SEMINAR 4

1 Moștenire

Suntem deseori în situația de a avea nevoie de a avea mai multe tipuri de date care au în comun mai multe caracteristici sau, o dată cu dezvoltarea aplicației noastre, sa fim nevoiți sa exitindem un tip de date existent fără sa pierdem forma curenta. O soluție evidentă (non-POO) ar fi sa cream noul tip plecând de la o copie a tipului curent. Genul acesta de soluție nu este foarte practică. Duplicarea codului, pe lângă faptul ca va crește numărul de linii de cod din proiect, duce la o durată mai mare a procesului de compilare. Mai mult o problema identificată în codul comun trebuie fixată separat în fiecare copie a codului.
În programarea orientată pe obiecte putem crea tipuri noi de date plecând da la un tip de date, fără a fi nevoie să duplicăm cod. Această funcționalitate se numește *moștenire*. Astfel putem crea funcționalități noi fără să pierdem funcționalități vechi și fără să creștem dimensiunea proiectului într-o manieră necontrolată (refolosim cod).
În C++ (dar nu numai) putem moșteni unul sau mai multe tipuri de date în crearea tipului nou de date. Clasa din care se moștenește se numește clasă de bază, iar clasa care moștenește se numește clasă derivată. Sintaxă moștenire:

```
class <nume> : <spec_acces_1> <baza_1>, ... , <spec_acces_n> <baza_n> {
    // definitie clasa
};
```

Specificatorii de access au rolul de specifica ce tip de access vor avea proprietățile moștenite în clasa derivată. Următorul tabel descrie cum specificatorul de access folosit la moștenire influențează specificatorul final de acces al unei prorietăți moștenite:

Moștenire \ **bază**	**public**	**protected**	**private**
public	public	protected	inaccesibil
protected	protected	protected	inaccesibil
private	private	private	inaccesibil

Când instanțiem un obiect al unei clase derivate, înainte de a se executa contrutorul clasei, se va apela implicit, de catre compilator, contructorul fără parameterii al clasei de bază. Acest comportament poate fi modificat prin folosirea listei de inițializare în constructorul clasei derivate și apelarea explicită a constructorului clasei de bază dorit.
Când este distrus un obiect, destructorii sunt apelați în ordine inversă constructorilor: mai întâi destructorul clase de derivate, apoi constructorul clase de bază.

```
#include <iostream>

using namespace std;

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

```

```
class B: public A {
    public:
    B () : A(3) {cout << "B";}
    ~B () {cout << "~B";}
};

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

int main () {
    C c;
    return 0;
}

// afiseaza A3BC~C~B~A
```

Câmpurile și metodele statice sunt și moștenite în clasa derivată. Cămpurile statice sunt partajate cu clasa de bază: o modificare a unui câmp static în clasa de bază va fi vazută și în clasa derivată și viceversa.
Dacă implementarea metodelor moștenite nu este potrivită pentru clasa derivată, atunci putem să suprascriem metoda și să îi oferim o nouă implementare.
Atunci când lucram cu pointeri și referințe către tipul de bază, putem asigna în acestea obiecte de tip derivat. Acest procedeu se numește upcasting și este posibil deoarece clasa derivat o parte comună cu clasa de bază. Aceeași afirmație nu este adevarată și în sens invers doarece nu întotdeauna tipul de bază conține exact aceleași proprietăți ca tipul derivat (există downcasting însă nu e mereu posibil).

```
#include <iostream>

using namespace std;

class A {
public:
    static void f () {
        cout << "A" << endl;
    }
};

class B: public A {
public:
    static void f () {
        cout << "B" << endl;
    }
};


int main () {
    A *a = new B();
    a->f(); // se apeleaza A::f()
    return 0;
}
```

Dacă moștenim multiple clase, este posibil să avem două proprietăți cu același nume. Pentru a putea accesa proprietățiele e nevoie sa specificăm numele clasei de bază din care vrem sa accesăm proprietatea (dacă nu facem asta compilatorul va spune ca simbolul accesat este ambiguu):

```
#include <iostream>

using namespace std;

class A {
    protected:
        int x;
};

```

```
class B {
    protected:
        int x;
};

class C: public A, public B {
    public:
        C () {
            A::x = 3;
        }
        friend void showC (C c);
};

void showC (C c) {
    cout << c.A::x << " ";
}

int main () {
    C c;
    showC(c);
    return 0;
}
```

2 Upcasting

În programarea orintată pe obiecte putem asigna un obiect de tip derivat către o variabilă / referința / pointer de tip bază. Acest fenomen se numește *upcasting*. Acest lucru este permis deoarece clasa derivată are aceleași proprietăți și metode cu clasa de bază, prin urmare un obiect de tip derivat poate substitui orcând un obiect de tip bază.

```
#include <iostream>
class B {
public:
    B () {std::cout << "B";}
};

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

int main () {
    D d;
    B b = d;
    B *pb = &d; // B *pb = new D();
    B &rb = d;
    return 0;
}
```

Atenție! Daca mostenirea este `private` sau `protected` atunci relatia de mostenire intre baza si derivata, deci implicit posibilitatea de a face upcasting, vor fi vizibile doar in zone unde specificatorul de acces permite.

3 Proprietăți, metode și obiecte const

Putem declara o variabilă ca fiind constantă (a cărei valoare nu se modifică) adăugând cuvântul cheie `const` înainte de a specifica tipul de date. Când declarăm o variabilă `const` compilatorul va încerca să nu aloce spațiu în memorie pentru acea variabilă.

```
int main () {
    const int i = 3;
    // i = 4;      eroare de compilare
```

```
    int j = i; // constanta se comporta ca orice variabila de tip intreg
    return 0;
}
```

Cuvântul cheie const poate fi "amestecat" cu pointeri și referințe. Putem declara pointer către constante, pointeri constanți dar și referințe către constante:

```
int main () {
    int i = 6;
    int b = 3;
    const int *p = &i;    // pointer catre o constanta de tip intreg
    // *p = 2;               eroare
    p = &b;               // functioneaza
    int* const cp = &i;   // pointer constant
    *cp = 6;              // functioneaza
    // cp = &b;              eroare
    const int &r1 = i;    // referinta catre o constanta
    return 0;
}
```

Într-o clasă putem avea proprietăți și metode `const`. Proprietățile constante pot fi inițializate cu parametri dați către constructor doar în lista de inițializare. Metodele `const` sunt metode care nu pot modifica starea obiectului, i.e. nu pot altera nici una dintre propietățile obiectului. O metodă poate fi declarată const prin adăugarea cuvântului cheie `const` după lista de parametrii. Exemplu:

```
class A {
    const int i;
    float f;
public:
    A (int a = 0, float b = 3.0) : i(a) {
        f = b;
        // i = (int) f - a; eroare de compilare
    }

    void foo () const { // metoda const
        int j = i + 22;
        // f = 33;          eroare de compilare
    }

    void bar () {
        f = 25;          // metoda nu e constanta, putem modifica
    }
};
```

Odată definită o clasă, putem declara obiecte constante de tipul clasei repective. Cu un obiect constant nu putem apela decât metode `const`.

```
int main () {
    const A a;
    // a.bar();   eroare de compilare
    a.foo();   // functioneaza
}
```

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

class B: public A {
public:
    B(int y=3){cout<<"B"<<4;}
};

class C: public B {
public:
    C ():B(10) {cout<<"C";}
};

int main () {
    A *pa = new C();
    delete pa;
    return 0;
}
```

```
main.cpp:12:5: error: constructor for 'B' must explicitly initialize the base class '
    A' which does not have a default constructor
    B(int y=3){cout<<"B"<<4;}
    ^
```

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

class C {
    int * const p;
public:
    C(int x) : p(&x) {(*p)+=3;}
    void set (int x) {p = &x;}
    friend ostream& operator<<(ostream& o, C x) {
        o << *x.p; return o;
    }
};

int main () {
    cout << C(3);
    return 0;
}
```

```
main.cpp:9:25: error: cannot assign to non-static data member 'p' with const-
    qualified type 'int *const'
    void set (int x) {p = &x;}
```

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

class A {
public:
    A () {cout << "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 () {
    B b; C c;
    return 0;
};
```

```
ABABC~C~B~A~B~A
```

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

struct A {
    static int i;
    A () {cout << "A"; i++;}
    ~A () {cout << "~A";}
};
int A::i = 0;
class B: public A {
public:
    B () {cout << "B"; i++;}
    ~B () {cout << "~B"; }
};

int main () {
    A a; B b;
    cout << a.i << " " << b.i;
    return 0;
};
```

```
AAB3 3~B~A~A
```

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

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

int main () {
    B b;
    return 0;
};
```

```
main.cpp:16:7: error: calling a private constructor of class 'B'
B b;
    ^
```

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

class B {
    B () {cout << "B";}
    ~B () {cout << "~B";}
    friend B foo ();
};
B foo () {
    return B();
}

int main () {
    B b = foo();
    return 0;
};
```

```
main.cpp:15:11: error: temporary of type 'B' has private destructor
B b = foo();
        ^
```

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

struct Base {
    Base (int x) { cout << "Base(" << x << ")"; }
    ~Base () {cout << "~Base";}
};

class Derived : Base {
public:
    Derived () { cout << "Derived"; }
    ~Derived () {cout << "~Derived"; }
};

int main () {
    return 0;
};
```

```
main.cpp:12:5: error: constructor for 'Derived' must explicitly initialize the base
    class 'Base' which does not have a default constructor
Derived () { cout << "Derived"; }
^
```

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

class A {
public:
    A () {cout << "A";}
    ~A () {cout << "~A";}
};
class B: public A {
public:
    B () {cout << "B";}
    ~B () {cout << "~B";}
};
class C: A, public B {
    A a;
public:
    C () {cout << "C";}
    ~C () {cout << "~C";}
};
int main () {
    C c;
    return 0;
};
```

```
AABAC~C~A~B~A~A
```