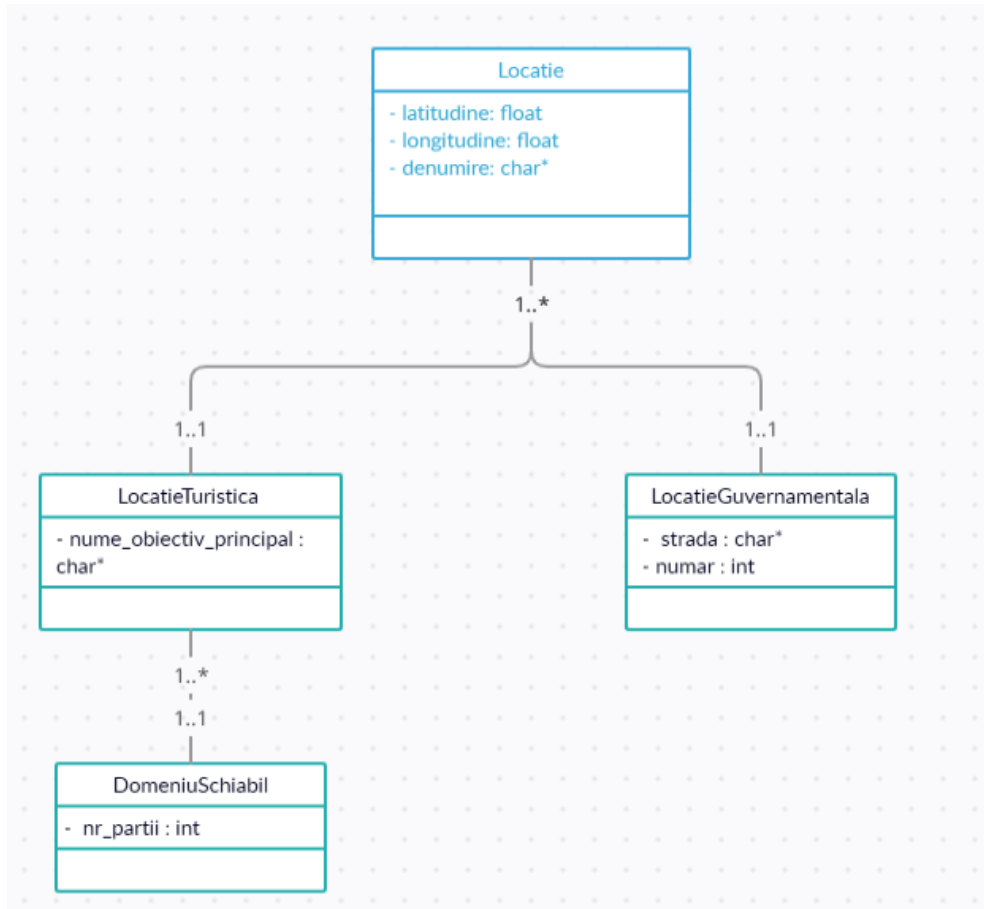


Explicații suplimentare LP2

Pentru a putea ilustra și explica mai detaliat conceptele ce se doreau atinse în cadrul LP2 și care ar fi fost componentele de bază ale unei rezolvări corecte vom considera următoarea ierarhie de clase:



Precizări legate de moștenire

Chiar dacă în figură pentru clasa **DomeniuSchiabil** de exemplu avem precizată doar variabila `nr_partii` de tip `int`, clasa **DomeniuSchiabil** are și parametrii `nume_obiectiv_principal` de tip `char*` (din cadrul clasei părinte **LocațieTuristică**), cât și parametrii `latitudine`, `longitudine` și `denumire` (din cadrul superclasei/clasei părinte a clasei **LocațieTuristică** și anume clasa **Locație**).

Clasa **DomeniuSchiabil** este responsabilă de inițializarea (și eventual distrugerea) propriilor parametrii, restul parametrilor săi fiind administrați de către clasele aflate mai sus în ierarhie. Prezintă implementarea schematică în C++ a ierarhiei de mai sus:

```

class Locatie {
protected:
    float latitudine;
    float longitudine;
    char* denumire;
};

class LocatieTuristica : public Locatie {
protected:
    char* nume_obiectiv_principal;
};

class LocatieGuvernamentala : public Locatie {
protected:
    char* strada;
    int numar;
};

class DomeniuSchiabil : public LocatieTuristica {
private:
    int nr_partii;
};

```

O scurtă precizare aici:

Observați faptul că pentru majoritatea locațiilor variabilele membre sunt declarate protected cu excepția clasei DomeniuSchiabil unde nr_partii este declarat private.

Este bine să vă gândiți dacă în cadrul ierarhiei implementate de voi pentru clasele aflate la baza ierarhiei (aici DomeniuSchiabil și LocatieGuvernamentala) este realist sau nu scenariul în care cineva (poate chiar voi mai târziu) va dori să folosească mai târziu respectiva clasa drept clasă părinte și să îi particularizeze implementarea moștenind din aceasta.

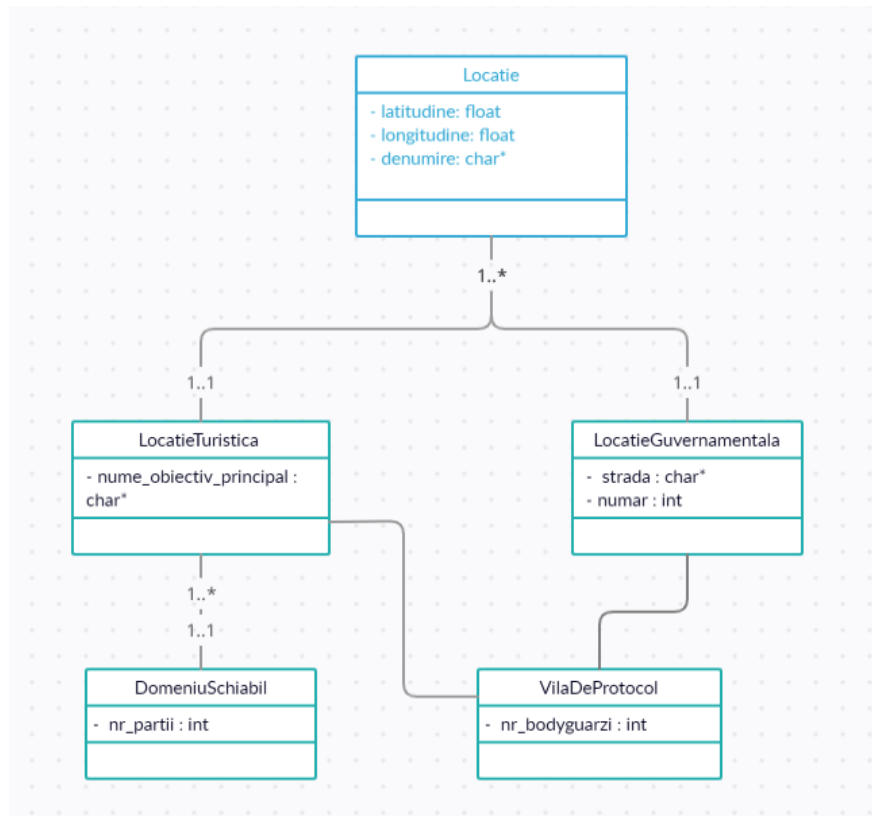
În acest caz este destul de posibil să existe variante mai particulare de locații guvernamentale, dar e puțin mai greu de imaginat o particularizare pentru DomeniuSchiabil. În acest sens am considerat potrivit să lăsăm variabilele acestei clase să fie private. Nu este ceva obligatoriu, însă este util de avut în vedere și acest aspect, deoarece tendința ar fi poate să declarați direct toate variabilele protected într-o ierarhie de moștenire, însă nu tot timpul este neapărat necesar/util pe viitor acest aspect. Un alt exemplu de situații în care o variabilă poate fi private într-o ierarhie de clase, air celelalte să rămână protected ar fi dacă pentru LocatieGuvernamentala am avea o variabilă statică char* cod_secret_de_acces. În acest caz ne putem imagina faptul că și alte variante particulare de locații guvernamentale au o strada și un număr asociate însă poate nu toate au un cod secret de acces și mai mult, nu am dori neapărat să lăsăm posibilitatea claselor derivate din LocatieGuvernamentala de a putea configura și

ele acest cod secret, ci aceasta treaba ar trebui gestionată strict de către clasa LocatieGuvernamentala. (din nou, doar un exercițiu de imaginație, nu este ceva obligatoriu de implementat de către voi).

Moștenirea virtuală

Înainte de a trece la implementarea propriu-zisă a componentelor ierarhiei aş mai vrea să lămuresc un aspect şi anume moștenirea virtuală. Aceasta NU are sens dacă în ierarhia noastră nu există situații de moștenire multiplă (nu ne va aduce niciun beneficiu, mai mult moștenirea multiplă este un exemplu de design impropriu/nepotrivit al unei ierarhii de clase ce vine şi complică implementarea; de exemplu alte limbaje cum ar fi Java nu oferă posibilitatea moștenirii multiple).

Moștenirea virtuală are sens doar pentru a evita apariția duplicatelor în situații de moștenire multiplă, DOAR în cazul în care clasele din care se moștenește au o superclasă comună ambelor (problema diamantului). În exemplul de mai sus ne putem imagina că Președintele României își petrece vacanțele la o vilă de protocol care are diferite atracții (obiective de vizitat) și deci este în același timp o locație guvernamentală (adresele proprietăților statului fiind cel mai adesea cunoscute publicului), cât și o locație turistică (strict pentru președinte, acesta dorind să știe care este principalul obiectiv de vizitat pentru fiecare vilă de protocol din patrimoniu).



Aici am avea de făcut următoarele modificări în ceea ce privește moștenirea:

```
class LocatieTuristica : public virtual Locatie {
protected:
    char* nume_obiectiv_principal;
};

class LocatieGuvernamentala : public virtual Locatie {
protected:
    char* strada;
    int numar;
};

class VilaDeProtocol : public LocatieTuristica, public LocatieGuvernamentala {
protected:
    int nr_bodyguarzi;
};
```

Avem nevoie de moștenire virtuală pentru a nu prelua de 2 ori atributele clasei Locație și a genera un conflict (Locație fiind clasă părinte atât pentru LocatieGuvernamentala, cât și pentru LocațieTuristica).

În continuare, vom face referire la ierarhia de la începutul documentului, din care lipsește VilaDeProtocol, pentru a simplifica explicațiile și a condensa prezentarea pe cât posibil.

Constructorii de inițializare (fără parametrii)

Scopul constructorului de inițializare este de a instanția toate atributele clasei pentru care este definit. Uzual se preferă să inițializăm variabilele numerice cu 0 și pointerii cu valoarea NULL. Având aceste valori implicite putem de exemplu să facem niște teste (este obiectul meu doar declarat sau este și deja instanțiat? Dacă nu este probabil că vreau să îl instanțiez eu atribuindu-i un alt obiect sau citind valori). Uzual, pointerii sunt lăsați NULL pentru a nu ocupa memorie în mod inutil, dat fiind că vom salva eventual în acel loc niște informație utilă pentru program (dacă am alocat spațiu și am salvat altceva înainte în acea locație, va trebui să am grijă să eliberez memoria înainte să realoc spațiul respectiv).

```

public:
    Locatie() {
        latitudine = 0;
        longitudine = 0;
        denumire = NULL;
    }
};

```

În cazul unei ierarhii de moștenire beneficiem de faptul că anumite variabile primesc deja valori în clasa de bază, astfel încât putem delega construcția lor către clasa de bază cu ajutorul listei de inițializare (aici pentru LocatieTuristica superclasa este Locatie):

```

public:
    LocatieTuristica() : Locatie() {
        nume_obiectiv_principal = NULL;
    }
};

```

În cazul unei ierarhii mai complexe nu este necesar să menționăm în lista de inițializare întreaga ierarhie, este suficient să menționăm clasa/clasele care sunt un părinte direct al clasei noastre (aici DomeniuSchiabil e moștenită din LocatieTuristica, iar LocatieTuristica e moștenită din Locatie; e suficient să delegăm către LocatieTuristica și apoi după cum puteți observa mai sus, va constructorul din LocatieTuristica va delega el către constructorul din Locatie fără să mai fie nevoie să facem noi ceva):

```

public:
    DomeniuSchiabil() : LocatieTuristica() {
        nr_partii = 0;
    }
};

```

Constructorii parametrizați

Cu ajutorul lor putem construi un obiect de tipul clasei respective în mai multe moduri de exemplu pentru Locatie:

```

Locatie(const char* denumire) {
    latitudine = 0;
    longitudine = 0;
    this->denumire = new char[strlen(denumire) + 1];
    strcpy(this->denumire, denumire);
}

```

```

Locatie(float latitudine, float longitudine) {
    this->latitudine = latitudine;
    this->longitudine = longitudine;
    denumire = strdup("DENUMIRE_DEFAULT");
}

```

```

Locatie(float latitudine, float longitudine, const char* denumire) {
    this->latitudine = latitudine;
    this->longitudine = longitudine;
    this->denumire = new char[strlen(denumire) + 1];
    strcpy(this->denumire, denumire);
}

```

În cazul moștenirii ne putem folosi din nou și este chiar recomandat să ne folosim de lista de inițializare:

```

LocatieTuristica(const char* nume_obiectiv_principal, float latitudine, float longitudine, const char* denumire) : Locatie(latitudine, longitudine, denumire) {
    this->nume_obiectiv_principal = strdup(nume_obiectiv_principal);
}

```

La fel ca mai sus, pentru DomeniuSchiabil e suficient să delegăm către LocatieTuristică:

```

DomeniuSchiabil(int nr, const char* nume_obiectiv_principal, float latitudine, float longitudine, const char* denumire) : LocatieTuristica(nume_obiectiv_principal, latitudine, longitudine, denumire) {
    this->nr_partii = nr;
}

```

Constructor de copiere

Sunt folosiți pentru a crea o copie locală a unui element de tipul clasei respective. În acest sens TREBUIE să alocăm memorie pentru pointeri (de orice fel, fie că vorbim de char* sau chiar de o clasa creată de noi (dacă variabila respectivă e de tip pointer îi vom alocă spațiu).

```

Locatie(const Locatie& loc) {
    this->latitudine = loc.latitudine;
    this->longitudine = loc.longitudine;
    this->denumire = new char[strlen(loc.denumire) + 1];
    strcpy(this->denumire, loc.denumire);
}

```

```

LocatieGuvernamentala(const LocatieGuvernamentala& l) : Locatie(l) {
    this->strada = new char[strlen(l.strada) + 1];
    strcpy(this->strada, l.strada);
    this->numar = l.numar;
}

```

Dacă nu se alocă explicit spațiu, nu se va crea o copie propriu-zisă, variabilele încă rămân legate între ele.

Exemplu de așa nu cu șiruri de caractere (char*):

```

LocatieGuvernamentala(const LocatieGuvernamentala& l) : Locatie(l) {
    strcpy(this->strada, l.strada);
    this->numar = l.numar;
}

```

Nu suntem siguri că în this->strada există spațiu suficient cât să cuprindă toate caracterele lui l.strada.

Un alt exemplu de așa nu necesită să ne imaginăm că o locație guvernamentală conține și un vector de angajați unde un angajat e de forma Angajat(num, prenume). Avem:

```

class LocatieGuvernamentala : public Locatie {
protected:
    char* strada;
    int numar;
    Angajat *angajati;
    int numar_angajati;
}

```

Avem grijă să alocăm spațiu pentru variabila angajati în ceilalți constructori, iar în constructorul de copiere presupunem că uităm și anume:


```

LocatieGuvernamentala(const LocatieGuvernamentala& l) : Locatie(l) {
    strcpy(this->strada, l.strada);
    this->numar = l.numar;

    this->angajati = l.angajati;
}
};

```

Sau




```

LocatieGuvernamentala(const LocatieGuvernamentala& l) : Locatie(l) {
    strcpy(this->strada, l.strada);
    this->numar = l.numar;

    this->numar_angajati = l.numar_angajati;
    for(int i = 0; i < numar_angajati; i++)
        this->angajati[i] = l.angajati[i];
}
};

```

Corect ar fi:



```

LocatieGuvernamentala(const LocatieGuvernamentala& l) : Locatie(l) {
    strcpy(this->strada, l.strada);
    this->numar = l.numar;

    this->numar_angajati = l.numar_angajati;
    this->angajati = new Angajat[numar_angajati];
    for(int i = 0; i < numar_angajati; i++)
        this->angajati[i] = l.angajati[i];
}
};

```

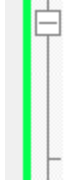
Altfel respectivele variabile rămân legate între ele în loc să existe o copie separată.

PENTRU TOATE TIPURILE DE CONSTRUCTOR este important să menționăm că NU este necesară eliberarea memoriei de orice fel (prin delete) înainte să alocăm memorie cu new.

În cazul în care codul nostru este corect, putem considera că nu fusese nimic stocat înainte/eliberat înainte de la acea zonă de memorie din punctul nostru de vedere. Practic noi în orice constructor, alocăm pentru prima dată în program memorie pentru variabilele obiectului curent, deci este suficient să alocăm cu new și nu trebuie să avem delete (delete poate fi prezent, dar nu ne-ar ajuta cu nimic, deci e practic inutil).

Destructorii

Definesc modul în care este eliberată memoria pentru clasa respectivă. Trebuie să eliberăm neapărat doar pointerii și este important să ne asigurăm că aceștia sunt diferiți de NULL, altfel posibil să crape programul deoarece nu se va putea șterge acel element (dacă ar fi NULL cel mai probabil e a doua oară când încercăm să eliberăm acel obiect prin program, din cauza unei erori logice din program sau din cauza unor apeluri recursive în urma implementării unui algoritm recursiv de exemplu). Este de asemenea recomandat, dar nu obligatoriu, să atribuim respectivului pointer în mod explicit valoarea NULL după delete (pentru siguranță).

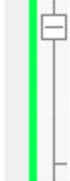


```

~Locatie() {
    if(denumire != NULL)
        delete [] denumire;
    denumire = NULL;
}

```

Pentru o clasă derivată e suficient să ne ocupăm de variabilele sale proprii. Variabilele moștenite vor fi eliberate de către destructorul din clasa/clasele de bază (acești destructori se vor apela automat imediat după ce se execută codul din destructorul curent):



```

~LocatieTuristica() {
    if(ume_obiectiv_principal != NULL)
        delete [] nume_obiectiv_principal;
    nume_obiectiv_principal = NULL;
}

```

Observați așadar că nu folosim aici lista de inițializare (evitați asta complet la destructor). De asemenea destructorii nu se apelează NICIODATĂ explicit de către voi în program, ei se apelează implicit când variabila iese din scop. Mai multe despre alocarea și eliberarea memoriei la finalul documentului.

Supraîncărcarea operatorului egal

Am observat că sunteți tentați să considerați operatorul egal drept similar cu constructorul de copiere. Cu toate acestea există o diferență semnificativă între cele două din punct de vedere al memoriei.

Am povestit mai devreme cum că pentru constructorii putem considera că e prima oară în program când se alocă spațiu pt obiectul respectiv (în cazul constructorului de copiere e vorba de un obiect temporar). Ei bine, pentru operatorul egal această presupunere nu este întotdeauna valabilă. Aș spune mai degrabă că în 90% din situațiile în care ați folosit operatorul egal, obiectul căruia îi atribuiți valori (aka operandul din partea stângă) conținea deja anumite informații (fie citiserăți ceva înainte în variabila respectivă, fie fusese construită printr-un constructor cu parametri ș.a.m.d.)

Ce e important de reținut este că în operatorul egal este obligatoriu să dezalocăm memoria înainte de a o alocă din nou și a atribui operandului din stânga (aka this) valorile operandului din partea dreaptă (pasat ca și parametru în metoda operator=): Mai multe detalii/explicații în secțiunea de alocare/eliberare a memoriei de la finalul documentului.

```

Locatie& operator=(const Locatie& loc) {
    if(this == &loc)
        return *this;

    latitudine = loc.latitudine;
    longitudine = loc.longitudine;

    if(denumire != NULL)
        delete [] denumire;

    denumire = new char[strlen(loc.denumire) + 1];
    strcpy(denumire, loc.denumire);

    return *this;
}

```

Un lucru de notat aici este primul if care tratează cazul de self-assignment. În cazul în care încercăm să atribuim un obiect sieși folosind operatorul egal, trebuie să ne asigurăm că nu îi alterăm conținutul (aici fără primul if, am ajunge să ștergem câmpul de denumire deoarece el este același atât pentru operandul din stânga, cât și pentru cel din dreapta, cei doi împărțind aceleași variabile dacă ar fi identici / situați la aceeași adresă).

De reținut și faptul că puteți refolosi operatorul egal în cadrul constructorului de copiere și nu este necesar să rescrieți toate atribuirile variabilă cu variabilă, rând cu rând (din moment ce operatorul egal face puțin mai mult decât constructorul de copiere, dar dpdv al constructorului de copiere acele verificări și delete-uri adiționale din op egal nu îi afectează cu nimic funcționalitatea)

Și acum un exemplu de reutilizare de cod:

```

LocatieTuristica& operator=(const LocatieTuristica& l) {
    if(this == &l)
        return *this;

    this->Locatie::operator=(l);

    if(ume_obiectiv_principal != NULL)
        delete [] ume_obiectiv_principal;

    ume_obiectiv_principal = new char[strlen(l.ume_obiectiv_principal) + 1];
    strcpy(ume_obiectiv_principal, l.ume_obiectiv_principal);

    return *this;
}

```

Getteri și setteri

În cadrul unei ierarhii de moștenire poate fi o idee să faceți metodele de get și set să fie protected (pentru a nu permite accesul din exteriorul ierarhiei la datele din ierarhia voastră și a le izola în esență de alte clase definite în cadrul proiectului vostru dpdv al accesului la informații pentru siguranță). Acest lucru este opțional aici, dar vă poate fi utilă pe viitor la alte materii această informație.

Acum o precizare legată de settere:

Puteți să vă simplificați viața foarte tare dacă în cadrul funcțiilor de set includeți și partea de alocare/dezallocare a memoriei și apelați funcțiile de set în operatorul egal sau în operatorii de citire și afișare (unde uzual alocați dinamic și trebuie să aveți grijă să și eliberați spațiul alocat înainte).

```
void set_denumire(const char* denumire) {
    if(this->denumire != NULL)
        delete [] this->denumire;
    this->denumire = new char[strlen(denumire) + 1];
    strcpy(this->denumire, denumire);
}
```

De asemenea funcțiile de set pot conține condițiile care se aplică variabilei respective (de exemplu nu ați dori să aveți valori negative pentru laturile unui pătrat și faceți verificarea respectivă în set / vreți să tratați o anumită excepție pentru atributul Z al clasei voastre întotdeauna și atunci faceți asta tot în set).

Supraîncărcarea operatorilor de citire și afișare

Aici nu au fost probleme, iar cerința de la LP3 vă oferă cred eu suficiente hint-uri celor care nu au reușit să implementeze afișări custom în funcție de tipul de date care trebuia afișat (folosindu-vă de polimorfism). Vă las totuși aici un exemplu de a refolosi cod din clasa de bază în clasele derivate, poate vi se pare util (e și un exemplu de upcasting totodată):

```
friend istream& operator>>(istream& in, Locatie& loc){
    char buffer[256];

    in>>loc.latitudine;
    in>>loc.longitudine;

    in.getline(buffer, 256);

    loc.set_denumire(buffer);

    return in;
}

friend istream& operator>>(istream& in, LocatieTuristica& loc){
    char buffer[256];

    in>>(Locatie&)loc;

    in.getline(buffer, 256);

    loc.set_nume_obiectiv_principal(buffer);

    return in;
}
```

Funcții virtuale. Polimorfism

O metodă trebuie declarată virtuală în cadrul ierarhiei dacă observați că aveți nevoie de un comportament diferit în funcție de clasa pentru care încercați să apelați metoda respectivă. Exemplele clasice includ funcțiile de citire și afișare (pe care nu le voi exemplifica pentru a încerca să le implementați cu toții pentru LP3). O altă posibilitate ar fi o funcție `get_tip()` care să întoarcă o valoare diferită de tip întreg în funcție de clasa din care a fost apelată. Astfel puteți spre exemplu recunoaște tipul clasei fără să folosiți `dynamic_cast`, metoda aceasta `get_tip()` fiind în esență similară cu ceea ce face `type_id` de care poate ați auzit deja la curs. Alte exemple includ, dar nu sunt limitate la calculul de arii și volume pentru diferite figuri, salarii pentru diferite tipuri de angajați, chirii pentru diferite tipuri de locuințe, taxe de timbru pentru procese, preț pentru abonamente/produse ș.a.m.d.

În acest caz vom considera că fiecare locație are o culoare diferită (exprimată în hexazecimal și returnată ca și șir de caractere) pentru bulletpoint-ul ce apare pe harta când ea este căutată de utilizator în funcție de tipul locației (gri - Locatie, verde - LocatieTuristica, galben - LocatieGuvernamentala, albastru - DomeniuSchiabil). Mai jos funcția virtuală implementată în clasa Locație:

```
virtual const char* get_bullet_RGB_color() const {  
    return "#808080"; //gray  
}
```

Odată ce am creat o funcție virtuală în clasa Locație trebuie să ne asigurăm că și destructorul acesteia este declarat virtual (pentru eliberarea corectă a memoriei).

```
virtual ~Locatie() {  
    if(denumire != NULL)  
        delete [] denumire;  
    denumire = NULL;  
}
```

În acest moment pentru cazul nostru nu mai este necesar ca metoda `get_bullet_RGB_color` sau destructorii din clasele derivate să mai fie prefixate în cadrul declarației de cuvântul virtual. Comportamentul polimorfic al programului este deja prezent și exprimat corect pe baza celor 2 declarații de mai sus.

Nota: În momentul în care aveți un destructor virtual declarat în ierarhia voastră este recomandat să îi declarați explicit pe toți ceilalți din clasele derivate. Știu că pare overkill, însă spre exemplu pentru DomeniuSchiabil dacă nu am declara noi destructorul de forma:

```
~DomeniuSchiabil() {  
}
```

Compilatorul îl va considera implicit a fi de fapt de forma:

```
virtual ~DomeniuSchiabil() {  
}
```

Ceea ce ne creează în memorie în mod inutil un VPTR și pentru clasa DomeniuSchiabil (ceea ce înseamnă că irosim puțină memorie în plus fără nici un rost deoarece acel VPTR ocupă memorie dar nu ne ajută să dezvoltăm un comportament polimorfic, noi neavând poate funcții virtuale în clasa DomeniuSchiabil, ci doar un destructor declarat implicit virtual - nerecomandat).

În continuare pentru a ilustra puțin mai bine conceptele de upcasting și downcasting putem defini o nouă clasă:

```
class Harta {  
private:  
    Locatie **locatii;  
    int nr_locatii;
```

Practic considerăm că avem o hartă ce conține mai multe tipuri de locatii (turistice, guvernamentale și domenii schiabile).

Dacă am decide ca harta noastră nu conține locatii simple (cum este cazul în situația de mai sus) ar fi de asemenea un bun prilej să redefinim funcția virtuală din clasa Locație, drept funcție virtuală pură:

```
virtual const char* get_bullet_RGB_color() const = 0;
```

(Destructorul din Locație va rămâne virtual)

Dacă decidem să facem acest pas, harta noastră nu va mai putea conține obiecte ale clasei Locație, doar pointeri către diferite locații (mai multe despre pointeri la final). Clasele ce conțin metode pur virtuale se consideră a fi clase abstracte. Fiind abstracte nu putem instanția obiecte de tipul lor.

Pentru clasa Harta voi încerca să vă prezint o abordare proprie pentru implementarea celor mai importante concepte. Au existat bineînțeles și alte alternative corecte implementate de către voi, ceea ce urmează în continuare este numai o sugestie:

Constructorul fără parametri este trivial:

```
public:
    Harta() {
        locatii = NULL;
        nr_locatii = 0;
    }
```

Constructorul parametrizat poate fi exprimat de forma:

```
Harta(Locatie **l, int numar_locatii) {
    this->locatii = new Locatie*[numar_locatii];
    this->nr_locatii = numar_locatii;

    for(int i = 0; i < nr_locatii; i++) {
        if(dynamic_cast<DomeniuSchiabil*>(l[i])) {
            locatii[i] = new DomeniuSchiabil(static_cast<DomeniuSchiabil*>(*l[i]));
        }
        else if(dynamic_cast<LocatieTuristica*>(l[i])) {
            this->locatii[i] = new LocatieTuristica(static_cast<LocatieTuristica*>(*l[i]));
        }
        else if(dynamic_cast<LocatieGuvernamentala*>(l[i])) {
            this->locatii[i] = new LocatieGuvernamentala(static_cast<LocatieGuvernamentala*>(*l[i]));
        }
    }
}
```

Aici aş avea următoarele precizări de făcut:

- Pentru ca informația să nu fie trunchiată cea mai simplă variantă este să folosim un vector de adrese (aici **locatii) care să rețină adresele a N locații (astfel încât să putem lucra cu un vector de locații).
- La respectivele adrese din vectorul de locații trebuie să salvăm obiecte de un anumit tip (fie DomeniuSchiabil, fie LocatieTuristica, fie LocatieGuvernamentala). Pentru a ști ce element trebuie să salvăm ne folosim de dynamic_cast (care va fi diferit de NULL dacă obiectul respectiv e de tipul către care încercăm să facem downcast).
- Odată ce am descoperit tipul ne alocăm memorie pentru locația curentă a obiectului nostru folosind new și ajutându-ne de constructorul de copiere deja implementat pentru fiecare clasă în parte. Pentru a putea însă să apelăm acest constructor de copiere va trebui să dereferențiem pointerul respectiv iar pentru acest lucru folosim static_cast și & care în combinație cu * ne permite să accesăm direct valoarea stocată la adresa respectivă, valoare ce știm că este de tipul de date dorit (deoarece tocmai am verificat cu dynamic_cast)

Dacă acest proces de dereferențiere vi se pare greu de făcut pe cont propriu și nu vă simțiți confortabil cu așa ceva, vă prezint o altă variantă, de data aceasta pentru operatorul egal:

```
Harta& operator=(const Harta& harta) {
    if(this == &harta)
        return *this;

    if(locatii != NULL){
        for(int i = 0; i < nr_locatii; i++)
            delete locatii[i];
        delete [] locatii;
    }

    this->nr_locatii = harta.nr_locatii;
    this->locatii = new Locatie*[harta.nr_locatii];

    for(int i = 0; i < nr_locatii; i++) {
        if(dynamic_cast<DomeniuSchiabil*>(harta.locatii[i])) {
            DomeniuSchiabil *d = dynamic_cast<DomeniuSchiabil*>(harta.locatii[i]);
            locatii[i] = new DomeniuSchiabil(*d);
        }
        else if(dynamic_cast<LocatieTuristica*>(harta.locatii[i])) {
            LocatieTuristica *l = dynamic_cast<LocatieTuristica*>(harta.locatii[i]);
            this->locatii[i] = new LocatieTuristica(*l);
        }
        else if(dynamic_cast<LocatieGuvernamentala*>(harta.locatii[i])) {
            LocatieGuvernamentala *l = dynamic_cast<LocatieGuvernamentala*>(harta.locatii[i]);
            this->locatii[i] = new LocatieGuvernamentala(*l);
        }
    }

    return *this;
}
```

Inițial verificăm pentru self-assignment. Apoi, eliberăm memoria dacă există deja memorie alocată și în cele din urmă copiem vectorul de locații.

Pașii pentru construirea vectorului de locații sunt următorii:

- Alocăm spațiu pentru N locații (unde N e nr de locatii ale variabilei harta) sub forma unui vector de pointeri (ca mai devreme)
- Apoi pentru fiecare locație pe care o vom copia venim și îi testăm tipul cu ajutorul `dynamic_cast` (verificăm dacă putem face down cast)
- Dacă tipul este conform putem atribui această valoare unui pointer de tipul tipului de date descoperit prin `dynamic_cast`
- Ultimul pas, de alocare a memoriei cu ajutorul constructorului de copiere, care mai devreme folosea `static_cast` și o sintaxă poate ușor mai greoaie, acum se rezolvă prin pasarea pointerului nou declarat către constructorul de copiere al clasei pe care o dorim, iar dereferențierea se va întâmpla automat.

Constructorul de copiere pentru clasa Harta poate fi declarat pe scurt așa (și va utiliza codul tocmai explicat mai sus):

```
Harta(const Harta& harta) {
    *this = harta;
}
```

Destructorul are următoare formă (vezi la final de ce):

```
~Harta() {  
    if(locatii != NULL){  
        for(int i = 0; i < nr_locatii; i++)  
            delete locatii[i];  
        delete [] locatii;  
    }  
    locatii = NULL;  
}
```

Iar în main putem ilustra conceptul de upcasting și respectiv testa funcționalitatea corectă a codului (într-un mod foarte simplist) prin:

```
int main()  
{  
    Locatie** l;  
    l = new Locatie*[3];  
    l[0] = new DomeniuSchiabil(3, "Postavaru", 42.5, 44.7, "PoianaBV");  
    l[1] = new LocatieGuvernamentala("CharlesDeGaulle", 1, 43.1, 42.1, "PalatulVictoria");  
    l[2] = new LocatieTuristica("Plaja", 30.5, 32, "Mamaia");  
  
    for(int i = 0; i < 3; i++){  
        l[i]->afisare();  
        cout<<endl;  
    }  
  
    Harta h(l, 3);  
    cout<<h;  
  
    return 0;  
}
```

(Unde funcția de afișare pentru diferitele locații e declarată virtual)

Dacă pentru upcasting au mai existat exemple prezentate pe parcursul acestui document și în general v-ați descurcat foarte bine cu el, aș mai dori să fac câteva precizări legate de downcasting.

În ierarhia considerată în document nu am putea face downcasting de la LocatieGuvernamentala către DomeniuSchiabil deoarece cele 2 au în comun doar clasa părinte Locatie către care amandoua pot face upcasting. Pentru downcasting ar trebui ca LocatieGuvernamentala să fie o clasă părinte/bunic/supercasă pentru DomeniuSchiabil. Dar cele 2 se află pe ramuri diferite, deci nu este cazul.

Legat de utilitatea concretă a downcasting-ului aș enumera câteva cazuri suplimentare în care mai putea fi evidențiat acesta (pe lângă exemplele de mai sus) deoarece poate nu toți ați avut de implementat o clasă care să conțină un dublu pointer. Alte cazuri în care downcasting putea fi util erau cerințele în care vi se cerea să numărați câți membri

de tipul X există (testat tot cu `dynamic_cast`) sau în cazul în care alegeați să implementați comanda de modificare a unui element în baza indexului în meniu (din nou trebuia testat tipul cu `dynamic_cast` pt asta) sau poate pur și simplu vă faceați o funcție generică de afișare/calcul cerință suplimentară cu care să parcurgeți întreg vectorul de N elemente. De exemplu puteați avea o comandă în care să afișați toate elementele de tipul X (verificând dacă tipul X există printre elem din vector cu `dynamic_cast`) și apelând apoi operatorul de afișare deja implementat (în loc să faceți afișarea virtuală). Sau la fel, pentru cerința alocată puteați implementa totul cu downcasting într-o singură funcție în loc să vă ajutați de polimorfism prin virtual pentru acea cerință suplimentară și ați fi făcut doar citirea/afișarea cu metode virtuale. Sau, când ați fi eliminat un element din vector de la o poziție dată puteați afișa un mesaj "Am eliminat un element de tipul X de pe poziția cerută". Practic acela a și fost scopul meniului mai variat, de a vă da idei de situații în care vă folosiți de vectorul de N elemente și să vă gândiți cum ați fi identificat tipul variabilelor de prin vector pentru a realiza corect comanda din meniu.

Variabile statice

Am observat că acest aspect a rămas ușor neclar pentru unii dintre voi. Câteva exemple de variabile statice ar fi: contribuția la Casa de Asigurări de Sănătate pentru un angajat (este 10% din salariu conform legii deci puteți defini un static procent_CASS = 0.1 și să îl înmulțiți cu salariul brut al angajatului pentru a obține diferența dintre salariul brut și salariul net), vârsta majoratului unei persoane (în majoritatea țărilor este 18 ani), numărul de persoane din Gara de Nord (deocamdată 14).

Toate aceste valori sunt bine cunoscute să zicem așa (în orice caz pot fi determinate printr-o căutare apriori) și nu se vor schimba prea des în timp. Ceea ce face variabilele statice să difere de niște simple constante este faptul că dacă programul vostru ar rula pentru suficient de mult timp în producție, valoarea s-ar putea modifica la un moment dat (și e posibil să nu putem opri și reporni sistemul pentru asta, ci am prefera să apelăm o comandă simplă/o metodă în producție). De exemplu un sistem ce generează ștate de plată la comandă pentru angajați va fi afectat de apariția unei legi care schimbă cota contribuțiilor la CASS de la 10 la 15%, sau va deveni brusc posibil să angajăm persoane chiar și de 15 ani în firmă (legea reglementând momentan această speță la cei peste 16 ani și impunând niște restricții legate de nr de ore pe care le pot ei presta de exemplu).

O intuiție pe care eu v-aș recomanda-o ar fi să priviți variabilele statice drept niște valori relativ constante pe care le veți inițializa la începutul programului vostru, dar care s-ar putea schimba la un moment dat în timp (din cauza apariției unui eveniment deosebit de rar, de ex o nouă lege a muncii).

Funcții statice

Sunt o serie de funcții ce lucrează cu variabile statice însă nu sunt restrânse la acestea, le putem transmite și anumite valori ca și parametru.

Cel mai clasic exemplu este acela de a folosi metodele statice pentru a ne seta un id unic pentru fiecare membru al clasei noastre (sa zicem de exemplu nr matricol al unor studenți - poate ați observat în grupa voastră ca nr voastre matricole sunt consecutive, sau cel puțin așa fusese cazul pentru mine la licență). Practic pe măsură ce s-au adăugat studenți în anul 1 la Informatică a trebuit să se incrementeze un număr matricol pentru a permite secretariatului să vp identifice în mod unic. O metodă care face acest lucru este un exemplu de metodă statică.

Dar, nu trebuie să vă limitați neapărat la acel exemplu clasic. Puteți considera drept un candidat bun de funcție statică orice funcție a unei clase care trebuie să lucreze cu un vector de obiecte de tipul clasei respective.

De exemplu:

- avem clasa Angajat(nume, prenume, salariu_brut) și dorim să putem calcula salariul net mediu din firma știind salariul brut al tuturor angajaților primind ca parametru un vector de Angajati (dat fiind că pentru orice salariu al unui angajat avem de dat aceleași valori procentuale din acesta către stat de principiu)

- avem clasa Locuinta(strada, numar, metri_patrati) și dorim să putem calcula impozitul pe metru pătrat maxim pentru o locuință de pe o anumită stradă și avem un vector de locuințe primit ca și parametru (dat fiind ca formula de calcul a impozitului este ceva standard)

Evident astfel de metode cum e cazul ultimelor 2 amintite mai sus pot fi la fel de bine implementate ca și metode de sine stătătoare, dar dat fiind faptul că ele lucrează în teorie doar cu membrii unei anumite clase puteți considera mai natural să le implementați drept metode statice pentru clasa respectivă (la anul, în Java de exemplu neavând posibilitatea să creați funcții în afara claselor va trebui să începeți să vă formați și acest mod de a gândi)

Variabile și funcții constante

Aici am văzut că vă descurcați foarte bine, pentru cei care încă au ușoare probleme o să încerc să vă dau următoarele 2 sugestii:

Dacă vă doriți să hardcodați o valoare peste tot în program (de exemplu toate șirurile de caractere ce reprezintă prenumele unei persoane să aibă maxim lungimea 20) e mai simplu să definiți în cadrul clasei o constanta FIRST_NAME_SIZE. Astfel, dacă descoperiți la un moment dat (să zicem anul viitor) că prenumele au început să fie mai lungi din cauză că noilor părinți le place să folosească mai nou nume inspirate din Star

Trek care sunt foarte lungi, veți modifica codul într-un singur loc și anume veți face `FIRST_NAME_SIZE = 50`. Similar și dacă e vorba de vectori de elemente pe care vi-i doriți neapărat alocați static de dimensiune 100 folosiți în loc de `v[100]` mai bine `v[VECTOR_SIZE]` unde `VECTOR_SIZE = 100` undeva într-o clasă (eventual declarat `protected` în clasa de bază). Pentru astfel de constante puteți de asemenea considera potrivită și o declarație de genul:

```
static const VECTOR_SIZE = 100;
```

Alocarea și eliberarea memoriei

Se face în perechi și anume dacă ați alocat ceva cu `new` tot voi trebuie să faceți și `delete`:

De pildă pentru o variabilă de tip `LocatieTuristica`:

```
LocatieTuristica *l1 = new LocatieTuristica();  
  
//-----operatii realizate cu ajutorul l1-----  
  
delete l1;
```

Sau dacă era un vector de locații turistice:

```
LocatieTuristica *l1 = new LocatieTuristica[10];  
  
//-----operatii realizate cu ajutorul l1-----  
  
delete [] l1;
```

Pentru exemplul din main de mai sus dacă dorim să alocăm memoria avem inițial următoarea logică:

- Avem nevoie să reținem N locații și alegem varianta cu dublu pointer (ce reprezintă la o adică adresa mai multor adrese)
- Alocăm un vector de N adrese (aici `N = 3`); aceste N adrese reprezintă locurile din memorie unde știm să ne uităm după valori (dar ținem cont că valorile nu sunt încă alocate); exemplu: o matrice `MxN` pentru care inițial alocăm adresele

celor M rânduri ale sale, iar mai apoi dorim să completăm cele N coloane pentru fiecare rând (și deci trebuie să alocăm spațiu pentru a reține N valori pe rând)

- La acele adrese trebuie să alocăm și spațiu pentru elementele ce vor fi conținute
- Aici elementele sunt de tipuri diferite, dar ce e cel mai important de reținut e că fiecare e alocat prin new și e vorba de un singur element din fiecare (un domeniu Schiabil, o locație guvernamentală și o locație turistică)
- După ce termin de lucrat cu **l trebuie să eliberez memoria (din moment ce eu am alocat-o) așa că o să o luăm invers. Mai întâi eliberăm cele 3 elemente distincte în for
- Acum ne rămâne de eliberat vectorul de adrese prin delete [] l (deoarece noi l-am alocat ca vector de adrese).

```
int main()
{
    Locatie** l;
    l = new Locatie*[3];
    l[0] = new DomeniuSchiabil(3, "Postavaru", 42.5, 44.7, "PoianaBV");
    l[1] = new LocatieGuvernamentala("CharlesDeGaulle", 1, 43.1, 42.1, "PalatulVictoria");
    l[2] = new LocatieTuristica("Plaja", 30.5, 32, "Mamaia");

    for(int i = 0; i < 3; i++){
        l[i]->afisare();
        cout<<endl;
    }

    Harta h(l, 3);
    cout<<h;

    for(int i = 0; i < 3; i++)
        delete l[i];
    delete [] l;

    return 0;
}
```

Exemplu alternativ aici:

```
int main()
{
    Locatie* l[3];

    l[0] = new DomeniuSchiabil(3, "Postavaru", 42.5, 44.7, "PoianaBV");
    l[1] = new LocatieGuvernamentala("CharlesDeGaulle", 1, 43.1, 42.1, "PalatulVictoria");
    l[2] = new LocatieTuristica("Plaja", 30.5, 32, "Mamaia");

    for(int i = 0; i < 3; i++){
        l[i]->afisare();
        cout<<endl;
    }

    Harta h(l, 3);
    cout<<h;

    for(int i = 0; i < 3; i++)
        delete l[i];

    return 0;
}
```

Observați faptul că nu mai alocăm noi spațiu pentru vectorul de adrese, acesta a fost declarat static. Ce ne rămâne nouă de făcut e ca mai devreme să alocăm spațiu pentru elementele conținute la acele adrese. La final tot ce avem de șters sunt elementele alocate. Vectorul de adrese nu a fost alocat și el dinamic de această dată, deci nu noi trebuie să îl ștergem.

Mai jos voi încerca să sumarizez tipurile de variabile tot cu ajutorul clasei `LocatieTuristica`:

1. `LocatieTuristica l;` // o singură variabilă alocată static
2. `LocatieTuristica *l;` // un pointer de tip `LocatieTuristica` (poate primi adresa unei //singure variabile sau putem stoca chiar și un întreg vector de //locații alocate dinamic în l)
3. `LocatieTuristica l[10];` //un vector de 10 locații turistice alocat static
4. `LocatieTuristica l[N];` //un vector de N locații turistice alocat static (trebuie ca N //să existe în prealabil în cod
5. `LocatieTuristica *l[10];` // un vector de 10 adrese către locații turistice alocat static
6. `LocatieTuristica *l[N];` // un vector de N adrese către locații turistice alocat static //(trebuie ca N să existe în prealabil în cod
7. `LocatieTuristica **l;` //un pointer către alți pointeri (poate fi văzut drept un vector //de pointeri sau un vector de adrese)

Cum se alocă de către noi memoria pentru ele dinamic:

1. Nu se alocă, e treaba compilatorului
2. `l = new LocatieTuristica();` sau `l = new LocatieTuristica[N];`
3. Nu se alocă, e treaba compilatorului
4. Nu se alocă, e treaba compilatorului
5. Se alocă spațiu doar pentru elementele de la cele 10 adrese; de exemplu:
`for(int i = 0; i < 10; i++)`
`l[i] = new LocatieTuristica();`
6. Se alocă spațiu doar pentru elementele de la cele N adrese; de exemplu:
`for(int i = 0; i < N; i++)`
`l[i] = new LocatieTuristica();`
7. Alocăm noi tot ca mai sus (și vectorul de adrese și conținutul de la fiecare adresă)

Cum se eliberează de către noi memoria pentru ele dinamic:

1. Nu se eliberează, e treaba compilatorului
2. delete l; sau dacă am alocat un vector acolo cu new, facem acum delete [] l;
3. Nu se eliberează, e treaba compilatorului
4. Nu se eliberează, e treaba compilatorului
5. Se eliberează spațiul doar pentru elementele de la cele 10 adrese; de exemplu:
delete [] l;
6. Se eliberează spațiul doar pentru elementele de la cele N adrese; de exemplu:
delete [] l;
7. Eliberăm noi tot spațiul ca mai sus (și vectorul de adrese și conținutul de la fiecare adresă)