Curs 7 Moștenire, Polimorfism

- **Moștenire**
- **Polimorfism – Metode pur virtuale, Clase abstracte**
- **Operații de intrări ieșiri în C++**
  - **Fișiere**

Curs 6 Gestiunea memoriei in C++

- Alocare dinamica. Destructor. RAII. Rule of three.
- Tratarea excepțiilor – exception safe code
- Moștenire

De ce moștenire, polimorfism:

1. **Unele concepte din lumea reală au o natura ierarhică, aceste lucruri de multe ori se pot modela in program cel mai ușor creând ierarhii de clase folosind moștenire.**
   Ex. clasele din QT care modelează elemente de interfața grafica utilizator
2. **Dorim sa adăugam o nouă clasă in aplicație si avem deja o clasă existentă care are funcționalități comune cu ce dorim noi sa adăugăm.**
   O opțiune este sa folosim moștenire pentru a reutiliza partea de funcționalitate deja scrisă in clasa existentă si a adăuga doar partea care diferă in clasa fiu nou creată. Încercăm sa reprezentăm relația „is a", ex: FileRepository este („is a") Repository.
3. **Vrem sa scriem o parte a aplicației astfel încât sa fie ușor de extins ulterior cu noi funcționalități. Vrem sa tratam un set de obiecte de tipuri diferite in același fel, sa grupam clase diferite.**
   O sa scriem cod care funcționează doar cu interfața (metodele expuse) de la clasa de baza. Ulterior putem sa adăugam clase noi care moștenesc din clasa de baza si codul deja scris o sa funcționeze si cu aceste noi clase fără a rescriem nimic din ceea ce exista înainte.

Obs. De multe ori moștenirea/polimorfismul nu este singura soluție. In funcție de situație, capabilitățile limbajul de programare si natura aplicației/problemei alte alternative sunt posibile si cate-o data preferate.
Mecanisme/concepte care pot constitui alternative de luat in considerare:
Compoziție: In loc să moștenim dintr-o clasă putem folosi clasa (un câmp in clasa noua) si să refolosim astfel ceea ce exista deja scris.
Higher order functions (Funcții care primesc ca parametru alte funcții): pot fi o soluție pentru a crea cod generic, pentru a elimina duplicare de cod, pentru a oferi un punct de extensie in aplicație.
Duck typing : ajută in scrierea de cod general, care funcționează cu un set de obiecte de tipuri diferite
Template: Permite scrierea de metode/clase generice, similar cu duck typing in acest context.
Mixins, Aspects: Mecanisme existente in unele limbaje

Moștenire

Moștenirea permite definirea de clase noi (clase derivate) reutilizând clase existente (clasă de bază). Clasa nou creată moștenește comportamentul (metode) și caracteristicile (variabile membre, starea) de la clasa de bază

Dacă A și B sunt două clase unde B moștenește de la clasa A (B este derivat din clasa A sau clasa B este o specializare a clasei A) atunci:

- clasa B are toate metodele si variabilele membre din clasa A
- clasa B poate redefini metode din clasa A
- clasa B poate adaugă noi membrii (variabile, metode) pe lângă cele moștenite de la clasa A.

```
class Person {
public:
	Person(string cnp, string
name);
	const string& getName() const
{
		return name;
	}

	const string& getCNP() const
{
		return cnp;
	}
	string toString();
protected:
	string name;
	string cnp;

};
```

```
class Student: public Person {
public:
	Student(string cnp, string name,
string faculty);
	const string& getFaculty() const {
		return faculty;
	}
	string toString();
private:
	string faculty;
};
```

Moștenire simplă. Clase derivate.

Dacă clasa B moștenește de la clasa A atunci:

- orice obiect de tip B are toate variabilele mebre din clasa A
- funcțiile din clasa A pot fi aplicate si asupra obiectelor de tip B (daca vizibilitatea permite)
- clasa B poate adăuga variabile membre și sau metode pe lângă cele moștenite din A

```
class A:public B{
….
}
```

clasa B = Clasă de bază (superclass, base class, parent class)

clasa A = Clasă derivată (subclass, derived class, descendent class)

membrii (metode, variabile) moșteniți = membrii definiți în clasa A și nemodificați în clasa B

membrii redefiniți (overridden) = definit în A și în B (în B se crează o nouă definiție)

membrii adăugați = definiți doar în B

Vizibilitatea membrilor moșteniți

Dacă clasa A este derivat din clasa B:

- clasa A are acces la membri publici din B
- clasa A nu are acces la membrii privați din B

```
class A:public B{
….
}
```

public membrii publici din clasa B sunt publice și in clasa B

```
class A:private B{
…
}
```

private membrii publici din clasa B sunt private în clasa A

```
class A:protected B{
…
}
```

protected membrii publici din clasa B sunt protejate în clasa A (se vad doar in clasa A și în clase derivate din A).

Modificatori de acces

Definesc reguli de acces la variabile membre și metode dintr-o clasă

`public`: poate fi accesat de oriunde

`private`: poate fi accesat doar în interiorul clasei

`protected`: poate fi accesat în interiorul clasei și în clasele derivate.

`protected` se comportă ca și **`private`**, dar se permite accesul din clase derivate

Access	**`public`**	**`protected`**	**`private`**
clasa	Da	Da	Da
clasa derivată	Da	Da	Nu
În exterior	Da	Nu	Nu

Constructor/Destructor în clase derivate

- Constructorii și destructorii nu sunt moșteniți
- Constructorul din clasa derivată trebuie sa apeleze constructorul din clasa de baza. Sa ne asigurăm ca obiectul este inițializat corect.
- Similar și pentru destructor. Trebuie sa ne asiguram ca resursele gestionate de clasa de bază sunt eliberate.

```
Student::Student(string cnp, string name, string faculty) :
            Person(cnp, name) {
      this->faculty = faculty;
}
```

- Dacă nu apelăm explicit constructorul din clasa de bază, se apelează automat constructorul implicit
- Dacă nu exista constructor implicit se generează o eroare la compilare

```
Student::Student(string cnp, string name, string faculty) {
      this->faculty = faculty;
}
```

Se apelează destructorul clasei de bază

```
Student::~Student() {
      cout << "destroy student\n";
}
```

Inițializare.

Când definim constructorul putem inițializa variabilele membre chiar înainte sa se execute corpul constructorului.

```
Person::Person(string c, string n) :
           cnp(c), name(n) {
}
```

Initializare clasă de bază

```
Manager(std::string name, int yearInFirm, float payPerHour, float
bonus) :
                 Employee(name, yearInFirm, payPerHour) {
     this->bonus = bonus;
}
```

Apel metodă din clasa de bază

```
float Manager::payment(int hoursWorked) {
     float rez = Employee::payment(hoursWorked);
     rez = rez + rez * bonus;
     return rez;
}
```

Creare /distrugere de obiecte (clase derivate)

Creare

- se alocă memorie suficientă pentru variabilele membre din clasa de bază
- se alocă memorie pentru variabile membre noi din clasa derivată
- se apelează constructorul clasei de bază pentru a inițializa atributele din clasa de bază
- se execută constructorul din clasa derivată

Distrugere

- se apelează destructorul din clasa derivată
- se apelează destructorul din clasa de bază

Principiul substituției.

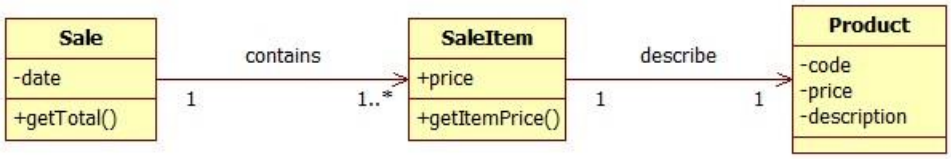
Un obiect de tipul clasei derivate se poate folosi în orice loc (context) unde se cere un obiect de tipul clasei de bază. (upcast implicit!)

```
Person p = Person("1", "Ion");
cout << p.toString() << "\n";

Student s("2", "Ion2", "Info");
cout << s.toString() << "\n";

Teacher t("3", "Ion3", "Assist");
cout << t.getName() << " " << t.getPosition() << "\n";

p = s;
cout << p.getName() << "\n";

p = t;
cout << p.getName() << "\n";

s = p;//not valid, compiler error
```

Pointer

```
Person *p1 = new Person("1", "Ion");
cout << p1->getName() << "\n";

Person *p2 = new Student("2", "Ion2", "Mat");
cout << p2->getName() << "\n";

Teacher *t1 = new Teacher("3", "Ion3", "Lect");
cout << t1->getName() << "\n";

p1 = t1;
cout << p1->getName() << "\n";

t1 = p1;//not valid, compiler error
```

Diagrame UML (Is a vs Has a)



- Un Sale are una sau mai multe SaleItem
- Un SaleItem are un Product

Relația de asociere UML (Associations): Descriu o relație de dependență structurală între clase

Elemente posibile:

- nume
- multiplicitate
- nume rol
- uni sau bidirecţional

Tipuri de relații de asociere

- Asociere
- Agregare (compoziție) (whole-part relation)
- Dependență
- Moștenire

Are (has a):

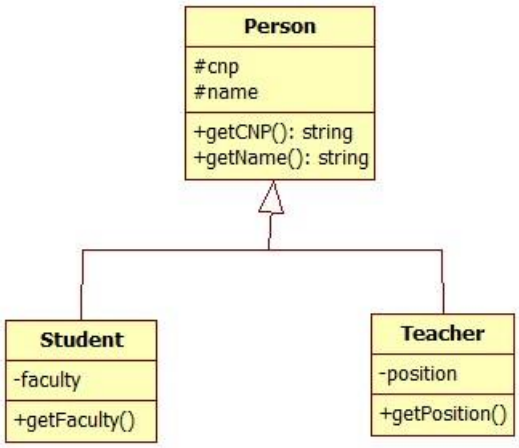
- Orice obiect de tip A are un obiect B.
- SaleItem are un Product. Persoana are nume (string)
- in cod apare ca şi o variabilă membră

Este ca şi (is a ,is like a):

- Orice instanță de tip A este şi de tip B
- Orice student este o persoană
- se implementează folosind moștenirea

Relația de specializare/generalizare – Reprezentarea UML .

Folosind moștenirea putem defini ierarhii de clase



Studentul este o Persoană cu câteva atribute adiționale
Studentul moștenește (variabile și metode) de la Persoană
Student este derivat din Persoană. Persoana este clasă de bază, Student este clasa derivată

Persoana este o generalizare a Studentului
Student este o specializare a Persoanei

Suprascriere (redefinire) de metode.

Clasa derivată poate redefini metode din clasa de bază

`string Person::toString() {`
`  return "Person:" + cnp + " " + name;`
`}`
	`string Student::toString() {`
	`  return "Student:" + cnp + " " + name + " " + faculty;`
	`}`

`Person p = Person("1", "Ion");` `cout << p.toString() << "\n";`	`Student s("2", "Ion2", "Info");` `cout << s.toString() << "\n";`

În clasa derivata descriem ce este specific clasei derivate, ce diferă față de clasa de bază

Suprascriere (overwrite) ≠ Supraîncărcare (overload)

```
string Person::toString() {
     return "Person:" + cnp + " " + name;
}

string Person::toString(string prefix) {
     return prefix + cnp + " " + name;
}
```

toString este o metodă supraîncărcată(**`toString()`**,**`toString(`**`string prefix)`**)**

Polimorfism

Proprietatea unor entități de:

- a se comporta diferit în funcție de tipul lor
- a reacționa diferit la același mesaj

Obiecte din diverse clase care sunt legate prin relații de moștenire să răspundă diferit la același mesaj (apel de metodă).

Proprietate a unui limbaj OO de a permite manipularea unor obiecte diferite prin intermediul unei interfețe comune

Tipul declarat vs tipul actual

Orice variabilă are un tip declarat (la declararea variabilei se specifică tipul).
În timpul execuției valoarea referită de variabila are un tip actual care poate diferi de tipul declarat

```
Student s("2", "Ion2", "Info");
Teacher t("3", "Ion3", "Assist");
Person p = Person("1", "Ion");

cout << p.toString() << "\n";

p = s; //slicing
cout << p.toString() << "\n";

p = t; //slicing
cout << p.toString() << "\n";
```

Tipul declarat pentru p este Persoană, dar în timpul execuției p are valori de tip Person, Student și Teacher.

Object slicing

Daca asignam un object de tipul clasei derivate la o variabila de tipul clasei de baza, obiectul pierde partea adăugată de clasa derivata.

Ex. p=s unde p e de tip `Person` si s e de tip `Student`: in acest caz p o sa conțină doar atribute din clasa `Person`.
Slicing se întâmpla la orice copiere de acest fel: assignment, transmitere prin valoare, return prin valoare.

Pentru a evita slicing se pot folosi referințe (&) sau pointeri (*).

```
string Person::toString() {
	return "Person:" + cnp + " " + name;
}
string Student::toString() {
	return "Student:" + cnp + " " + name + " " + faculty;
}
string Teacher::toString() {
	string rez = Person::toString();
	return "Teacher " + rez;
}
Student s("2", "Ion2", "Info");
Person* aux = &s;
cout << aux->toString() << "\n";
Person p = Person("1", "Ion");
aux = &p;
cout << aux->toString() << "\n";
```

- Person, Student,Teacher are metoda toString , fiecare clasă definește propria versiune de toString.
- Sistemul trebuie sa determine dinamic care dintre variante trebuie executată în momentul în care metoda toString este apelată.
- Decizia trebuie luată pe baza tipului actual al obiectului.
- Funcționalitate importantă (prezent în limbaje OO) numit legare dinamică - dynamic binding (late binding, runtime binding).

Legare dinamică (Dynamic binding).

Legarea (identificarea) codului de executat pe baza numelui de metode se poate face:

- în timpul compilării => legare statică (static binding)
- în timpul execuției => legare dinamică (dynamic binding)

Legare dinamică:

- selectarea metodei de executat se face timpul execuției.
- Când se apelează o metodă, codul efectiv executat (corpul funcției) se alege la momentul execuției (la legare statică decizia se ia la compilare)
- legarea dinamica în C++ funcționează doar pentru referințe și pointeri
- În C++ doar metodele virtuale folosesc legarea dinamică

Metode virtuale.

Legarea dinamică în c++: Folosind metode virtuale
O metodă este declarată virtual în casa de bază:
virtual <function-signature>

- metoda suprascrisă în clasele derivate are legarea dinamică activată
- metoda apelată se va decide în funcție de tipul actual al obiectului (nu în funcție de tipul declarat).

- Constructorul nu poate fi virtual – pentru a crea un obiect trebuie sa știm tipul exact
- Destructorul poate fi virtual (este chiar recomandat sa fie când avem ierarhii de clase)

```
class Person {
protected:
	string name;
	string cnp;

public:
	Person(string cnp, string name);
	virtual ~Person();

	const string& getName() const {
		return name;
	}

	const string& getCNP() const {
		return cnp;
	}
	virtual string toString();
	string toString(string prefix);
};
```

Mecanism C++ pentru polimorfism

Orice obiect are atașat informații legate de metodele obiectului

Pe baza acestor informații apelul de metodă este efectuat folosind implementarea corectă (cel din tipul actual). Orice obiect are referință la un tabel prin care pentru metodele virtuale se selectează implementarea corectă.

Orice clasă care are cel puțin o metodă virtuală (clasă polimorfică) are un tabel numit VTABLE (virtual table). VTABLE conține adrese la metode virtuale ale clasei.

Când invocăm o metodă folosind un pointer sau o referință compilatorul generează un mic cod adițional care în timpul execuției o sa folosească informația din VTABLE pentru a selecta metoda de executat.

Destructor virtual

- Destructorul este responsabil cu dealocarea resurselor folosite de un obiect
- Dacă avem o ierarhie de clasă atunci este de dorit să avem un comportament polimorfic pentru destructor (să se apeleze destructorul conform tipului actual)
- Trebuie să declarăm destructorul ca fiind virtual

Moștenire multiplă

În C++ este posibil ca o clasă să aibă multiple clase de bază, să moștenească de la mai multe clase

```
class Car : public Vehicle , public InsuredItem {

};
```

Clasa moștenește din toate clasele de bază toate atributele.
Moștenirea multipla poate fi periculoasă și în general ar trebui evitat

- se poate moșteni același atribut de la diferite clase
- putem avea clase de bază care au o clasă de bază comună

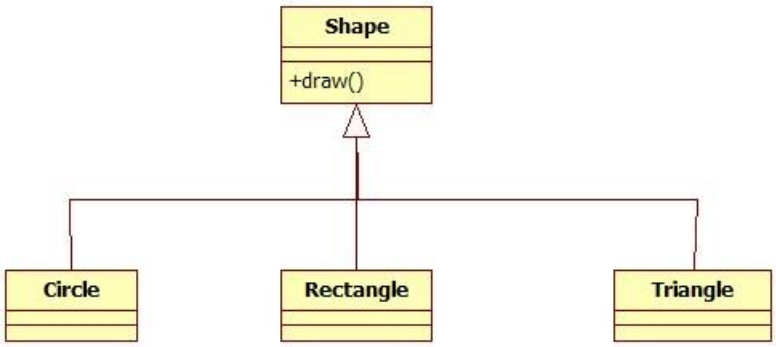
Funcții pur virtuale

Funcțiile pur virtuale nu sunt definite (avem doar declarația metodei). Folosim metode pur virtuale pentru a ne asigura că toate clasele derivate (concrete) o sa definească metoda.

```
class Shape {
public:
     Shape();
     virtual ~Shape();
     virtual void draw() = 0;//pure virtual
};
```

=0 indică faptul ca nu există implementare pentru această metodă în clasă. Clasele care au metode pur virtuale nu se pot instanția

Shape este o clasă abstractă - definește doar interfața, dar nu conține implementări.



Clase abstracte

O clasă abstractă poate fi folosită ca și clasă de bază pentru o colecție de clase derivate;
Oferă:

- o interfață comună pentru clasele derivate (metodele pur virtuale se vor implementa în clasele derivate)
- pot conține atribute comune tuturor claselor derivate

o clasă abstractă nu are instanțe
o clasă abstractă are cel puțin o metodă pur virtuală:
`virtual` <return-type> <name> (<parameters>) = 0;

clasă pur abstractă = clasă care are doar metode pur virtuale
clasă pur abstractă = interfață

În UML font italic

Clase care extind clase abstracte

- O clasă derivată dintr-o clasă abstractă moștenește interfața publică a clasei abstracte

- clasa suprascrie metodele definite în clasa abstractă, oferă implementări specifice pentru funcțiile definite în clasa abstracta

- putem avea instanțe

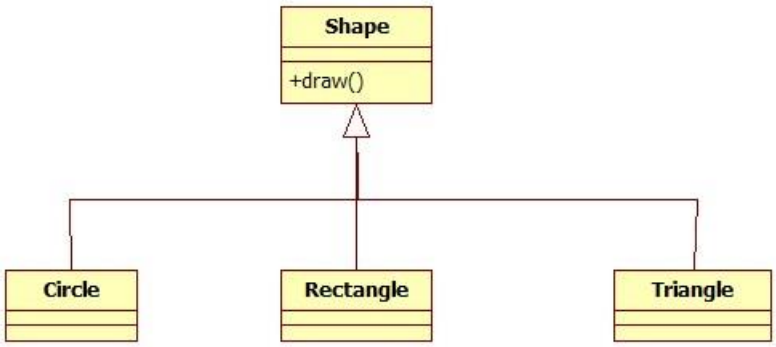
Funcții pur virtuale

Funcțiile pur virtuale nu sunt definite (avem doar declarația metodei).
Folosim metode pur virtuale pentru a ne asigura că toate clasele derivate (concrete) o sa definească metoda.

```
class Shape {
public:
     Shape();
     virtual ~Shape();
     virtual void draw() = 0;//pure virtual
};
```

=0 indică faptul ca nu există implementare pentru această metodă în clasă.
Clasele care au metode pur virtuale nu se pot instanția

Shape este o clasă abstractă - definește doar interfața, dar nu conține implementări.



Clase abstracte

O clasă abstractă poate fi folosită ca și clasă de bază pentru o colecție de clase derivate;
Oferă:

- o interfață comună pentru clasele derivate (metodele pur virtuale se vor implementa în clasele derivate)
- pot conține atribute comune tuturor claselor derivate

o clasă abstractă nu are instanțe

o clasă abstractă are cel puțin o metodă pur virtuală:
`virtual` <return-type> <name> (<parameters>) = 0;

clasă pur abstractă = clasă care are doar metode pur virtuale
clasă pur abstractă = interfață

În UML font italic

Clase care extind clase abstracte

- O clasă derivată dintr-o clasă abstractă moștenește interfața publică a clasei abstracte

- clasa suprascrie metodele definite în clasa abstractă, oferă implementări specifice pentru funcțiile definite în clasa abstracta

- putem avea instanțe

Moștenire. Polimorfism

De ce folosim moștenire:

Moștenire de implementare (Implementation Inheritance):

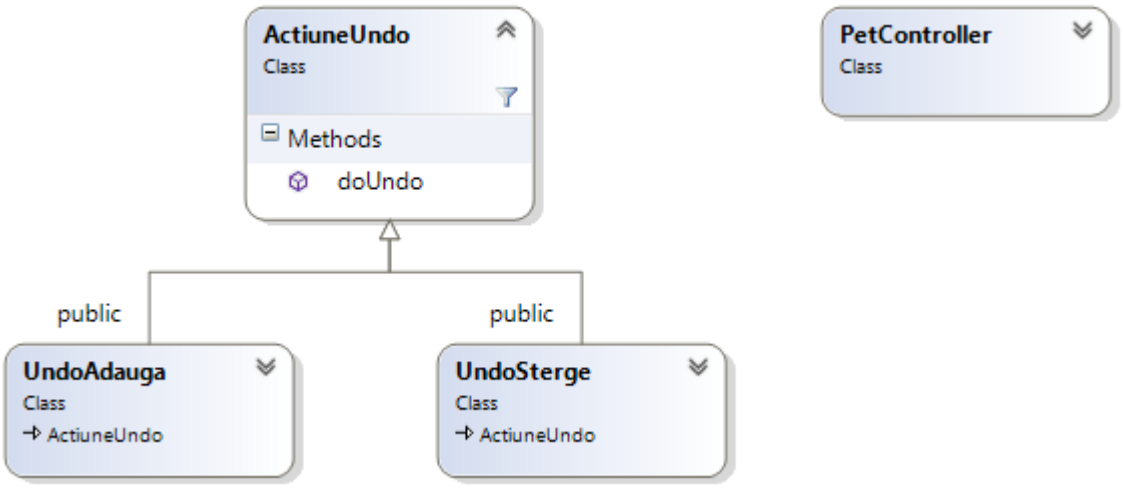
Clasa de baza oferă funcționalitate (metode, câmpuri) ce ușurează implementarea clasei derivate. Folosim moștenire pentru a reutiliza codul din clasa de baza (Ex. MemoryRepository -> FileRepository)

Moștenire de interfața (Interface Inheritance):

Obiectul de clasa derivata poate fi folosit oriunde se cere ceva de tipul clasei de baza (cele doua expun același interfața). Folosim pentru a oferi un punct de extensie (adăugare de funcționalitate noua fără a modifica codul existent). Ex: Undo

- **reutilizare de cod**
  - **clasa derivată moștenește din clasa de bază**
  - **se evită copy/paste – mai ușor de întreținut, înțeles**
- **extensibilitate**
  - **permite adăugarea cu ușurință de noi funcționalități**
  - **extindem aplicația fără să modificăm codul existent**

Exemplu : Undo




```
class ActiuneUndo {
public:
	virtual void doUndo() = 0;
	//destructorul e virtual pentru a ne asigura ca daca dau delete pe un
	// pointer se apeleaza destructorul din clasa care trebuie
	virtual ~ActiuneUndo(){};
};

class UndoAdauga : public ActiuneUndo {
	Pet petAdaugat;
	PetRepo& rep;
public:
	UndoAdauga(PetRepo& rep,const  Pet& p) :rep{ rep }, petAdaugat{ p } {}
	void doUndo() override {
		rep.sterge(petAdaugat);
	}
};
```

```
void PetController::add(const char* type, const char* species, int price) {
	Pet p{ type,species,price };
	repo.store(p);
	undoActions.push_back(new UndoAdauga{repo, p});
}

void PetController::undo() {
	if (undoActions.empty()) {
		throw PetException{"Nu mai exista operatii"};
	}
	 ActiuneUndo* act = undoActions.back();
	act->doUndo();  //nu e exception safe - vezi varianta cu unique_ptr
	undoActions.pop_back();
	delete act;
}
```

Operații de intrare/ieșire

IO (Input/Output) în C

<stdio.h> -> **scanf**(), **printf**(), **getchar**(), **getc**(), **putc**(), **open**(), **close**(), **fgetc**(), etc.

- Funțiile din C nu sunt extensibile
- funcționează doar cu un set limitat de tipuri de date (**char**, **int**, **float**, **double**).
- Nu fac parte din librăria standard => Implementările pot diferi (ANSI standard)
- pentru fiecare clasă nouă, ar trebui sa adăugăm o versiune nouă (supraîncărcare) de funcții **printf**() and **scanf**() și variantele pentru lucru cu fișiere, șiruri de caractere
- Metodele supraîncărcate au același nume dar o listă de parametrii diferit. Metodele printf și variantele pentru string, fișier folosesc o listă de argumente variabilă – nu putem supraîncărca.

Biblioteca de intrare/ieșire din C++ a fost creat să:

- fie ușor de extins
- ușor de adăugat/folosit tipuri noi de date

I/O streams. I/O Hierarchies of classes.

Iostream este o bibliotecă folosit pentru operații de intrări ieșiri in C++.
Este orientat-obiect și oferă operații de intrări/ieșiri bazat pe noțiunea de flux (stream)

iostream este parte din C++ Standard Library și conține un set de clase template și funcții utile in C++ pentru operații IO

Biblioteca standard de intrări/ieșiri (iostream) conține:

Clase template

O ierarhie de clase template, implementate astfel încât se pot folosi cu orice tip de date.

Instanțe de clase template

Biblioteca oferă instațe ale claselor template speciale pentru manipulare de caractere **char** (narrow-oriented) respectiv pentru elemente de tip wchar (wide-oriented).

Obiecte standard

În fișierul header **<iostream>** sunt declarate obiecte care pot fi folosite pentru operații cu intrare/ieșire standard.

Tipuri

conține tipuri noi, folosite biblioteca standard, cum ar fi: streampos, streamoff and streamsize (reprezintă poziții, offset, dimensiuni)

Manipulatori

Funții globale care modifică proprietăți generale, oferă informații de formatare pentru streamuri**.**

Streamuri standard – definite în <iostream>

cin - corespunde intrării standard (stdin), este de tip **istream**

cout - corespunde ieșirii standard (stdout) , este de tip **ostream**

cerr - corespunde ieșirii standard de erori (stderr), este de tip **ostream**

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

void testStandardIOStreams() {
	cout << "!!!Hello World!!!" << endl;
	// prints !!!Hello World!!! to the console
	int i = 0;
	cin >> i; //read an int from the console
	cout << "i=" << i << endl; // prints !!!Hello World!!! to the console

    cerr << "Error message";//write a message to the standard error stream
}
```

Operatorul de inserție (Insertion operator - output)

- Pentru operațiile de scriere pe un stream (ieșire standard, fișier, etc) de folosește operatorul “<<“, numit operator de inserție

- pe partea stângă trebuie sa avem un obiect de tip ostream (sau derivat din ostream). Pentru a scrie pe ieșire standard (consolă) se folosește **cout** (declarat in modulul iostream)

- pe dreapta putem avea o expresie.

- Operatorul este supraîncărcat pentru tipurile standard, pentru tipurile noi programatorul trebuie sa supraîncarce.

```
void testWriteToStandardStream() {
	cout << 1 << endl;
	cout << 1.4 << endl << 12 << endl;
	cout << "asdasd" << endl;
	string a("aaaaaaaa");
	cout << a << endl;

	int ints[10] = { 0 };
	cout << ints << endl; //print the memory address
}
```

- Se pot înlănțui operații de inserție, evaluarea se face în ordinea inversă scrierii. Înlănțuirea funcționează fiindcă operatorul << returnează o referință la stream

Operatorul de extragere – citire (Extraction operator)

- Citirea dintr-un stream se realizează folosind operatorul “>>“
- operandul din stânga trebuie să fie un obiect de tip istream (sau derivat din istream). Pentru a citi din intrarea standard (consolă) putem folosi cin, obiect declarat în iostream
- operandul de pe dreapta poate fi o expresie, pentru tipuri standard operatorul de extragere este supraîncărcat.
- programatorul poate supraîncărca operatorul pentru tipuri noi.

```
void testStandardInput() {
	int i = 0;
	cout << "Enter int:";
	cin >> i;
	cout << i << endl;
	double d = 0;
	cout << "Enter double:";
	cin >> d;
	cout << d << endl;
	string s;
	cin >> s;
	cout << s << endl;
}
```

Fișiere

Pentru a folosi fișiere in aplicații C++ trebuie sa conectăm streamul la un fișier de pe disk

fstream oferă metode pentru citire/scriere date din/in fișiere.

<fstream.h>

- ifstream (input file stream)
- ofstream (output file stream)

Putem atașa fișierul de stream folosind constructorul sau metoda open

După ce am terminat operațiile de IO pe fișier trebuie sa închidem (dezasociem) streamul de fișier folosind metoda close. Ulterior, folosind metoda **open,** putem folosi streamul pentru a lucra cu un alt fișier.

Metoda is_open se poate folosi pentru a verifica daca streamul este asociat cu un fișier.

Output File Stream

```
#include <fstream>

void testOutputToFile() {
  ofstream out("test.out");
  out << "asdasdasd" << endl;
  out << "kkkkkkk" << endl;
  out << 7 << endl;
  out.close();
}
```

- Dacă fișierul “test.out” există pe disc, se deschide fișierul pentru scriere și se conectează streamul la fișier. Conținutul fișierului este șters la deschidere.

- Daca nu există “test.out”: se creează, se deschide fișierul pentru scriere și se conectează streamul la fișier.

Input File Stream

```
void testInputFromFile() {
     ifstream in("test.out");
    //verify if the stream is opened
      if (in.fail()) {
           return;
     }
     while (!in.eof()) {
           string s;
           in >> s;
           cout << s << endl;
     }
     in.close();
}
```

```
void testInputFromFileByLine() {
     ifstream in;
     in.open("test.out");
    //verify if the stream is opened
     if (!in.is_open()) {
           return;
     }
     while (in.good()) {
           string s;
           getline(in, s);
           cout << s << endl;
     }
     in.close();
}
```

- Dacă fișierul “test.out” există pe disc, se deschide pentru citire și se conectează streamul la fișier.
- Dacă nu există fișierul streamul nu se asociază, nu se poate citi din stream
- Unele implementări de C++ creează fișier dacă acesta nu există.

Open

Funcția open deschide fișierul și asociează cu stream-ul:

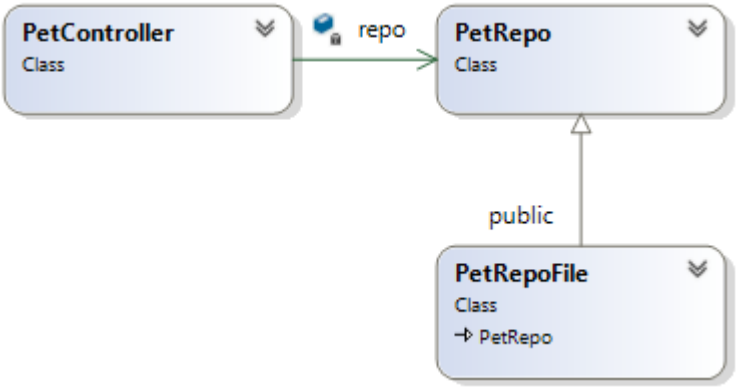
open (filename, mode);

filename sir de caractere ce indică fișierul care se deschide
mode parametru opțional, indică modul în care se deschide fișierul. Poate fo o combinație dintre următoarele flaguri:

ios::in	Deschide pentru citire.
ios::out	Deschide pentru scriere.
ios::binary	Mod binar.
ios::ate	Se poziționează la sfârșitul fișierului. Dacă nu e setat, după deschidere se poziționează la început.
ios::app	Toate operațiile de scriere se efectuează la sfârșitul fișierului, se adaugă la conținutul existent. Poate fi folosit doar pe stream-uri deschise pentru scriere.
ios::trunc	Șterge conținutul existent.

Flag-urile se pot combine folosind operatorul pe biți OR (|).

Salvare date in fisier - FileRepository




```
class PetRepoFile :public PetRepo {
private:
	std::string fName;
	void loadFromFile();
	void writeToFile();
public:
	PetRepoFile(std::string fName) :PetRepo(), fName{ fName } {
		loadFromFile();//incarcam datele din fisier
	}
	void store(const Pet& p) override {
		PetRepo::store(p);//apelam metoda din clasa de baza
		writeToFile();
	}
	void sterge(const Pet& p) override {
		PetRepo::sterge(p);//apelam metoda din clasa de baza
		writeToFile();
	}
};
```

```
#include <fstream>
void PetRepoFile::loadFromFile(){
   std::ifstream in(fName);
    if (!in.is_open()) {
      //verify if the stream is opened

      throw PetException("Error open:"+fName);
    }
    while (!in.eof()) {
       std::string species;
       in >> species;
        //poate am linii goale
       if (in.eof()) break;
       std::string type;
       in >> type;
       int price;
       in >> price;

   Pet p{type.c_str(),species.c_str(), price};
       PetRepo::store(p);
    }
   in.close();
}
```

```
void PetRepoFile::writeToFile() {
	std::ofstream out(fName);
	if (!out.is_open()) { /
	  //verify if the stream is opened
	  std::string msg("Error open file");
	  throw PetException(msg);
	}
	for (auto& p:getAll()) {
		out << p.getSpecies();
		out << std::endl;
		out << p.getType();
		out << std::endl;
		out << p.getPrice();
		out << std::endl;
	}
	out.close();
}
```

Erori la citire/scriere - Flag-uri

Indică starea internă a unui stream:

Flag	**Descriere**	**Metodă**
fail	Date invalide	fail()
badbit	Eroare fizică	bad()
goodbit	OK	good()
eofbit	Sfârșid de stream detectat	eof()

```
void testFlags(){
	cin.setstate(ios::badbit);
	if (cin.bad()){
		//something wrong
	}
}
```

Flag de control :

```
cin.setf(ios::skipws);  //Skip white space. (For input; this is the default.)
cin.unsetf(ios::skipws);
```

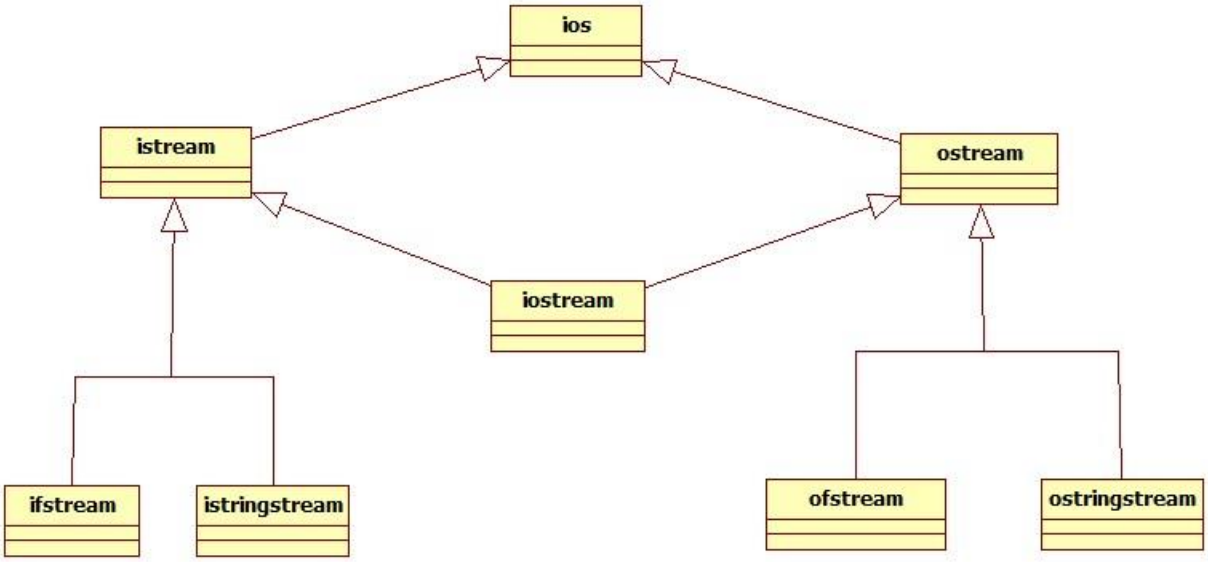
```
/*
  Citeste date de la consola (int,float, double, string, etc)
  Reia citirea pana cand utilizatorul introduce corect
*/
template<typename T>
T myread(const char* msg) {
	T cmd;
	while (true) {
		std::cout<<std::endl << msg;
		std::cin >> cmd;
		bool fail = std::cin.fail();
		std::cin.clear();//resetez failbit
		auto& aux = std::cin.ignore(1000, '\n');
		if (!fail && aux.gcount()<=1) {
			break; //am reusit sa citesc numar
		}
	}
	return cmd;
}
```

Formatare scriere

width(int x)	Numărul minim de caractere pentru scrierea următoare
fill(char x)	Caracter folosit pentru a umple spațiu dacă e nevoie să completeze cu caractere (lungime mai mică decât cel setat folosind width).
precision(int x)	Numărul de zecimale scrise

```
void testFormatOutput() {
	cout.width(5);
	cout << "a";
	cout.width(5);
	cout << "bb" << endl;
	const double PI = 3.1415926535897;
	cout.precision(3);
	cout << PI << endl;
	cout.precision(8);
	cout << PI << endl;
}
/*
Tipareste lista de pet
*/
void PetUI::printPets(const std::vector<Pet>& v) {
	std::cout << "\n Pets("<<v.size()<<"):\n";
	printTableHeader();
	for (const Pet& p : v) {
		std::cout.width(10);
		std::cout << p.getType();
		std::cout.width(20);
		std::cout << p.getSpecies();
		std::cout.width(10);
		std::cout << p.getType();
		std::cout << std::endl;
	}
	std::cout << std::endl;
}
void printTableHeader() {
	std::cout.width(10);
	std::cout << "Type";
	std::cout.width(20);
	std::cout << "Species";
	std::cout.width(10);
	std::cout << "Price";
	std::cout << std::endl;
}
```

Hierarhie de clase din biblioteca standard IO C++



Clasele folosite pentru intrări/ieșiri sunt definite în fișiere header:

- <ios> formatare , streambuffer.
- <istream> intrări formatate
- <ostream> ieșiri formatate
- <**iostream**> implementează intrări/ieșiri formatate
- <fstream> intrări/ieșiri fișiere.
- <sstream> intrări/ieșiri pentru streamuri de tip string.
- <iomanip> conține manipulatori.
- <iosfwd> declarații pentru toate clasele din biblioteca IO.

Supraîncărcare operatori <<, >> pentru tipuri utilizator

- Se face similar ca și pentru orice operator
- sunt operatori binari
- primul operand este un stream (pe stânga), pe partea dreaptă avem un obiect de tipul nou (user defined).

```
class Product {
public:
  Product(int code, string desc,
double price)
 ~Product();
      double getCode() const {
            return code;
      }
      double getPrice() const {
            return price;
      }

friend ostream& operator<< (ostream&
stream, const Product& prod);

private:
      int code;
      string description;
      double price;
};
```

```
ostream& operator<<(ostream&
stream, const Product& prod) {
  stream << prod.code <<" ";
  stream << prod.description <<" ";
  stream  << prod.price;
  return stream;
}
```

```
void testStandardOutputUserType() {
      Product p = Product(1, "prod", 21.1);
      cout << p << "\n";
      Product p2 = Product(2, "prod2", 2.4);
      cout << p2 << "\n";
}
```

```
void testInsertionOperator() {
      Product p{ 1,"prod1",100 };
      std::ostringstream os;
      os << p;
      assert(os.str() == "1 prod1 100");
}
```

Manipulatori.

- Manipulatorii sunt funcții cu semantică specială, folosite împreună cu operatorul de inserare/extragere (<< , >>)

- Sunt funcții obișnuite, se pot și apela (se da un argument de tip stream)

- Manipulatorii se folosesc pentru a schimba modul de formatare a streamului sau pentru a insera caractere speciale.

- Există o variabilă membră în ios (x_flags) care conține informații de formatare pentru operare I/O , x_flags poate fi modificat folosind manipulatori

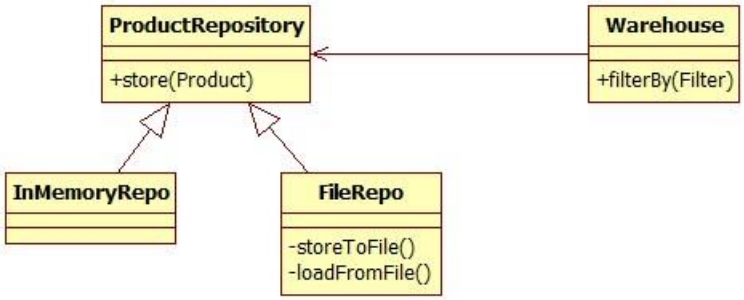
- sunt definite în modulul **iostream.h** (endl, dec, hex, oct, etc) și **iomanip.h** (setbase(int b),setw(int w),setprecision(int p))

```
void testManipulators() {
     cout << oct << 9 << endl << dec << 9 << endl;
     oct(cout);
     cout << 9;
     dec(cout);
}
```

Citire/scriere obiecte

```
void testWriteReadUserObjFile() {
	ofstream out;
	out.open("test2.out", ios::out | ios::trunc);
	if (!out.is_open()) {
		return;
	}
	Product p1(1, "p1", 1.0);
	out << p1 << endl;
	Product p2(2, "p2", 2.0);
	out << p2;
	out.close();

	//read
	ifstream in("test2.out");
	if (!in.is_open()) {
		cout << "Unable to open";
		return;
	}
	Product p(0, "", 0);
	while (!in.eof()) {
		in >> p;
		cout << p << endl;
	}
	in.close();
}
```

Exemplu: FileRepository



Varianta 2

Flux - Stream

Noțiunea de flux este o abstractizare, reprezintă orice dispozitiv pe care executăm operații de intrări / ieșiri (citire/scriere)

Stream este un flux de date de la un set de surse (tastatură, fișier, zonă de memorie) către un set de destinații (ecran, fișier, zonă de memorie)

În general fiecare stream este asociat cu o sursă sau destinație fizică care permite citire/scriere de caractere.

De exemplu: un fișier pe disk, tastatura, consola. Caracterele citite/scrise folosind streamuri ajung / sunt preluate de la dispozitive fizice existente (hardware).
Stream de fișiere - sunt obiecte care interacționează cu fișiere, dacă am atașat un stream la un fișier orice operație de scriere se reflectă în fișierul de pe disc.

Buffer

buffer este o zonă de memorie care este un intermediar între stream și dispozitiv.

De fiecare dată când se apelează metoda put (scrie un caracter), caracterul nu este trimis la dispozitivul destinație (ex. Fișier) cu care este asociat streamul. Defapt caracterul este inserat în buffer

Când bufferul este golit (flush) , toate datele se trimit la dispozitiv (daca era un stream de ieșire). Procesul prin care conținutul bufferului este trimis la dispozitiv se numește sincronizare și se întâmplă dacă:

- streamul este închis, toate datele care sunt în buffer se trimit la dispozitiv (se scriu in fișier, se trimite la consolă, etc.)
- bufferul este plin. Fiecare buffer are o dimensiune, dacă se umple atunci se trimite conținutul lui la dispozitiv
- programatorul poate declanșa sincronizarea folosind manipulatoare: **flush**, **endl**
- programatorul poate declanșa sincronizarea folosind metoda **sync()**

Fiecare obiect stream din biblioteca standard are atașat un buffer (**streambuf**)