
}
catch (const char*){       // grija cum prinzi eroarea!!!!
assert(true);
}

assert(f(2) == true);
assert(f(4) == false);
}

int main() {

teste();
return 0;
}*/

2)

//2 a (1p)
/*
#include <vector>
#include <iostream>
using namespace std;
class A {
public:
virtual void f() = 0;
};
class B :public A {
```

```
public:
void f() override {
cout << "f din B";
}
};
class C :public B {
public:
void f() override {
cout << "f din C";
}
};
int main() {
vector<A> v;
B b;
v.push_back(b);
C c;
v.push_back(c);
for (auto e : v) { e.f(); }
return 0;
}*/
// Functia main este gresita deoarece se incearca apelarea
// unei functii ce nu este scrisa, metoda din A fiind virtuala,
// si marcata cu = 0, asteapta sa fie suprascrisa
// O solutie ar fi ca vectorul sa fie de tipul A*, cand adaugam
// B si C sa le alocam dinamic, astfel, apelul polimorfic putand
// fi realizat

/*
//2 b (0.5p)
```

```
#include <iostream>
using namespace std;
class A {
public:
A() { cout << "A" << endl; }
~A() { cout << "~A" << endl; }
void print() {
cout << "print" <<
endl;
}
};
void f() {
A a[2];
a[0].print();
}
int main() {
f();
return 0;
}*/
// Rezultat:
//          A
//          A
//          print
//          ~A
//          ~A
```

3)

```
/*
```

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Channel {

public:

virtual void send(string msg) = 0;

virtual ~Channel() = default;
};

class Telefon : public Channel {

private:

int nrTel;

public:

Telefon(int nr) : nrTel{ nr } {};

virtual void send(string msg) override{

int nr = rand();
```

```
if (nr == 0)

throw exception("Linia este ocupata!\n");

cout << "dial: " << nrTel << " " << msg << " "; // trb sters msg dar pt teste
}
};

class Fax : public Telefon {

public:

Fax(int nr) : Telefon{ nr } {};

void send(string msg) override {

try {
Telefon::send(msg);
cout << "sending fax ";
}
catch (exception) {
throw exception("Linia este ocupata!\n");
}
}
};

class SMS : public Telefon {

public:
```

```
SMS(int nr) : Telefon{ nr } {};

void send(string msg) override {

try {
Telefon ::send(msg);
cout << "sending sms ";
}
catch (exception) {
throw exception("Linia este ocupata!\n");
}
}
};

class Failover : public Channel {

private:

Channel* c1;
Channel* c2;

public:

Failover(Channel* c01, Channel* c02) : c1{ c01 }, c2{ c02 }{};

Failover(Failover&& ot) noexcept{

c1 = ot.c1;
```

```
c2 = ot.c2;
ot.c1 = nullptr;
ot.c2 = nullptr;
}

void send(string msg) override {

try {

c1->send(msg);
}
catch (exception) {

try {

c2->send(msg);
}
catch (exception) {
cout << "esuat ";
}
}
}

~Failover() {

if (c1 != nullptr)
delete c1;

if (c2 != nullptr)
```

```
delete c2;
}
};

class Contact {

private:

Channel* c1;
Channel* c2;
Channel* c3;

public:

Contact(Channel* c01, Channel* c02, Channel* c03) : c1{ c01 }, c2{ c02 }, c3{ c03 }{};

Contact(Contact&& ot) noexcept{

this->c1 = ot.c1;
this->c2 = ot.c2;
this->c3 = ot.c3;
ot.c1 = nullptr;
ot.c2 = nullptr;
ot.c3 = nullptr;
}

void sendAlarm(string msg) {

int k = 0;
```

```
while (true) {

try {

if (k == 0) // c1

c1->send(msg);

if (k == 1) // c2

c2->send(msg);

if (k == 2) // c3

c3->send(msg);

break;
}
catch (exception) {

k = (k + 1) % 3;
}
}
}

~Contact() {

if(c1 != nullptr)
```

```
delete c1;

if (c2 != nullptr)
delete c2;

if (c3 != nullptr)
delete c3;
}
};

Contact functie2() {

Contact c{
new Telefon{111},
new Failover(
new Fax{222},
new SMS{333}
),
new Failover(
new Telefon{444},
new Failover{
new Fax{555},
new SMS{666}
}
)
};

return c;
}
```

```
int main() {

Contact c = functie2();

c.sendAlarm("Salut1");
cout << '\n';

c.sendAlarm("Salut2");
cout << '\n';

c.sendAlarm("Salut3");
cout << '\n';

return 0;
}*/

4)

/*
#include <iostream>
#include <string>
#include <vector>

using namespace std;

template<typename TElem>
class Expresie {
```

```
private:

vector<pair<TElem, int>> operatii;

public:

Expresie(TElem elem) {

operatii.push_back(make_pair(elem, 1));
}

Expresie& operator +(const TElem& elem) {

operatii.push_back(make_pair(elem, 1));
return *this;
}

Expresie& operator -(const TElem& elem) {

operatii.push_back(make_pair(elem, -1));
return *this;
}

TElem valoare() {

TElem rez = 0;

for (const auto& el : operatii)
```

```
rez += el.first * el.second;

return rez;
}

Expresie& undo() {

if (operatii.size() != 0)

operatii.pop_back();

return *this;
}
};

void operatii() {
Expresie<int> exp{ 3 };//construim o expresie,pornim cu operandul 3
//se extinde expresia in dreapta cu operator (+ sau -) si operand
exp = exp + 7 + 3;
exp = exp - 8;
//tipareste valoarea expresiei (in acest caz:5 rezultat din 3+7+3-8)
cout << exp.valoare() << "\n";
exp.undo(); //reface ultima operatie efectuata
//tipareste valoarea expresiei (in acest caz:13 rezultat din 3+7+3)
cout << exp.valoare() << "\n";
exp.undo().undo();
cout << exp.valoare() << "\n"; //tipareste valoarea expresiei (in acest caz:3)
}
```

```
int main() {


operatii();

return 0;

}*/
```

# Varianta 5

1 Specificați si testați funcția: **(1.5p)**

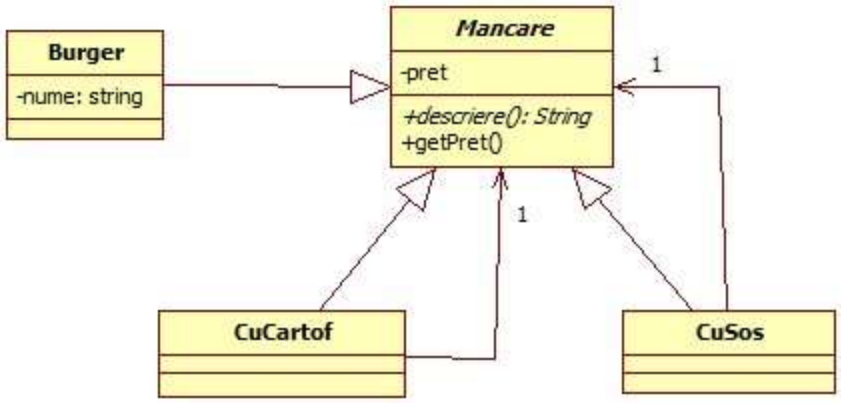
```
void f(vector<int>& l, int poz) {
	if (poz < 0 || poz >= l.size()) throw exception{};
	int a = 0;
	int b = l.size()-1;
	int nr = l[poz];
	while (a < b) {
		while (a<b && l[a] < nr ) a++;
		while (b>a && l[b] > nr) b--;
		if (a < b) {
			swap(l[a], l[b]);
			if (l[a] == l[b] && l[b] == nr) a++;
		}
	}
}
```

**2** Indicați rezultatul execuției pentru următoarele programe c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

```
//2 a  (1p)
#include <vector>
#include <iostream>
class A {
public:
	virtual void print() = 0;
};
class B : public A {
public:
	virtual void print() {
		std::cout << "printB";
	}
};
class C : public B {
public:
	virtual void print() {
		std::cout << "printC";
	}
};
int main() {
	std::vector<A> v;
	B b; C c;
	v.push_back(b);
	v.push_back(c);
	for (auto e : v) { e.print(); }
	return 0;
}
```

```
//2 b  (0.5p)
void f(bool b) {
	std::cout << "1";
	if (b) {
		throw
std::exception("Error");
	}
	std::cout << "3";
}
int main() {
	try {
		f(false);
		f(true);
		f(false);
	}
	catch (std::exception& ex) {
		std::cout << "4";
	}
	return 0;
}
```

**3** Scrieți codul C++ ce corespunde diagramei de clase UML. (**4p**)




- Clasa abstracta **Mancare** are o metoda pur virtuala descriere()
- **CuCartof** si **CuSos** conțin o mâncare si metoda descriere() adaugă textul "cu cartof" respectiv "cu sos" la descrierea mâncării conținute. Prețul creste cu 3 RON pentru cartofi, mâncarea cu sos costa in plus 2 RON. Clasa **Burger** reprezintă un hamburger fără cartof si fără sos, metoda descriere() returnează denumirea hamburgerului.

**Se cere:**

1 Codul C++ **doar pentru clasele**: **Mancare**, **Burger**, **CuSos (0.75)**
2 Scrieți o funcție C++ care returnează o lista de mâncăruri: un burger „McPuisor", un burger „BigTasty" cu cartof si sos, un burger „Booster" cu cartof si un burger „Booster" cu sos (alegeți voi prețul de baza pentru fiecare mâncare). **(0.5)**
3 In programul principal creați o lista de mâncăruri (folosind funcția descrisa mai sus), apoi tripartiți descrierea si prețul pentru fiecare in ordinea descrescătoare a preturilor. **(0.25)**

Creați doar metode si atribute care rezulta din diagrama UML (adăugați doar lucruri specifice C++ ex: constructori). Nu adăugați câmpuri, metode, nu schimbați vizibilitatea, nu folosiți friend si static. Folosiți STL unde exista posibilitatea. Detalii barem: **1.5p** Polimorfism, **1p** Gestiunea memoriei**, 1.5p** Restul (defalcat mai sus)

**4** Definiți clasa Conferinta si Sesiune astfel încât următoarea secvență C++ să fie corectă sintactic si să efectueze ceea ce indică comentariile. **(2p)**

```
int main() {
	Conferinta conf;
	//add 2 attendants to "Artifiial Inteligente" track
	conf.track("Artifiial Inteligente") + "Ion Ion" + "Vasile Aron";
	//add 2 attendants to "Software" track
	Sesiune s = conf.track("Software");
	s + "Anar Lior" + "Aurora Bran";
	//print all attendants from group "Artifiial Inteligente" track
	for (auto name : conf.track("Artifiial Inteligente")) {
		std::cout << name << ",";
	}
	//find and print all names from Software track that contains "ar"
	vector<string> v = conf.track("Software").select("ar");
	for (auto name : v) { std::cout << name << ","; }
}
```

1)

```
/*
#include <iostream>
#include <vector>
#include <cassert>

using namespace std;

// Rolul acestei functii este de a pune in stanga elementului de pe pozitia poz, trimisa
// ca si parametru, elementele mai mici decat acesta, respectiv, in partea dreapta, elementele
// mai mari
// param de intrare: l - vector de intregi, poz - intreg
// param de iesire: nu avem
// @ Arunca exceptie in cazul in care parametru poz este negativ sau depaseste lungimea vectorului trimis
// ca si parametru
void f(vector<int>& l, int poz) {
if (poz < 0 || poz >= l.size()) throw exception{};
int a = 0;
int b = l.size() - 1;
int nr = l[poz];
while (a < b) {
while (a < b && l[a] < nr) a++;
while (b > a && l[b] > nr) b--;
if (a < b) {
swap(l[a], l[b]);
if (l[a] == l[b] && l[b] == nr) a++;
}
```

```
}
}

void teste() {

vector<int> l{ 7, 9, 5, 4, 3 };
f(l, 2);

assert(l.at(0) == 3);
assert(l.at(1) == 4);
assert(l.at(2) == 5);
assert(l.at(3) == 9);
assert(l.at(4) == 7);

try {
f(l, -1);
assert(false);
}
catch (exception) {
assert(true);
}
try {
f(l, 5);
assert(false);
}
catch (exception) {
assert(true);
}
}
```

```
int main() {


teste();
return 0;
}*/


2)


//2 a (1p)
/*
#include <vector>
#include <iostream>
class A {
public:
virtual void print() = 0;
};
class B : public A {
public:
virtual void print() {
std::cout << "printB";
}
};
class C : public B {
public:
virtual void print() {
std::cout << "printC";
}
};
```

```
int main() {
std::vector<A> v;
B b; C c;
v.push_back(b);
v.push_back(c);
for (auto e : v) { e.print(); }
return 0;
}*/
// Rezultatul este: eroare de compilare pentru ca nu se pot instantia clase pur abstracte

//2 b (0.5p)
/*
#include <iostream>
#include <exception>
void f(bool b) {
std::cout << "1";
if (b) {
throw
std::exception("Error");
}
std::cout << "3";
}
int main() {
try {
f(false);
f(true);
f(false);
}
catch (std::exception& ex) {
```

```
std::cout << "4";
}
return 0;
}*/
// Rezultatul este: 1314
```

3)

```
/*
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>

using namespace std;

class Mancare {

private:

int pret;

public:

Mancare(int p) : pret{ p } {};
```

```
virtual string descriere() = 0;

virtual int getPret() {

return pret;
}

virtual ~Mancare() = default;
};

class CuSos : public Mancare {

private:

Mancare* m;

public:

CuSos(Mancare* m0) : m{ m0 }, Mancare{ m0->getPret() } {};

string descriere() override{

return "cu sos " + m->descriere() + " ";
}

int getPret() override {

return 2 + m->getPret();
```

```
}

~CuSos() {

delete m;
}
};

class CuCartof : public Mancare {

private:

Mancare* m;

public:

CuCartof(Mancare* m0) : m{ m0 }, Mancare{ m0->getPret() } {};

string descriere() override {

return "cu cartof " + m->descriere() + " ";
}

int getPret() override {

return 3 + m->getPret();
}

~CuCartof() {
```

```
delete m;
}
};

class Burger : public Mancare {

private:

string nume;

public:

Burger(int pret, string n) : Mancare{ pret }, nume{ n }{};

string descriere() override {

return nume + " ";
}
};

vector<Mancare*> functie2() {

vector<Mancare*> rez;

rez.push_back(new Burger(1, "McPuisor"));
rez.push_back(new CuCartof(new CuSos(new Burger(1, "BigTasy"))));
rez.push_back(new CuCartof(new Burger(1, "Booster")));
rez.push_back(new CuSos(new Burger(1, "Booster")));
```

```
return rez;
}

int main() {

{
vector<Mancare*> l = functie2();

sort(l.begin(), l.end(), [](Mancare* m1, Mancare* m2) {

return m1->getPret() > m2->getPret();
});

for (const auto& m : l) {

cout << m->descriere() << m->getPret() << '\n';
delete m;
}
}

_CrtDumpMemoryLeaks();

return 0;
}*/

4)

/*
```

```
#include <iostream>                    // nu e corecta
#include <string>
#include <vector>
#include <unordered_map>

using namespace std;

class Sesiune {

public:

vector<string> studenti;
string cheie;

Sesiune& operator+(const string& s) {

studenti.push_back(s);
return *this;
}

vector<string>& select(const string& s) {

vector<string> rez;

for (const auto& st : studenti)

if (st.find(s) != string::npos)

rez.push_back(st);
```

```
return rez;
}

vector<string>::iterator begin() {

return this->studenti.begin();
}

vector<string>::iterator end() {

return this->studenti.end();
}
};

class Conferinta {

private:

vector<Sesiune> lista;

public:

Sesiune& track(const string& c) {

for (auto& s : lista)

if (s.cheie == c)
```

```
return s;


Sesiune s1;
s1.cheie = c;
lista.push_back(s1);
return s1;
}


};


int main() {
Conferinta conf;
//add 2 attendants to "Artifiial Inteligente" track
conf.track("Artifiial Inteligente") + "Ion Ion" + "Vasile Aron";
//add 2 attendants to "Software" track
Sesiune s = conf.track("Software");
s + "Anar Lior" + "Aurora Bran";
//print all attendants from group "Artifiial Inteligente" track
for (auto name : conf.track("Artifiial Inteligente")) {
std::cout << name << ",";
}
//find and print all names from Software track that contains "ar"
vector<string> v = conf.track("Software").select("ar");
for (auto name : v) { std::cout << name << ","; }
return 0;
}*/
```

# Varianta 6

1 Specificați si testați funcția: (1.5p)

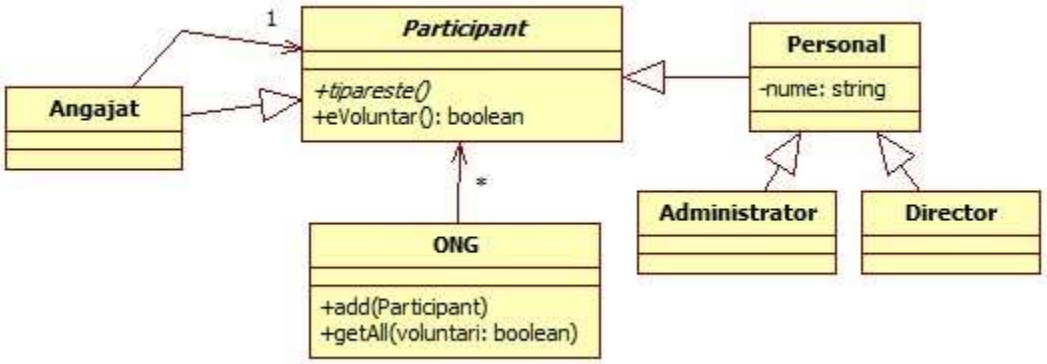
```
using namespace std;
#include <vector>
#include <string>
#include <algorithm>
#include <map>
vector<int> f(vector<int> l) {
	if (l.size() == 0)
		throw exception("Illegal argument");
	map<int, int> c;
	for (auto e : l)  {
		c[e]++;
	}
	sort(l.begin(), l.end(), [&](int a, int b) {
		return c[a] > c[b]; });
	return l;
}
```

2 Indicați rezultatul execuției pentru următoarele programe c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

```
//2 a  (1p)
#include <vector>
#include <iostream>
class A {
public:
	A() {
		std::cout << "A";
	}
	virtual void print() {
		std::cout << "printA";
	}
};
class B : public A {
public:
	B() {
		std::cout << "B";
	}
	virtual void print() {
		std::cout << "printB";
	}
};
int main() {
	std::vector<A> v;
	A a;
	B b;
	v.push_back(a);
	v.push_back(b);
	for (auto e : v) {e.print();}
	return 0;
}
```

```
//2 b  (0.5p)
#include <iostream>
using namespace std;
class A {
public:
  A() {cout << "A" << endl;}
  ~A() {cout << "~A" << endl;}
  void print() {
    cout << "print" << endl;
  }
};
void f() {
	A a[2];
	a[1].print();
}
int main() {
	f();
	return 0;
}
```

**3** Scrieți codul C++ ce corespunde diagramei de clase UML. **(4p)**




- Clasa abstracta **Participant** are o metoda pur virtuala *tipareste*.
- Metoda *tipareste* din clasa **Personal** tipărește numele persoanei.
- Clasa **Administrator** si **Director** pe lângă ce tipărește clasa de baza mai tipărește si cuvântul "Administrator" respectiv "Director" .
- Clasa **Angajat** tipărește, pe lângă ce tipărește personalul agregat de el, si textul " angajat". Metoda eVoluntar returnează false.
- Metoda *add* din clasa **ONG** permite adăugarea de orice participant, iar metoda *getAll* returnează doar participanții angajați sau participanții voluntari (in funcție de parametru). Implicit toți participanții sunt voluntari daca nu sunt decorate cu **Angajat**.

Se cere:

**1** Codul C++ **doar pentru clasele**: **Participant**, **Angajat**, **Director, ONG(0.75p)**

**2** O funcție C++ care creează si returnează un obiect **ONG** si adaugă următorii participanți (alegeți voi numele pentru fiecare): un administrator voluntar, un administrator angajat, un director voluntar si un director angajat. **(0.5p)**

**3** In funcția main tipăriți separat angajații si voluntarii din ONG. **(0.25p)**

Creați doar metode si atribute care rezulta din diagrama UML (adăugați doar lucruri specifice C++ ex: constructori). Nu adăugați câmpuri, metode, nu schimbați vizibilitatea, nu folosiți friend. Folosiți STL unde exista posibilitatea.

Detalii barem: **1.5p** Polimorfism, **1p** Gestiunea memoriei, **1.5p** Restul(Defalcat mai sus)

**4** Definiți clasa Cos generală astfel încât următoarea secvența C++ sa fie corecta sintactic si sa efectueze ceea ce indica comentariile. **(2p)**

```
void cumparaturi() {
      Cos<string> cos;//creaza un cos de cumparaturi
      cos = cos + "Mere"; //adauga Mere in cos
      cos.undo();//elimina Mere din cos
      cos + "Mere"; //adauga Mere in cos
      cos = cos + "Paine" + "Lapte";//adauga Paine si Lapte in cos
      cos.undo().undo();//elimina ultimele doua produse adaugate

      cos.tipareste(cout);//tipareste elementele din cos (Mere)
}
```

1.

Functia returneaza o copie a vectorului original sortata descrecator dupa frecventa elementelor

```
void test(){

vector<int> l;
try{
f(l);
assert false;
}
catch(exception){

}

l.push_back(2);
l.push_back(3);
l.push_back(5);
l.push_back(2);
l.push_back(2);
l.push_back(3);

vector<int> rez = f(l);
assert(rez.size() == 6);
assert(rez.at(0) == 2);
assert(rez.at(1) == 2);
assert(rez.at(2) == 2);
assert(rez.at(3) == 3);
assert(rez.at(4) == 3);
```

```
assert(rez.at(5) == 5);

}
```

2.

a.AABprintAprintA

b.

A

A

print

~A

~A

3.

```
class Participant{
protected:
bool voluntar = true;
public:
virtual void tipareste() = 0;
bool eVoluntar(){

return voluntar;
}

virtual ~Participant() = default;
```

```
};

class Personal : public Participant {
private:
string nume;
public:
Personal(string n) : nume { n } {};

void tipareste() override {

cout<<nume<<" ";
}

virtual ~Personal() = default;
};

class Administrator : public Personal {
public:
Administrator(string n): Personal(n) {};

void tipareste() override {

Personal::tipareste();
cout<< "Administrator"<<" ";
}

~Administrator() = default;
};
```

```
class Director : public Personal {
public:
Director(string n): Personal(n) {};

void tipareste() override {

Personal::tipareste();
cout<< "Director"<<" ";
}

~Director() = default;
};

class Angajat: public Participant{
private:
Participant* p;
public:
Angajat(Participant* pp): p {pp} {
voluntar=false;
};
void tipareste() override{
p->tipareste();
cout<<"angajat"<<" ";
}

~Angajat(){
delete p;
}
};
```

```
class ONG{
private:
vector<Participant*> all;
public:
ONG() = default;
void add(Participant* p){

all.push_back(p);
}

vector<Participant*> getAll(bool voluntari){
vector<Participant*> rez;
for(const auto el:all){

if(el->eVoluntar() == voluntari){
rez.push_back(el);
}
}
return rez;
}

};

ONG f2(){
ONG o;
o.add(new Administrator{"George"});
o.add(new Angajat{new Administrator {"Emil"}});
o.add(new Director{"Vasile"});
```

```
o.add(new Angajat{new Director{"Mihai"}});

return o;

}

int main(){

ONG o = f2();
vector<Participant*> voluntari = o.getAll(true);
vector<Participant*> angajati = o.getAll(false);

for (auto el : voluntari) {
el->tipareste();
cout << endl;
delete el;
}

for (auto el : angajati) {
el->tipareste();
cout << endl;
delete el;
}

return 0;
}
```

4.

```
#include <iostream>
```

```
#include <vector>
#include <string>

using namespace std;

template<typename Telem>
class Cos{
private:
vector<Telem> v;
public:
Cos() = default;

Cos& operator+(const Telem& e) {
v.push_back(e);
return *this;
}

Cos& undo(){
v.pop_back();
return *this;
}

void tipareste(ostream& out){
for(const auto el : v){

out<<el<<" ";
}
out<<endl;
}
```

```
};
```

# Varianta 7

Subject 3

1 Specificati si testati functia: **(1.5p)**

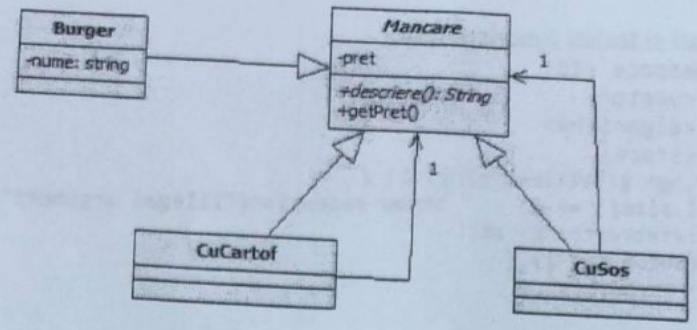
```
using namespace std;
#include <vector>
#include <algorithm>
#include <stack>
vector<string> g(vector<string> l) {
	if (l.size() == 0)	throw exception("Illegal argument");
	std::stack<string> st;
	for (auto& s : l) {
		st.push(s);
	}
	vector<string> r;
	while (!st.empty()) {
		r.push_back(st.top());
		st.pop();
	}
	return r;
}
```

**2** Indicati rezultatul executiei pentru urmatoarele programe c++. Daca sunt erori indicati locul unde apare eroarea si motivul.

```
//2 a  (1p)
#include <vector>
#include <iostream>
class A {
public:
	virtual void print() = 0;
};
class B : public A {
public:
	virtual void print() {
		std::cout << "printB";
	}
};
class C : public B {
public:
	virtual void print() {
		std::cout << "printC";
	}
};
int main() {
	std::vector<A> v;
	B b; C c;
	v.push_back(b);
	v.push_back(c);
	for (auto e : v) { e.print(); }
	return 0;
}
```

```
//2 b  (0.5p)
void f(bool b) {
	std::cout << "1";
	if (b) {
		throw
std::exception("Error");
	}
	std::cout << "3";
}
int main() {
	try {
		f(false);
		f(true);
		f(false);
	}
	catch (std::exception& ex) {
		std::cout << "4";
	}
	return 0;
}
```

3 Scrieti codul C++ ce corespunde diagramei de clase UML. (2p)




- Clasa abstracta **Mancare** are o metoda pur virtuala descriere()
- **CuCartof** si **CuSos** contin o mancare si metoda descriere() adauga textul "cu cartof" respectiv "cu sos" la descrierea mancarii continute. Pretul creste cu 3 RON pentru cartofi, mancarea cu sos costa in plus 2 RON.
- Clasa **Burger** reprezinta o un hamburger fara cartof si fara sos, metoda descriere() returneaza denumirea hamburgerului.

Scrieti o functie C++ care returneaza o lista de mancaruri: un burger BigMac, un burger BigMac cu cartof si sos, un burger Zinger cu cartof si un burger Zinger cu sos (alegeti voi pretul de baza pentru fiecare mancare).
In programul principal se creaza o lista de mancaruri (folosind functia descrisa mai sus), apoi tripariti descrierea si pretul pentru fiecare in ordinea descrescatoare a preturilor. Creati doar metode si atribute care rezulta din diagrama UML (adaugati doar lucruri specifice C++ ex: constructori). Implementati corect gestiunea memoriei. (2p)

4 Definiti clasa Carnet generala astfel incat urmatoarea secventa C++ sa fie corecta sintactic si sa efectueze ceea ce indica comentariile. **(2p)**

```
void anscolar() {
	Carnet<int> cat;
	cat.add("SDA", 9);//adauga nota pentru o materie
	cat.add("OOP", 7).add("FP", 10);
	cout << cat["OOP"]; //tipareste nota de la materia data (7 la OOP);
	//removeLast() sterge ultima nota adaugata in carnet
	cat.removeLast().removeLast();//sterge nota de la FP si OOP
	try{
		//se arunca exceptie daca nu exista nota pentru materia ceruta
		cout << cat["OOP"];
	}catch (std::exception& ex) {
		cout << "Nu exista nota pentru OOP";
	}
}
```

1)

Rolul acestei functii este de a returna vectorul ce continele elementele luate de la cap la coada din vectorul primit ca si parametru

Parametrii de intrare: l - vector de string

param de iesire: r - vector de string

@ Functia arunca exceptii daca vectorul primit ca si parametru este vid

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
#include <exception>
#include <cassert>

using namespace std;

vector<string> g(vector<string> l) {

if (l.size() == 0)

throw exception("Illegal arg");

stack<string> st;
for (auto& s : l) {
st.push(s);
}
vector<string> r;
while (!st.empty()) {
```

```
r.push_back(st.top());
st.pop();
}
return r;
}

void teste() {

vector<string> l;

try {

g(l);
assert(false);
}
catch (exception) {

assert(true);
}

l.push_back("a");
l.push_back("aa");
l.push_back("b");
l.push_back("bb");

vector<string> rez = g(l);
assert(rez.at(0) == "bb");
assert(rez.at(1) == "b");
assert(rez.at(2) == "aa");
```

```
assert(rez.at(3) == "a");
}

int main() {

teste();
return 0;
}
```

2) Mai e facut

3) Mai e facut

4)

```
#include <iostream>
#include <vector>
#include <string>
#include <exception>

using namespace std;

template<typename TElem>
class Carnet {

private:

vector<pair<string, TElem>> lista;
```

```
public:

Carnet& add(const string& s, const TElem& elem) {

lista.push_back(make_pair(s, elem));
return *this;
}

TElem operator [](const string& r) {

for (const auto& el : lista)

if (el.first == r)

return el.second;

throw exception();
}

Carnet& removeLast() {

lista.pop_back();
return *this;
}
};

void anscolar() {

Carnet<int> cat;
```

```
cat.add("SDA", 9);//adauga nota pentru o materie
cat.add("OOP", 7).add("FP", 10);
cout << cat["OOP"];// tipareste nota adaugata in carnet
//removelast() sterge ultima nota adaugata in carnet
cat.removeLast().removeLast();// sterge nota de la FP si OOP
try {
// se arunca exceptie daca nu exista nota pentru materia ceruta
cout << cat["OOP"];
}
catch (exception& ex) {

cout << "Nu exista nota pentru OOP";
}
}

int main() {

anscolar();
return 0;
}
```

# Varianta 8

**1** Specificați si testați funcția: (**1.5p**)

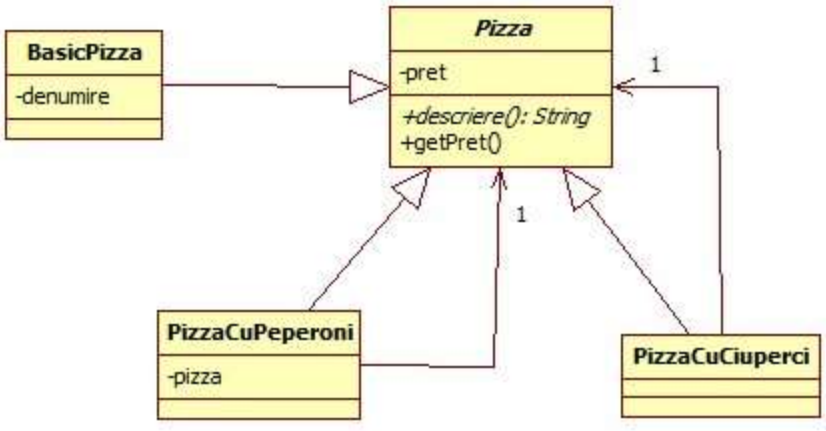
```
vector<int> f(int a) {
	if (a < 0)
		throw std::exception("Illegal argument");
	vector<int> rez;
	for (int i = 1; i <= a; i++) {
		if (a % i == 0) {
			rez.push_back(i);
		}
	}
	return rez;
}
```

**2** Indicați rezultatul execuției pentru următoarele programe c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

```
//2 a  (1p)
#include <iostream>
using namespace std;
int except(int v) {
	if (v < 0) {throw 1;  }
	else if (v > 0){
	   throw std::exception ("A");
	 }
	return 0;
}
int main(){
	try {
	   cout << except(1 < 1);
	   cout << except(-5);
	   cout << except(5);
	}catch (std::exception& e) {
		cout << "A";
	}catch (int x) {
		cout << "B";
	}
	cout << "C";
    return 0;
}
```

```
//2 b  (0.5p)
#include <iostream>
using namespace std;
class A {
	int x;
public:
	A(int x) : x{ x } {}
	 void print(){cout<< x <<",";}
};
A f(A a) {
	a.print();
	a = A{ 7 };
	a.print();
	return a;
}
int main() {
	A a{ 5 };
	a.print();
	f(a);
	a.print();
}
```

**3** Se da diagrama de clase UML: (**4p**)




- Clasa abstracta **Pizza** are o metoda pur virtuala descriere()
- **PizzaCuPeperoni** si **PizzaCuCiuperci** conțin o pizza si metoda descriere() adaugă textul "cu peperoni" respectiv "cu ciuperci" la descrierea pizzei conținute. Prețul unei pizza care conține peperoni creste cu 2 Ron, cel cu ciuperci costa in plus 3 RON.
- Clasa **BasicPizza** reprezintă o pizza fără ciuperci si fără peperoni, metoda descriere() returnează denumirea pizzei. În pizzerie exista 2 feluri de pizza de baza: Salami si Diavola, la prețul de 15 respectiv 20 RON.

Se cere:

1. Scrieți codul C++ **doar pentru clasele Pizza si PizzaCuPeperoni. (0.75p)**
2. Scrieți o **funcție** C++ care creează o comanda (**returnează o lista de pizza**) care conține: o pizza „Salami" cu ciuperci, o pizza „Salami" simplă, o pizza „Diavola" cu peperoni si ciuperci. **(0.5p)**
3. Scrieți programul principal care creează o comandă (folosind funcția descrisa mai sus), apoi tipărește descrierea si prețul pentru fiecare pizza in ordinea descrescătoare a preturilor. **(0.25p)**

Obs. Creați doar metode si atribute care rezultă din diagrama UML (adăugați doar lucruri specifice C++ ex: constructori). Nu adăugați câmpuri, metode, nu schimbați vizibilitatea, nu folosiți friend. Folosiți STL unde exista posibilitatea.
Detalii barem: **1.5p** Polimorfism, **1p** Gestiunea memoriei, **1.5p** Restul

**4** Definiți clasa Catalog astfel încât următoarea secvență C++ sa fie corectă sintactic si să efectueze ceea ce indica comentariile. (**2p**)

```
void catalog() {
      Catalog<int> cat{ "OOP" };//creaza catalog cu note intregi
      cat + 10; //adauga o nota in catalog
      cat = cat + 8 + 6;
      int sum = 0;
      for (auto n : cat) { sum += n; } //itereaza notele din catalog
      std::cout << "Suma note:" << sum << "\n";
}
```

1) mai e facut

2)

```
/*
//2 a (1p)
#include <iostream>
using namespace std;
int except(int v) {
if (v < 0) { throw 1; }
else if (v > 0) {
throw std::exception("A");
}
return 0;
}
int main() {
try {
cout << except(1 < 1);
cout << except(-5);
cout << except(5);
}
catch (std::exception& e) {
cout << "A";
}
catch (int x) {
cout << "B";
}
cout << "C";
return 0;
```

```
}*/
// Rezultatul este: 0BC

/*
//2 b (0.5p)
#include <iostream>
using namespace std;
class A {
int x;
public:
A(int x) : x{ x } {}
void print() { cout << x << ","; }
};
A f(A a) {
a.print();
a = A{ 7 };
a.print();
return a;
}
int main() {
A a{ 5 };
a.print();
f(a);
a.print();
}*/
// Rezultatul este: 5,5,7,5,
```

3)

```
/*
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>

using namespace std;

class Pizza {

private:

int pret;

public:

Pizza(int p) : pret{ p } {};

virtual string descriere() = 0;

int getPret() {

return pret;
}

virtual ~Pizza() = default;
```

```
};

class PizzaCuPeperoni : public Pizza {

private:

Pizza* p;

public:

PizzaCuPeperoni(Pizza* p0) : p{ p0 }, Pizza(p0->getPret() + 2){};

string descriere() override {

return p->descriere() + " cu peperoni ";
}

~PizzaCuPeperoni() {

delete p;
}
};

class PizzaCuCiuperci : public Pizza {

private:

Pizza* p;
```

```
public:

PizzaCuCiuperci(Pizza* p0) : p{ p0 }, Pizza(p0->getPret() + 3){};

string descriere() override {

return p->descriere() + " cu ciuperci ";
}

~PizzaCuCiuperci() {

delete p;
}
};

class BasicPizza : public Pizza {

private:

string denumire;

public:

BasicPizza(string nume, int pret) : Pizza{ pret }, denumire{ nume }{};

string descriere() override {

return denumire;
}
```

```
};

vector<Pizza*> functie_cerinta2() {

vector<Pizza*> vec;
vec.push_back(new PizzaCuCiuperci(new BasicPizza("Salami",15)));
vec.push_back(new BasicPizza("Salami", 15));
vec.push_back(new PizzaCuCiuperci(new PizzaCuPeperoni(new BasicPizza("Diavola", 20))));
return vec;
}

int main() {

{
vector<Pizza*> v = functie_cerinta2();

std::sort(v.begin(), v.end(), [](Pizza* p1, Pizza* p2) {

return p1->getPret() > p2->getPret();
});

for (auto el : v) {

cout << el->descriere() << " " << el->getPret() << '\n';
delete el;
}
}

_CrtDumpMemoryLeaks();
```

```
return 0;
}*/

4)

/*
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template<typename TElem>
class Catalog {

private:

string materie;
vector<TElem> vec;

public:

Catalog(string mat) : materie{ mat } {};

Catalog& operator+(const TElem& elem) {

vec.push_back(elem);
return(*this);
}
```

```
typename vector<TElem>::iterator begin() {

return vec.begin();
}

typename vector<TElem>::iterator end() {

return vec.end();
}
};

void catalog() {
Catalog<int> cat{ "OOP" };//creaza catalog cu note intregi
cat + 10; //adauga o nota in catalog
cat = cat + 8 + 6;
int sum = 0;
for (auto n : cat) { sum += n; } //itereaza notele din catalog
std::cout << "Suma note:" << sum << "\n";
}

int main() {

catalog();
return 0;
}*/
```