Seminar 5

1 Downcasting

Convertirea unui obiect de tip derivat către un obiect (pointer/referinta) de tip bază se numește upcasting și este facută implicit de către compilator de fiecare dată când are posibilitatea. Conversia inversă, de la bază către derivată, se numește downcasting și nu este mereu posibilă. Mai mult, pentru a efectua acest tip de conversie, ea trebuie cerută explicit compilatorului utilizând `dynamic_cast`, care permite convertirea unui pointer sau a unei referințe către un obiect în toate direcțiile dintr-o ierarhie de clase (de la bază către derivată, de la derivată către bază etc).
Sintaxa:

```
dynamic_cast<Type&|Type*>(<expression>);
// Se citeste: conversia parametrului <expression> catre tipul Type&/Type*
```

Considerați clasele de mai jos `B` si `D`:

```
class B {
public:
    virtual void f () {}
};
class D: public B {
};
```

În cazul conversiei de pointeri, `dynamic_cast` va încerca să facă conversia obiectul care se găsește la adresa indicată de pointerul primit ca parametru la un pointer către tipul indicat între paranteze ascuțite. Dacă acea conversie reușește, atunci `dynamic_cast` va întoarce un pointer către tipul de date cerut. În cazul în care conversia nu este posibila, `dynamic_cast` va întoarce null.

```
B *b = new D();
D *d = dynamic_cast<D*>(b);
if (d != NULL) {
    cout << "Conversia lui b la D* a reusit";
} else {
    cout << "Conversia lui b la D* a esuat"
}
// va afisa: Conversia lui b la D* a reusit
```

În cazul coversiei de pointeri, `dynamic_cast` va încerca să facă conversia obiectului primit prin referința pasată ca parametru, către un obiect de tipul indicat între parantezele ascuțite. Dacă acea conversie este posibilă, atunci `dynamic_cast` va întoarce o referință către obiectul rezultat în urma conversiei. În caz negativ, `dynamic_cast` va arunca eroare de tipul `std::bad_cast`.

```
D b;
B& rb = b;
try {
    D &rd = dynamic_cast<D&>(rb);
    cout << "Conversia lui rb la D& a reusit";
} catch (std::bad_cast e) {
    cout << "Conversia lui rb la D& nu a reusit";
}
// va afisa: Conversia lui rb la D& a reusit
```

Dacă parametrul pasat catre `dynamic_cast` nu este o referință sau un pointer către un tip de date polimorfic (i.e. care sa aiba cel putin o metodă virtuală sau destructor virtual), atunci programul nu va compila.

Pentru a intui rezultatul pe care îl poate avea `dynami_cast` folosiți urmatoarea regula: conversia va reusi dacă tipul de date real al obiectului din spatele referinței/pointerului este tipul de date specificat între paranteze ascuțite. Alfel conversia va eșua.

```
class A {
public:
    virtual void f() {}
};
class B: public A {};
class C: public A {};

int main () {
    C c;
    A &ra = c;

    try {
        B& rb = dynamic_cast<B&>(ra);
        cout << "Conversia lui ra catre B& a reusit";
    } catch (std::bad_cast e) {
        cout << "Conversia lui ra catre B& nu a reusit";
    }
}
// va afisa: Conversia lui ra catre B& nu a reusit
```

Atenție deosebita când trebuie determinat care este tipul real unui obiect dintr-o ierarhie de clase. Similar cu gestionarea excepțiilor, încercările de conversie trebuie să înceapă cu cele mai de jos tipuri din ierarhie, deoarece conversia catre o bază a tipui real al obiectului va reuși.

```
class A {
public:
    virtual void f() {}
};
class B: public A {};
class C: public B {};

int main () {
    C c;
    A &ra = c;

    try {
        B& rb = dynamic_cast<B&>(ra);
        cout << "Conversia lui ra catre B& a reusit";
    } catch (std::bad_cast e) {
        cout << "Conversia lui ra catre B& nu a reusit";
    }
}
// va afisa: Conversia lui ra catre B& a reusit
```

2 Mostenire virtuala

Moșteniea multiplă poate face ca unele proprietăți moștenite să aibă același nume, rezultând multe situații de ambiguitate. Pentru a trece peste această problemă avem deja o soluție: folosirea operatorului de rezoluție de scop si numele bazei pentru a spune exact compilatorului la care proprietate moștenită facem referire. Există un caz în care putem rezolva altfel problema abiguității: proprietățile sunt moștenite din aceeași baza comuna, cunoscută si sub numele "moștenire diamant" (vezi figura 1).

În cazul acestui tip de ambiguitate, rezolvarea constă folosirea cuvântului cheie `virtual` la moștenire obținând *moștenire virtuală*. Prin folosirea acestui tip de moștenire compilatorul va comprima instanțele multiple ale bazei comune într-o singură instantă asociată direct cu clasa în care se face compresia.

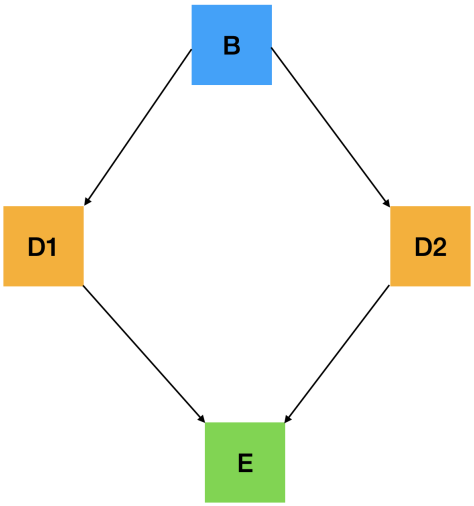

Figure 1: Ierarhie de clase diamant

```
class B {
protected:
    int x;
public:
    B() {cout << "B()";}
    B(int i) {cout << "B" << i;}
};

class D1: virtual public B { // mostenirea virtuala se foloseste cu nivel inainte de
public:                      // aparitia bazei duplicate
    D1() {cout << "D1()";}
};

class D2: virtual public B {
public:
    D2() {cout << "D2()";}
};

class E: public D1, public D2 {
public:
    E () {cout << "E()";}
    E (int i) : B(i) {    // constructorul bazei comune se apeleaza direct din clasa
        cout << "E" << i; // in care apare baza multipla; constructorul claselor care
        x = i;            // mostenesc baza comuna direct nu va mai apela
    }                     // constructorul acesteia
};


E ob1; // B()D1()D2()E()
E ob2(2022); // B2022D1()D2()E2022
```

3 Funcții template

Atunci când avem nevoie de o oferi aceeași funcție pentru mai multe tipuri de date, polimorfismul nu este întotdeauna bun (implementarea de mai multe ori ale aceleași funcții pentru diverse tipuri de parameterii nu este o practică foarte bună pentru a menține cod). Soluția este data de template-uri (sabloane).

O functie template este o functie în care tipul de date este parametrizat, deci putem avea multiple variante ale aceleasi functii scriind o singura data cod. Sintaxa pentru declararea unei functii

template este:

```
template <typename T_1, typename T_2, ..., typename T_n>
<tip - retur> <nume - functie> (<lista - parameterii>) {
    /* corp functie */
};
```

In loc de `typename` putem folosi si `class` pentru a enumera tipurile de date parametrizate in template-ul nostru.
Exemplu:

```
#include <iostream>

template <typename T> T add (T a, T b) {
    return a + b;
}

int main () {
    const int i = 4;
    const int j = 5;

    std::cout << add<int>(i, j); // instantiere explicita
    std::cout << add(i, j);      // instantiere implicita, compilatorul
                                 // deduce tipul de date
}
```

Codul asociat cu un template este compilat mai intai din punct de vedere al sintaxei. Compilarea semnatica se face la instanțiere. Orice eroare care ar putea rezulta datorită tipurilor de date cu care este instanțiat un template apare la instanțiere (daca nu instanțiem niciodata un template compilatorul nu va indica posibile erori de semnatica pentru acel template).

Pentru o funcție template putem oferi o specializare pentru anumite tipuri de date. Scopul unei specializari este sa oferim un comportament special pentru instantiere template-ului cu un anumit tip de date.

```
// specializarea functie template add pentru intregi
template <> int add <int> (int a, int b) {
    std::cout << "integer specialization ";
    return a + b;
}

int main () {
    const int i = 4;
    const int j = 5;

    std::cout << add(i, j); // integer specialization 9
}
```

In C++, o functie template nu e vazuta ca o functie propriu-zisa, transformarea template-ului intr-o functie are loc la instantiere, cand compilatorul genereaza automat of functie in care tipurile parameterizate primesc valorie (e.g. add ¡int¿). Instantierea functie template care este marcata ca functie si poate fi folosita pentru apeluri.

Putem avea in clase non template metode (statice sau nu) si functii prieten template:

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

class Point {
    int x, y;
public:
    Point (int a = 0, int b = 0) : x(a), y(b) {}
    friend ostream& operator<< (ostream&, const Point&);

    template <typename T>
    void foo (const T&);

    template <typename T>
```

```
    static Point convert (const T&, const T&);
};

template <typename T>
void Point::foo(const T& val) {
    x *= val;
    y *= val;
}

template <typename T>
Point Point::convert(const T& a, const T& b) {
    Point p;

    p.x = a; p.y = b;

    return p;
}

ostream& operator<< (ostream& out, const Point& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

int main () {
    Point p (2, 10);
    p.foo(2.4f);
    cout << p << endl;
    p = Point::convert(1.3f, 5.678f);
    cout << p << endl;
    return 0;
}
```

4 Clase template

C++ suporta si clase template pentru situatiile in care clasele pe care le scriem pot fi avea ca proprietati mai multe tipuri de date (e.g. stiva, vector, pereche, lista, matrice, hashmap etc.). Sintaxa pentru declarearea unei clase template este urmatoarea:

```
    template <typename T_1, typename T_2, ..., typename T_n>
    class <nume - clasa> {
        /* definitie clasa */
    };
```

Exemplu:

```
template <typename T> class Point;

template <typename T>
istream& operator >> (istream&, Point<T>&);


template <typename T> class Point {
    T x, y;
public:
    Point(const T&, const T&);
    int cadran ();

    template <typename U>
    void templateMethod (const U&);

    static void staticMethod ();

    template <typename U>
    static void templateStaticMethod (const U&);

    template <typename U>
    friend ostream& operator<<(ostream&, const Point<U>&);

    friend istream& operator>><T>(istream&, Point<T>&);
};

template <typename T>
Point<T>::Point(const T& a, const T& b) : x(a), y(b) { }

template <typename T>
int Point<T>::cadran() {
    if (x > 0) {
        if (y > 0) {
            return 1;
        } else if (y < 0) {
            return 2;
        }
    } else if (x < 0) {
        if (y < 0) {
            return 3;
        } else if (y > 0) {
            return 4;
        }
    }
    // the point is in origin or on axis
    return 0;
}



template<typename T>
template<typename U>
void Point<T>::templateMethod (const U& a) {
    cout << a;
}

template <typename T>
void Point<T>::staticMethod() {
    cout << "This is a static method in a template class";
}

template <typename T>
template <typename U>
void Point<T>::templateStaticMethod(const U& a) {
    cout << "This is template static method in a template class " << a;
}
```

```
67
68 template <typename T>
69 ostream& operator<<(ostream& out, const Point<T>& p) {
70     out << "(" << p.x << ", " << p.y << ")";
71     return out;
72 }
73
74 template <typename T>
75 istream& operator>>(istream& in, Point<T>& p) {
76     in >> p.x >> p.y;
77     return in;
78 }
79
80 int main () {
81     Point<int> p(0, 0);
82     cin >> p;
83     cout << p << " " << p.cadran() << endl;
84     return 0;
85 }
```

Datorită modului în care sunt compilate template-urile, o clasă template nu poate fi scrisă în header (.h) și sursă (.cpp). Pentru a putea defini separat clasa trebuie să fie instanțiat template-ul la sfârșitul fișierului cpp.

Ca in cazul functiilor template, codul pentru clasa template nu denota un tip de date. Cand are loc instantierea template-ului compilatorul declara un nou tip de date. Fiecare instantiere a unei clase template creaza un nou tip de date care nu este legat in niciun fel de celelalte instantieri ale template-ului.

```
1 template class Point <int>;   // instantiere explicita
2 Point<int> p;                 // instantiere implicita
```

Ca in cazul functiilor template, putem avea specializari pentru o clasa template, in care putem avea metode și proprietăți noi (care nu exista in definitia originala a clasei template).

```
1 template <>
2 class Point <float> {
3     /* template specialization */
4 };
```

5 Valori default in template

Ca in cazul parameterilor unei functii putem sa furnizam valori default pentru tipurile de date parameterizate intr-un template.

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T = int>
5 struct Pair {
6 public:
7     T first, second;
8 };
9
10 template <class T>
11 ostream& operator<< (ostream& out, const Pair<T>& p) {
12     out << "(" << p.first << " ," << p.second << ")";
13     return out;
14 }
15
16
17 int main () {
18     Pair<> p1 = {2, 4};
```

```
19      // in mod normal suntem obligati sa spune cu ce tip da date instantiem
20      // dar in acest caz putem omite si compilatorul va stii ca e vorba de int
21      Pair<float> p2 = {3.4f, 10.6f};
22
23      cout << p1 << " " << p2 << endl;
24      return 0;
25  }
```

Exerciții

1. Implementați o clasă template pentru listă liniară înlănțuită, cu următoarea interfață:
   - constructor cu parameterii și de copiere;
   - metodă pentru adăugarea de element nou;
   - metodă pentru căutarea unui element (rezultat boolean);
   - metodă pentru ștergerea unui element după valoare;
   - supraîncărcarea operatorului [] pentru obținerea elementului de pe poziția `i` ;
   - supraîncărcarea operatorilor `<<` și `>>` pentru citire si afișare;
   - destructor;
   - operator de atribuire;
2. Implementați o clasa template pentru stiva, cu urmatoarea interfata:
   - constructor cu parameterii și de copiere;
   - metodă pentru adăugarea de element nou;
   - metodă pentru intoarcerea elementului din varful stivei;
   - metodă pentru ștergerea unui element;
   - supraîncărcarea operatorilor `<<` și `>>` pentru citire si afișare;
   - destructor;
   - operator de atribuire;
   - metoda care elimina 1 sau mai multe elemente din stiva si le intoarce sub forma de vector.