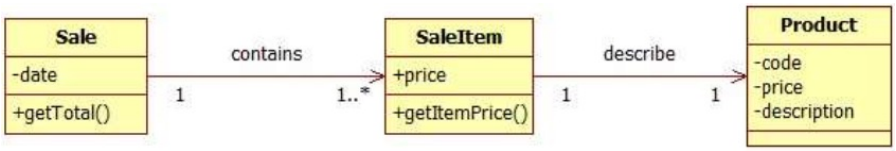
Diagrame UML



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

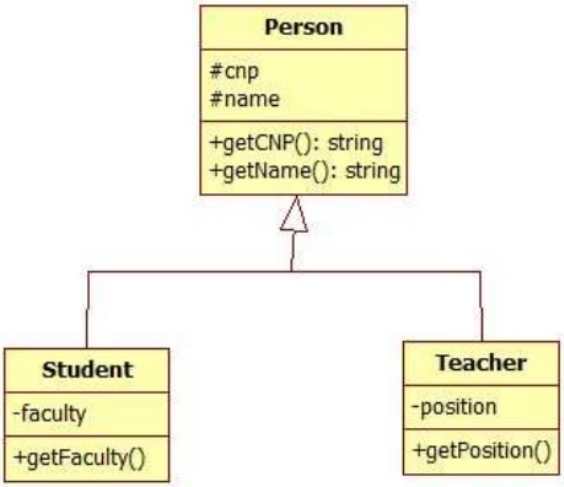
Relatia de asociere UML : Descriu o relatie de dependenta intre clase
Elemente posibile :

- nume
- multiplicitate
- nume rol
- unidirectional sau bidirectional

Tipuri de relatii de asociere

- Asociere
- Agregare
- Dependenta
- Mostenire

Relatia de specializare/generalizare
Folosind mostenirea putem defini ierarhii de clase



Studentul este o Persoana cu cateva atribute aditionale.
Studentul mosteneste (variabile si metode) de la Persoana.
Student este derivat din Persoana. Persoana este clasa de baza, Student este clasa derivata.
Persoana este o generalizare a Studentului, iar Student este o speializare a Persoanei.

Suprascriere (redefinire) de metode
Clasa derivata poate redefini metode din clasa de baza :

`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";`

In clasa derivata descriem ce este specific clasei derivate, ce difera fata de clasa de baza.

Suprascriere (overwrite) ≠ Supraincarcare (overload)

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

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

toString este o metoda supraincarcata deoarece este o functie cu parametrii diferiti dar cu acelasi nume.

Polimorfism

Este proprietatea unor entitati de a se comporta diferit in functie de tipul lor.

Tipul declarat vs tipul actual

Orice variabila are un tip declarat (la declararea variabilei se specifica tipul). In timpul executiei valoarea referita 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 Person, dar in timpul executiei p are valori de tip Person, Student si Teacher.

Object slicing

Daca avem o clasa de baza (Person) si o clasa derivata (Student) care mosteneste Person, iar tu copiezi un obiect Student intr-o variabila de tip Person, atunci se pierde partea specifica clasei Student. Asta se numeste slicing.

```
class Person {
public:
    std::string name;
};

class Student : public Person {
public:
    int grade;
};

int main() {
    Student s;
    s.name = "Ana";
    s.grade = 10;

    Person p = s; // slicing: se copiază DOAR partea de Person

    std::cout << p.name << std::endl; // OK
    // p.grade nu mai există aici! S-a pierdut!
}
```

Pentru a evita slicing se pot folosi referinte sau pointeri.

```
void afiseaza(const Person& p) {
    std::cout << p.name << std::endl;
}

int main() {
    Student s;
    s.name = "Ana";
    s.grade = 10;

    afiseaza(s); // Nu se pierde nimic, pentru că nu se face copiere
}
```

Legare dinamica vs Legare statica
Legarea dinamica reprezinta mecanismul prin care in momentul executiei programului, se decide care functie (metoda) va fi efectiv apelata, atunci cand exista mai multe variante posibile (de exemplu, in cazul mostenirii si a functiilor virtuale in C++)

Diferenta fata de legarea statica :
- Legare statica : Alegerea metodei se face la compilare, pe baza tipului cunoscut al obiectului.
- Legare dinamica : Alegerea metodei se face la rulare, pe baza tipului real al obiectului.

Metode virtuale
O metoda virtuala in C++ este o functie a unei clase care este declarata cu cuvantul cheie virtual si care permite redefinirea comportamentului ei in clasele derivate. Prin utilizarea metodelor virtuale, C++ permite polimorfismul - adica posibilitatea ca apelul unei metode sa execute comportamentul specific clasei reale a obiectului, chiar daca obiectul este accesat printr-un pointer sau referinta la clasa de baza.

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

class Forma {
public:
    virtual void deseneaza() {  // metodă virtuală
        cout << "Desenez o formă generică\n";
    }
};

class Cerc : public Forma {
public:
    void deseneaza() override {  // suprascrie metoda virtuală
        cout << "Desenez un cerc\n";
    }
};

int main() {
    Forma* f = new Cerc();  // pointer la clasa de bază, dar obiect Cerc
    f->deseneaza();         // Se apelează Cerc::deseneaza, nu Forma::deseneaza
    delete f;
}
```

Mecanism C++ pentru polimorfism
Orice obiect are atasat informatii legate de metodele obiectului. Pe baza acestor informatii apelul de metoda este efectuat folosind implementarea corecta (cel din tipul actual). Orice obiect are referinta la un tabel prin care pentru metodele virtuale se selecteaza implementarea corecta.

Orice clasa care are cel putin o metoda virtuala (clasa polimorfica) are un tabel numit virtual table (VTable). VTable contine adrese la metode virtuale ale clasei.

Cand invocam o metoda folosind un pointer sau o referinta compilatorul genereaza un mic cod aditional care in timpul executiei o sa foloseasca infromatia din VTable pentru a selecta metoda de executat.

Destructor virtual
- Destructorul este responsabil cu dealocarea resurselor folosite de un obiect
- Daca avem o ierarhie de clasa atunci este de dorit sa avem un comportament polimorfic pentru destructor (sa se apeleze destructorul conform tipului actual)
- Trebuie sa declaram destructorul ca fiind virtual

Mostenire multipla
In C++ este posibil ca o clasa sa aiba multiple clase de baza adica sa mostenească de la mai multe clase

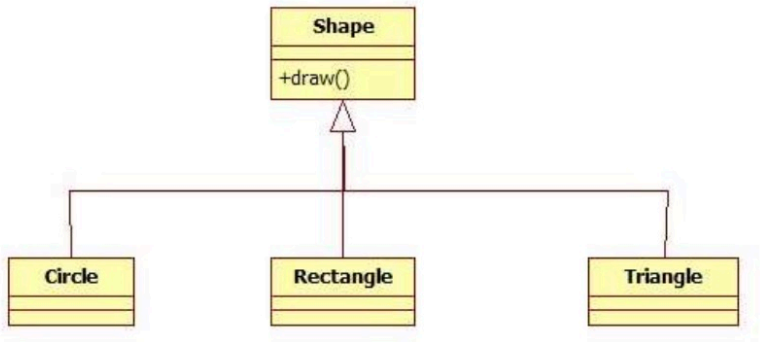
```cpp
class Car : public Vehicle , public InsuredItem {

};
```

Functii pur virtuale

Functiile pur virtuale nu sunt definite (avem doar declaratia metodie). Folosim metode pur virtuale pentru a ne asigura ca toate clasele derivata o sa defineasca metoda.

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

= 0 indica faptul ca nu exista implementare pentru aceasta metoda in clasa.
Clasele care au metode pur virtuale nu se pot instantia.
Shape este o clasa abstracta care defineste doar interfata, dar nu contine implementari.



Clase abstracte

O clasa abstracta poate fi folosita ca si clasa de baza pentru o colectie de clase derivate.
Oferta :

- O interfata comuna pentru clasele derivate (metodele pur virtuale se vor implementa in clasele derivate)
- Pot contine atribute comune tuturor claselor derivate.

O clasa abstracta nu are instante. O clasa abstracta are cel putin o metoda pur virtuala. Clasa pur abstracta = clasa care are doar metode pur virtuale.

Clase care extind clase abstracte

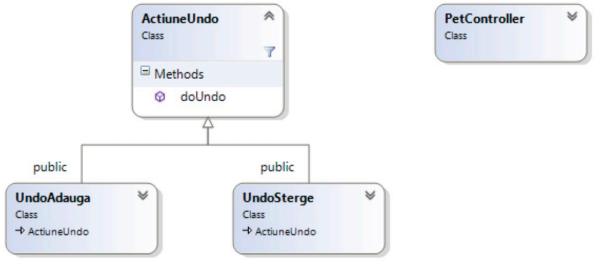
- O clasa derivata dintr-o clasa abstracta mosteneste interfata publica a clasei abstracte
- Clasa suprascrie metodele definite in clasa abstracta, ofera implementari specifice pentru funcitile definite in clasa abstracta
- Putem avea instante

Mostenire. Polimorfism

De ce folosim mostenire?
Clasa de baza ofera functionalitate (metode, campuri) ce usureaza implementarea clasei derivate.
Folosim mostenire pentru a reutiliza codul din clasa de baza (Ex : MemoryRepository -> FileRepository)

- reutilizare de cod, clasa derivata mosteneste din clasa de baza, se evita copy/paste si este mai usor de intretinut si inteles
- extensibilitate, permite adaugarea cu usurinta de noi functionalitati, extindem aplicatia fara sa modificam codul existent

Exemplu : Undo



```cpp
// Clasa de bază (abstractă)
class ActiuneUndo {
public:
    virtual void doUndo() = 0;  // metodă pur virtuală
    virtual ~ActiuneUndo() = default;
};

// Clasa derivată: Undo pentru adăugare
class UndoAdauga : public ActiuneUndo {
    string pet;
public:
    UndoAdauga(const string& pet) : pet(pet) {}
    void doUndo() override {
        cout << "Undo adaugare: " << pet << endl;
        // aici ai scrie logica de a elimina "pet" din listă
    }
};

// Clasa derivată: Undo pentru ștergere
class UndoSterge : public ActiuneUndo {
    string pet;
public:
    UndoSterge(const string& pet) : pet(pet) {}
    void doUndo() override {
        cout << "Undo stergere: " << pet << endl;
        // aici ai scrie logica de a reintroduce "pet" în listă
    }
};
```

Operatii de intrare/iesire

IO (Input/Output) in C

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

- Functioneaza doar cu un set limitat de tipuri de date (**char**, **int**, **float**, **double**).
- Nu fac parte din libraria standard.
- Pentru fiecare clasa noua, ar trebui sa adaugam o versiune noua de functii **printf**() si **scanf**().

Iostream este o biblioteca folosita pentru operatii de intrari si iesiri in C++. Este orientat-obiect si ofera operatii de intrari/iesiri bazat pe notiunea de flux (stream).

Streamuri standard definite in <iostream>

cin - corespunde intrarii standard (stdin), este de tip istream
cout - corespunde iesirii standard (stdout), este de tip ostream
cerr - corespunde iesirii standard de erori (stderr), este de tip ostream

```cpp
#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 insertie

- Pentru operatiile de scriere pe un stream se foloseste operatorul "<<", numit operator de insertie.
- Pe partea stanga a operatorului trebuie sa avem un obiect de tip ostream sau derivat din ostream. Pentru a scrie pe iesire standard (consola) se foloseste **cout** (declarat in ostream).
- Pe dreapta putem sa avem o expresie.
- Operatorul este supraincarcat pentru tipurile standard, pentru tipurile noi programatorul trebuie sa supraincarce.

```cpp
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
}
```

Operatorul de extragere - citire

- Citirea dintr-un stream se realizeaza folosind operatorul ">>".
- Operandul din stanga trebuie sa fie un obiect de tip istream sau derivat din istream. Pentru a citi din intrarea standard (consola) putem folosi **cin** (declarat in istream).
- Operandul din dreapta poate sa fie o expresie.
- Programatorul poate supraincarca 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;
}
```

Fisiere

Pentru a folosi fisiere in aplicatii C++ trebuie sa conectam streamul la un fisier pe disk. Biblioteca **fstream** ofera metode pentru citire/scriere date din/in fisiere.
<fstream.h>

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

Putem atasa fisierul de stream folosind constructorul sau metoda **open**. Dupa ce am terminat operatiile de IO pe fisier trebuie sa inchidem streamul de fisier folosind metoda **close**. Ulterior, folosind metoda **open**, putem folosi streamul pentru a lucra cu un alt fisier.
Metoda **is_open** se poate folosi pentru a verifica daca streamul este asociat cu un fisier.

Output File Stream

```
#include <fstream>

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

- Daca fisierul "test.out" exista pe disc, se deschide fisierul pentru scriere si se conecteaza streamul la fisier. Continutul fisierului este sters la deschidere.
- Daca nu exista "test.out" pe disc, atunci se creeaza, se deschide fisierul pentru scriere si se conecteaza streamul la fisier.

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();
}
```

- Daca fisierul "test.out" exista pe disc, se deschide pentru citire si se conecteaza streamul la fisier.
- Daca nu exista fisierul, atunci streamul nu se asociaza si nu se poate citi din stream.
- Unele implementari de C++ creeaza fisierul daca acesta nu exista.

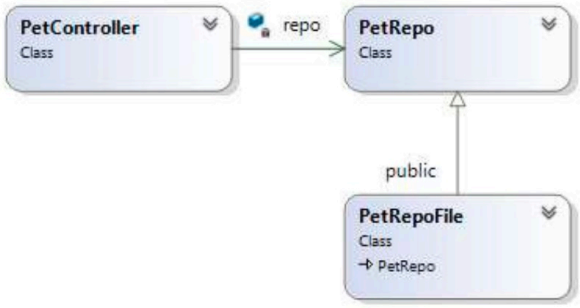
Open

Functia open deschide fisierul si asociaza streamul : **open**(filename, mode);

- **filename** este un sir de caractere ce indica fisierul care se deschide.
- **mode** este parametru optional si indica modul in care se deschide fisierul. Poate fi o combinatie dintre urmatoarele flaguri :
  ios::in → deschide pentru citire
  ios::out → deschide pentru scriere
  ios::binar → mod binar
  ios::ate → se pozitioneaza la sfarsitul fisierului, daca nu este setat atunci dupa deschidere se va pozitiona la inceput
  ios:app → toate operatiile de scriere se efectueaza la sfarsitul fisierului, se adauga la continutul existent. Poate fi folosit doar pe stream-uri deschide pentru scriere
  ios::trunc → sterge continutul existent

Flagurile se pot combina folosind operatorul pe biti 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

Indica starea interna a unui stream :

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

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

```
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

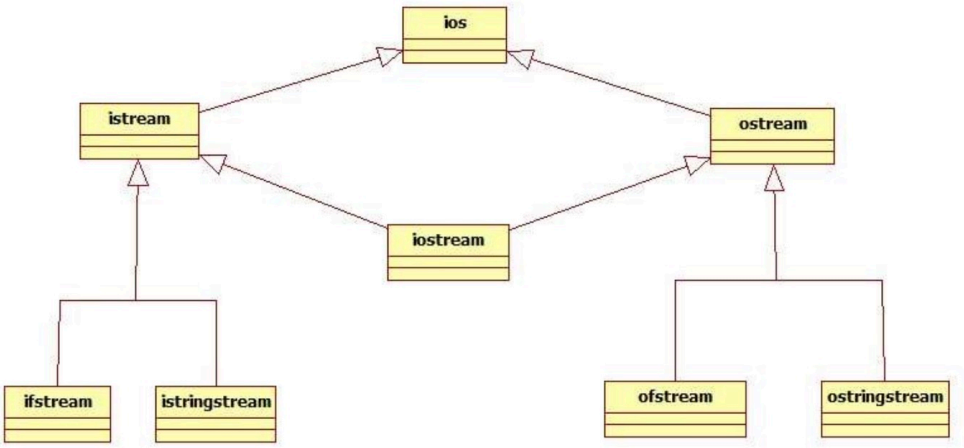
width(int x) → numarul minim de caractere pentru scrierea urmatoare.

fill(int x) → caracter folosit pentru a umple spatiu daca e nevoie sa completeze cu caractere (lungime mai mica decat cel setat folosind width).

precision(int x) → numarul 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++



Supraincarcare operatori << , >> pentru tipuri utilizator

- Se face similar ca si pentru orice operator.

```
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;
}
```

Spatii de nume

Introduc un domeniu de vizibilitate care nu poate contine duplicate

```
namespace testNamespace1 {
class A {
};
}
namespace testNamespace2 {
class A {
};
}
```

Accesul la elementele unui spatiu de nume se face folosind operatorul de rezolutie

```
void testNamespaces() {
      testNamespace1::A a1;
      testNamespace2::A a2;
}
```

Folosind directiva using putem importa :

1. Toate elementele definite intr-un spatiu de nume

```
void testUsing() {
      using namespace testNamespace1;
      A a;
}
```

2. Doar clasa/metoda pe care vrem sa o folosim

```
using std::vector;
using std::copy_if;
using std::cout;
```

Flux - Stream
Flux este o abstractizare, reprezinta orice dispozitiv pe care executam operatii de intrari / iesire.
Stream este un flux de date de la un set de surse (tastatura, fisier, zona de memorie) catre un set de destinatii (ecran, fisier, zona de memorie)

In general fiecare stream este asociat cu o sursa sau o destinatie fizica care permite citire/scriere de caractere.

Exemplu :
Un fisier pe disk, tastatura, consola. Caracterele citite/scrise folosind streamuri ajung / sunt preluate de la dispozitive fizice existente (hardware). Stream de fisiere - sunt obiecte care interactioneaza cu fisiere, daca am atasat un stream la un fisier orice operatie de scriere se reflecta in fisierul de pe disc.

Buffer
Este o zona de memorie care este un intermediar intre un dispozitiv de intrare/iesire (fisier, consola) si un stream. In termeni simpli, este o locatie unde datele sunt stocate temporar inainte de a fi transmise efectiv la destinatie.

1. Scrierea intr-un buffer :
Cand un program scrie un caracter intr-un stream, de obicei, acesta nu ajunge imediat la dispozitivul tinta (fisier sau consola). In schimb caracterul este plasat intr-un buffer, iar datele nu sunt scrise pe loc.

2. Gasirea unui moment pentru a trimite datele :
Datele din buffer sunt transmise la dispozitiv doar atunci cand :
- Streamul este inchis : tot continutul ramas in buffer se trimite la dispozitiv.
- Bufferul se umple : fiecare buffer are o dimensiune limitata, iar cand se umple datele sunt trimise automat la dispozitiv.
- Sincronizarea manuala : programatorul poate forta bufferul sa fie golit folosind metode ca flush(), endl sau sync().

3. De ce se foloseste bufferul?
Performanta crescuta, bufferul ajuta la imbunatatirea performantei, deoarece scrierea in memorie este mai rapida decat scrierea directa pe un dispozitiv de stocare.

4. Tipuri de streamuri
Fiecare obiect stream din biblioteca C++ are asociat un buffer. De exemplu, streamurile de iesire, cum ar fi std::cout, au un buffer care stocheaza datele inainte de a le trimite la consola.