Seminar 5

1 Metode virtuale

Metodele virtuale sunt metode ale căror funcționalitate poate fi rescrisă în clasa derivată. Spre deosebire de metodele normale, funcționalitate rescrisă se păstrează atunci când un obiect de tip derivat este utilizat folosind referințe și pointeri de tip bază.

Pentru a declara o metodă virtuală trebuie adăugat cuvântul cheie `virtual` înainte de de signatura metodei.

```
#include<iostream>
using namespace std;
class B {
    public:
    virtual void foo() {
        cout << "B";
    }
};
class D: public B {
public:
    void foo () {
        cout << "D";
    }
};

int main() public
{
    B b = D();
    b.foo();                // B
    B *pb = new D();
    pb->foo();              // D
    return 0;
}
```

Atunci când facem upcasting, la apelarea unei metode virtuale se folosește cea mai recentă redefinire a metodei (nu suntem nevoiți să redefinim la fiecare moștenire);

```
#include<iostream>
using namespace std;
class B {
    public:
    virtual void foo() {
        cout << "B::foo" << endl;
    }
    virtual void bar() {
        cout << "B::bar" << endl;
    }
};
class D: public B {
public:
    void foo () {
        cout << "D::foo" << endl;
    }

    void bar() {
        cout << "D::bar" << endl;
    }
```

```
};

class E: public D {
public:
    void bar () {
        cout << "E::bar" << endl;
    }
};

int main()
{
    B *b = new E();
    b->foo();        // D::foo
    b->bar();        // E::bar
    return 0;
}
```

Apelarea corectă a funcțiilor virtuale se realizează prin `*vptr` și `VTABLE` . Când declarăm o metodă virtuală, fiecare obiect al clasei are un nou membru `*vptr` pointer către tabela dinamică `VTABLE` care conține poineri către implementările funcțiilor virtuale.

Despre metode virtuale:

- e recomandat să fie declarate public;
- metodele virtuale nu pot fi statice;
- pentru a putea folosi polimorfismul trebuie, e recomandat să fie apelate folosind pointeri sau referințe.
- Putem avea si destructori virtuali (ajută la distrugerea obiectelor asignate prin upcasting);

2 Clase abstracte

Atunci când declarăm o metodă virtuală nu suntem obligați să furnizăm mereu o implementare. Putem transforma metoda virtuală într-o *metodă virtuală pură* adăugând `= 0` după signatura metodei.

```
class C {
public:
    virtual void foo() = 0;
};
```

Putem oferi implmenentări pentru metodele virtuale pure, iar singura metodă prin care acestea pot fi folosite sunt prin apelarea explicită în clase derivate.

```
#include<iostream>
using namespace std;
class B {
    public:
    virtual void foo() = 0;
};

void B::foo() {
    cout << "foo";
}


class D: public B {
public:
    void foo () {    // eliminarea acestei redefiniri
        B::foo();
    }
};
```

O clasă cu o metodă virtuală pură se numește *clasă abstractă*. O clasă abstractă nu poate fi instanțiată. Se pot declara doar pointeri și referințe către o astfel de clasă. Clasele care moștenesc o clasă abstractă trebuie să implementeze toate metodele virtuale pure.

3 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 unei referinte, `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
```

4 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.

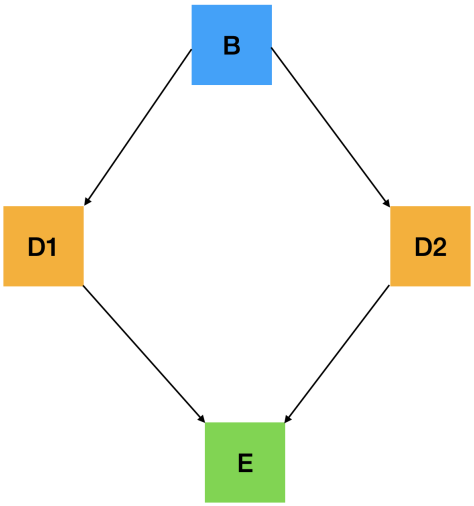

Figure 1: Ierarhie de clase diamant

```
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
```

Exerciții

Spuneți care dintre următoarele secvețe de cod compilează și care nu. În cazul secvențelor de cod care compileaza spuneți care este outputul programului. În cazul secvențelor care nu compilează sugerați o modifcare prin care secvența compilează și spuneți care este outputul secvenței modificate.

```
// 1
#include <iostream>
using namespace std;

#define ltlt <<

class B {
public:
    virtual void f() = 0;
};

void B::f(){cout ltlt "I'm so B";}

class C : public B {
};

int main () {
    B* pb = new C();
    return 0;
}
```

```
main.cpp:17:17: error: allocating an object of abstract class type 'C'
B* pb = new C();
            ^

```

```cpp
// 2
#include <iostream>
using namespace std;

class A {
public:
    A () {cout<<"A";}
    virtual ~A() = 0;
};
A::~A(){cout<<"~A";}

class B : public A {
public:
    B(){cout<<"B";}
    ~B(){cout<<"~B";}
};

class C: public B {
public:
    C () {cout << "C";}
    ~C () {cout << "~C";}
};

int main () {
    C c;
    B &pb = c;
    return 0;
}
```

```
ABC~C~B~A
```

```cpp
// 3
#include <iostream>
using namespace std;

class A {
protected:
    int x;
public:
    A () : x(1) {cout << "A";}
};
class B {
protected:
    int x;
public:
    B () : x(2) {cout << "B";}
};
class C: virtual public A {
public:
    C () {cout << "C";}
};
class D: virtual public B {
public:
    D () {cout << "D";}
};
class E: public C, public D {
public:
    E () {cout << "E" << x; }
};
int main () {
    E e;
}
```

```
main.cpp:26:26: error: member 'x' found in multiple base classes of different types
E () {cout << "E" << x; }
```

```cpp
// 4
#include <iostream>
using namespace std;

class A {
public:
    A () {cout<<"A";}
    virtual ~A() {cout<<"~A";}
};
class B : public A {
public:
    B(){cout<<"B";}
    ~B(){cout<<"~B";}
};
class C: public B {
public:
    C () {cout << "C";}
    ~C () {cout << "~C";}
};
int main () {
    C *c = new C;
    B ob = *c;
    return 0;
}
```

```
ABC~B~A
```

```cpp
// 5
#include <iostream>
using namespace std;

class C {
    int& i;
public:
    C (int& j) : i(j) {cout << "C" << " ";}
};

int main () {
    int i = 2022, j = 6;
    C ob1(i), ob2(j);
    ob1 = ob2;
    return 0;
}
```

```
main.cpp:4:7: error: cannot define the implicit copy assignment operator for 'C',
    because non-static reference member 'i' cannot use copy assignment operator
class C {
      ^
```

```cpp
// 6
#include <iostream>
using namespace std;

class B {
public:
    B () {cout << "B";}
    void plus () {cout << "+";}
};

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

int main () {
    B *pb = new D();
    try {
        D d = dynamic_cast<D&>(*pb);
    } catch (...) {
        cout << "Invalid conversion";
    }
    return 0;
}
```

```
main.cpp:19:15: error: 'B' is not polymorphic
    D d = dynamic_cast<D&>(*pb);
          ^                 ~~~
```

```cpp
// 7
#include <iostream>
using namespace std;

class B {
public:
    B () {cout << "B";}
    virtual void foo () {cout << "B::foo";}
    void bar () {cout << "B::bar";}
    virtual ~B() {}
};
struct D: B {
    D () {cout << "D";}
    void foo () {cout << "D::foo";}
    virtual void bar () {cout << "D::bar";}
};
class E: public D {
public:
    E () {cout << "E";}
    void foo () {cout << "E::foo";}
    void bar () {cout << "E::bar";}
};
int main () {
    B* pb[] = {new D, new E};
    for (int i = 0; i < 2; i++){
        pb[i]->bar(); pb[i]->foo();
        delete pb[i];
    }
    return 0;
}
```

```
BDBDEB::barD::fooB::barE::foo
```

```cpp
// 8
#include <iostream>
using namespace std;

class B {
public:
    B () {cout << "B";}
    virtual void foo () {cout << "B::foo";}
    virtual ~B() {}
};
struct D: B {
    D () {cout << "D";}
    void foo () {cout << "D::foo";}
    virtual void bar () {cout << "D::bar";}
};
class E {
public:
    E () {cout << "E";}
    virtual void foo () {cout << "E::foo";}
};
int main () {
    B *pb = new D(); E *e = new E;
    D d = dynamic_cast<D&>(*pb);
    B *b = dynamic_cast<B*>(e);
    return 0;
}
```

```
BDE
```

```cpp
// 9
#include <iostream>
using namespace std;

class B {
public:
    B () {cout << "B";}
    virtual void foo () = 0;
};
void B::foo() {cout<<"B::bar";}
struct D: B {
    D () {cout << "D";}
};
int main () {
    D d;
    return 0;
}
```

```
main.cpp:14:7: error: variable type 'D' is an abstract class
    D d;
      ^
```

```cpp
// 10
#include <iostream>
using namespace std;

class B {
public:
    B () {cout << "B";}
    virtual void foo () {cout << "B::foo";}
    virtual ~B() {}
};
struct D: B {
    D () {cout << "D";}
    void foo () {cout << "D::foo";}
    virtual void bar () {cout << "D::bar";}
};
int main () {
    B b;
    D d = dynamic_cast<D&>(b);
    return 0;
}
```

```
libc++abi: terminating due to uncaught exception of type std::bad_cast: std::bad_cast
BAbort trap: 6
```

```cpp
// 11
#include <iostream>
using namespace std;

class B {
public:
    B () {cout << "B";}
    virtual void foo () {cout<<"B::bar";}
};
class D: B {
public:
    D () {cout << "D";}
};
int main () {
    B *b = new D();
    return 0;
}
```

```
main.cpp:14:12: error: cannot cast 'D' to its private base class 'B'
    B *b = new D();
           ^

```

```cpp
// 12
#include <iostream>
using namespace std;

class B {
public:
    B () {cout << "B";}
    virtual void foo () {cout<<"B::bar";}
};
class D: virtual public B {
public:
    D () {cout << "D";}
    void foo () {cout<<"D::bar";}
};
class E: public B, public D {
public:
    E () {cout << "E";}
    void foo () {cout<<"E::bar";}
};
int main () {
    B *b = new E(); b-> foo(); delete b;
    return 0;
}
```

```
main.cpp:14:10: warning: direct base 'B' is inaccessible due to ambiguity:
    class E -> class B
    class E -> class D -> class B [-Winaccessible-base]
class E: public B, public D {
         ^~~~~~~
```

```cpp
// 13
#include <iostream>
using namespace std;

class C {
    const int i;
public:
    C (int& j) : i(j) {cout << "C" << " ";}
};

int main () {
    int i = 2022, j = 6;
    C ob1(i), ob2(j);
    ob1 = ob2;
    return 0;
}
```

```
main.cpp:4:7: error: cannot define the implicit copy assignment operator for 'C',
    because non-static const member 'i' cannot use copy assignment operator
class C {
      ^
```