Moştenirea Multiplă. Clase Virtuale

Runtime Type Information - RTTI

- Facilitate a limbajului ce se referă la capacitatea sistemului de a raporta cu privire la tipul dinamic al unui obiect:
  - Oferă informații despre acest tip *în momentul rulării* (spre deosebire de *momentul compilării*).
- Clasele trebuie să fie *polimorfice* pentru ca mecanismul să funcționeze (e.g. să conțină cel puțin o funcție virtuală).
  - `dynamic_cast<>` - operator = converteşte în mod sigur un pointer sau o referință către un tip de bază la un pointer sau o referință către un tip derivat.
  - `typeid` - operator = întoarce tipul actual al obiectului curent referit de către un pointer (sau o referință).
- Incluse în biblioteca `typeinfo:` `#include <typeinfo>`

RTTI - `typeid`

- `typeid` – *operator* utilizat pentru a determina clasa unui *obiect* în timpul rulării.

```
const std::type_info& info = typeid(object_expression);
const std::type_info& info = typeid(type);
```

- Întoarce o referință la un obiect de tip `std::type_info`, cu durata de viață până la sfârșitul programului, şi care descrie "*obiectul*".
- Dacă "*obiectul*" este un pointer `null` dereferențiat, atunci operația va arunca o excepție de tip `std::bad_typeid`.
- Clasa `std::bad_typeid` este derivată din `std::exception`.
- Clasa `std::type_info` conține informații despre tip.
- Metoda `name()` returnează un şir de caractere cu numele tipului obiectului.

```
const char* std::type_info::name() const;
```

RTTI - typeid

```
class Animal {
  public:
    virtual ~Animal() { }
};

class Mamifer : public Animal {
  public:
    virtual ~Mamifer() { }
};

int main() {
  Animal a;
  Mamifer m;
  Animal *pa = &m; // upcasting

  cout << typeid(a).name() << endl;  // Animal (determinat static la compilare)
  cout << typeid(m).name() << endl;  // Mamifer (determinat static la compilare)
  cout << typeid(pa).name() << endl; // Animal* (determinat static la compilare)

  cout << typeid(*pa).name() << endl; // Mamifer (determinat dinamic la executie
                                      // deoarece este un pointer la o clasa
                                      // polimorfica)
}
```

RTTI - `dynamic_cast`

- `dynamic_cast` - programul converteşte un pointer către clasa de bază la un pointer către o clasă derivată şi permite apelarea funcțiilor membre (declarate sau moştenite) din clasa derivată.

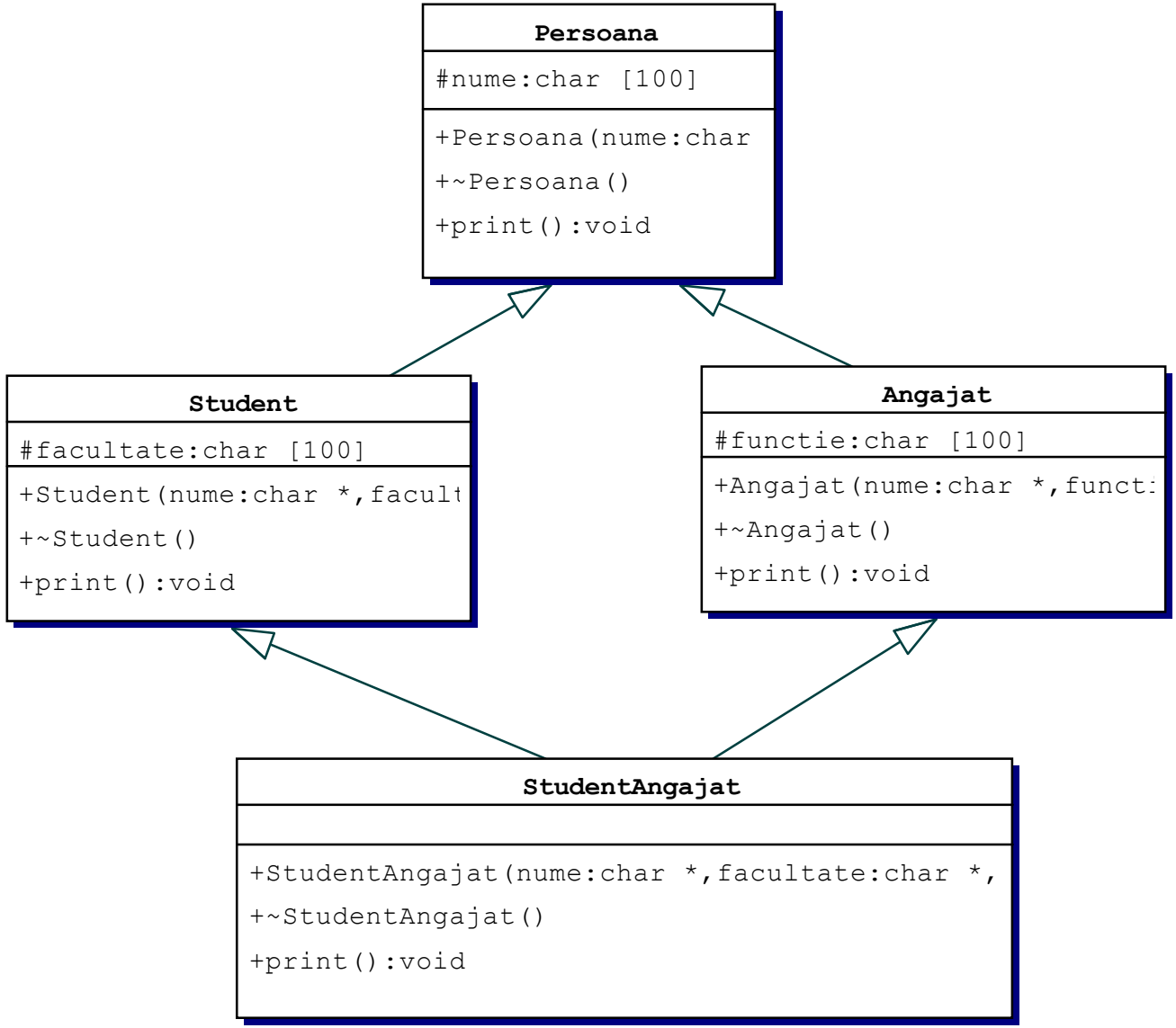
```
TYPE& dynamic_cast<TYPE&> (object);
TYPE* dynamic_cast<TYPE*> (object);
```

- Dacă încercați să convertiți la un tip pointer, iar acest tip nu este un tip real al obiectului argument, atunci rezultatul operației de conversie va fi `NULL`.
- Dacă încercați să convertiți la un tip referință, iar acest tip nu este un tip real al obiectului argument, atunci se va arunca o excepție `std::bad_cast`.
- `typeid` este preferat adesea față de `dynamic_cast` în situații în care este nevoie doar de informații despre clasă.
  - `typeid`, aplicat unui tip sau unei valori referință, are o durata de execuție constantă în timp, în timp ce `dynamic_cast` trebuie să traverseze în timpul rulării toată ierarhia de clase corespunzătoare argumentului său.

RTTI – dynamic_cast

```
class Animal {
  public:
    virtual ~Animal() { }
};

class Mamifer : public Animal {
  public:
    virtual ~Mamifer() { }
};


int main() {
  Animal a;
  Mamifer m;
  Animal *pa = &m;

  if (dynamic_cast<Mamifer*>(pa) != 0) { // downcasting
    Mamifer *pm = dynamic_cast<Mamifer*>(pa);
  }

  return 0;
}
```

Derivarea multiplă. Problema Diamantului (I)



Derivarea multiplă. Problema Diamantului (II)

```
class Persoana {
  protected:
    char nume[100];
  public:
    Persoana(const char* nume) {
      strcpy(this->nume, nume);
      cout << "Apel constructor Persoana\n";
    }

    ~Persoana() {
      cout << "Apel destructor Persoana\n";
    }

    void print() {
      cout << "Nume:" << nume << endl;
    }
};
```

```
class Student: public Persoana {
  protected:
    char facultate[100];
  public:
  Student(const char* nume, const char* facultate):
          Persoana(nume) {
      strcpy(this->facultate, facultate);
      cout << "Apel constructor Student\n";
    }

    ~Student() {
      cout << "Apel destructor Student\n";
    }

    void print() {
      Persoana::print();
      cout << "Facultate:" << facultate << endl;
    }
};
```

Derivarea multiplă. Problema Diamantului (II)

```
class Angajat: public Persoana {
  protected:
    char functie[100];
  public:
    Angajat(const char* nume,
            const char* functie):Persoana(nume) {
      strcpy(this->functie, functie);
      cout << "Apel constructor Angajat\n";
    }

    ~Angajat() {
      cout << "Apel destructor Angajat\n";
    }

    void print() {
      Persoana::print();
      cout << "Functie:" << functie << endl;
    }
};
```

```
class StudentAngajat: public Student,
                      public Angajat {
 public:
   StudentAngajat(const char* nume,
                  const char* facultate,
                  const char* functie)
                     : Student(nume, facultate),
                       Angajat(nume, functie) {
     cout<<"Apel constructor StudentAngajat\n";
   }

   ~StudentAngajat() {
     cout << "Apel destructor StudentAngajat\n";
   }

   void print();
};
```

Derivarea multiplă. Problema Diamantului (III)

```
StudentAngajat:: print() {
   cout << "Nume: " << nume << endl;
   Student::print();
   cout << "Functie: " << functie << endl;

}


int main(){
 StudentAngajat sa("Teddy", "Informatica",
"programator");
 sa.print();
 sa.Student::print();
 sa.Angajat::print();

 return 0;
}
```

```
Apel constructor Persoana
Apel constructor Student
Apel constructor Persoana
Apel constructor Angajat
Apel constructor StudentAngajat
Nume:Teddy
Facultate:Informatica
Functie:programator
Nume:Teddy
Facultate:Informatica
Nume:Teddy
Functie:programator
Apel destructor StudentAngajat
Apel destructor Angajat
Apel destructor Persoana
Apel destructor Student
Apel destructor Persoana
```

Probleme

- Duplicarea datelor membre din clasa `Persoana` la nivelul clasei `StudentAngajat`.
- La nivelul clasei `StudentAngajat` nu putem să referim direct data membru **`nume`** moştenită de la clasele de bază.
- Putem însă folosi operatorul de rezoluție pentru a referi această dată membru astfel:

`Student::nume` respectiv `Angajat::nume`

Clase virtuale

- Pentru a evita duplicarea datelor membre în cazul moştenirilor multiple, clasa de bază se declară **virtuală** în declarațiile claselor derivate:

```
class B { protected tip x; };
class B1: virtual public B { };
class B2: virtual public B { };
class D: public B1, public B2 { };
```

- Rezultat: data membru **x** se include o singură dată la nivelul clasei derivate **D**.

Clase virtuale. Constructori

- Constructorul clasei derivate **D** trebuie să apeleze explicit constructorul clasei **B**, în vederea creării copiei unice a datelor moştenite de la clasa **B**, pe lângă apelul constructorilor claselor de bază:

```
class D: public B1, public B2 {
     D(…): B1(…), B2(…), B(…) {
        …
     }
};
```

- Ordinea de apel a constructorilor este **B**, **B1**, **B2**, **D**.

Exemplu

```
class Student: virtual public Persoana {
  ...
};

class Angajat: virtual public Persoana {
  ...
};

class StudentAngajat: public Student, public Angajat {
  public:
    StudentAngajat(const char* nume, const const char* facultate, const char* functie):
                  Angajat(nume, functie), Student(nume, facultate), Persoana(nume) {
      cout << "Apel constructor StudentAngajat\n";
    }

    ~StudentAngajat(){
      cout << "Apel destructor StudentAngajat\n";
    }

    void print() {
      cout << "Nume: " << nume << endl;
      cout << "Facultate:" << facultate << endl;
      cout << "Functie:" << functie << endl;
    }
};
```

Exemplu (cont.)

```
int main() {
 StudentAngajat sa("Teddy", "Informatica",
                   "programator");
 sa.print();
 sa.Student::print();
 sa.Angajat::print();

 return 0;
}
```

```
Apel constructor Persoana
Apel constructor Student
Apel constructor Angajat
Apel constructor StudentAngajat
Nume:Teddy
Facultate:Informatica
Functie:programator
Nume:Teddy
Facultate:Informatica
Nume:Teddy
Functie:programator
Apel destructor StudentAngajat
Apel destructor Angajat
Apel destructor Student
Apel destructor Persoana
```

Operații de intrare/ieşire în C++

Operații I/O în limbajul C

- Biblioteci standard `stdio.h` şi `conio.h`
- Fişiere text
  - `printf / fprintf`
  - `scanf / fscanf`
- Fişiere binare
  - `fread`
  - `fwrite`

Operații I/O în limbajul C++

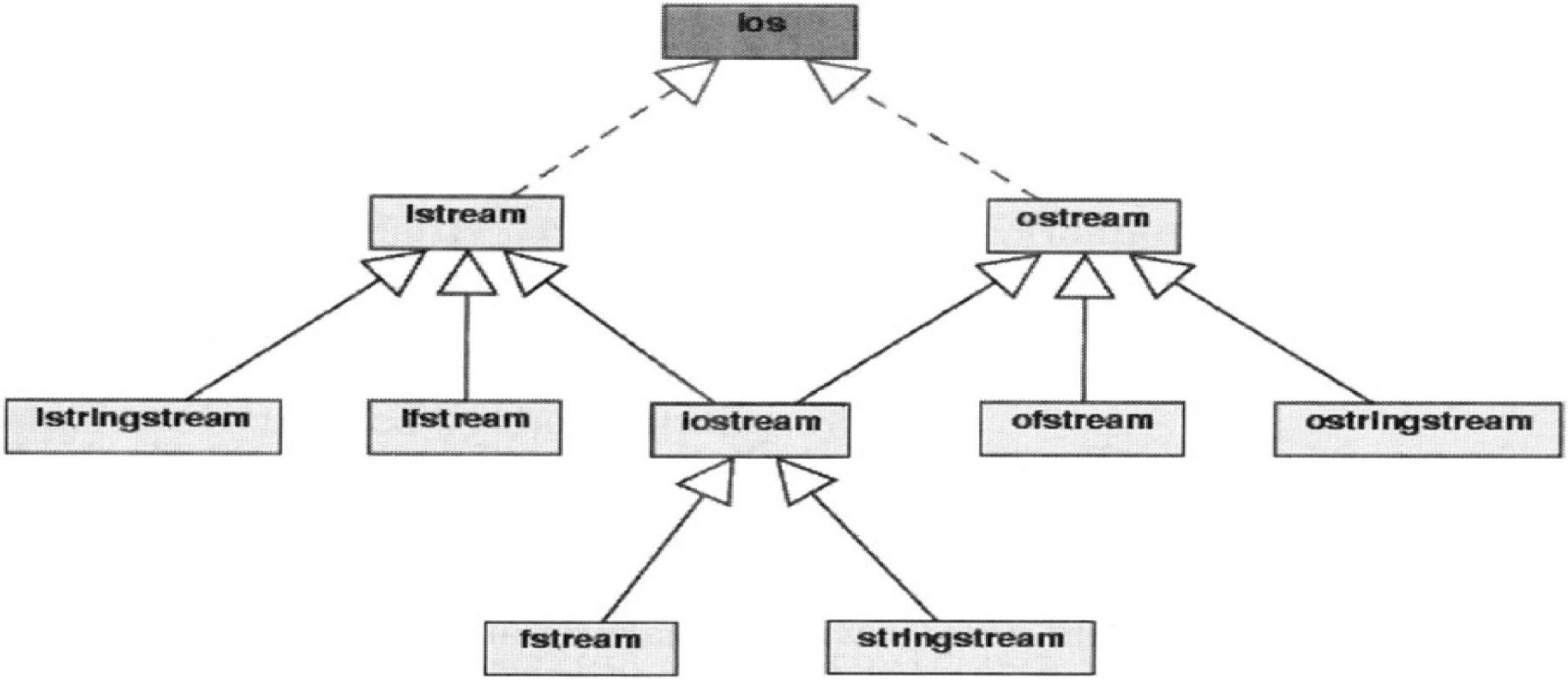
- Limbajul C++ moşteneşte de la limbajul C funcțiile de I/O.
  - Dezavantaj: permit manipularea doar a tipurilor de bază.
- Limbajul C++ introduce o ierarhie de clase pentru a realiza operațiile de intrare/ieşire (I/O).
- Această ierarhie are la bază concepul de **stream** (flux).
- Flux = secvență de octeți
  - *flux de intrare* – flux de octeți de la o sursă de date (tastatură, unitate de disc, conexiune rețea etc.) către program;
  - *flux de ieşire* – flux de octeți de la program către un consumator de date (ecran, imprimantă, unitate de disc etc.).

Ce este un stream?

- **Stream**-ul este un concept abstract ce înglobează orice ***flux*** de date de la o sursă (canal de intrare) la o destinație (canal de ieşire). Clasificare:
  - *flux de date fără formatare* (de nivel jos): flux binar, octetul fiind unitatea atomică, recomandat pentru prelucrarea rapidă a unor cantități mari de date.
  - *flux de date cu formatare* (flux de nivel înalt): flux text, caracterul fiind unitatea atomică, recomandat pentru majoritatea operațiilor de I/O, fără cerințe de performanță la prelucrare.
- Biblioteca standard ce înglobează ierarhia de clase pentru operații de I/O:

```
#include <iostream.h>  // compilatoare vechi
sau
#include <iostream>
using namespace std;
```

Ierarhia de clase I/O



Stream-uri standard

- Stream-uri standard:
  - *cin* – obiect de tip *istream* (*stream*-ul standard de intrare);
  - *cout* – obiect de tip o*stream* (*stream*-ul standard de ieşire);
  - *cerr* – un obiect de tip o*stream* (*stream*-ul standard de eroare).
- Operatori
  - inserare: ‘<<‘
  - extragere: ‘>>’

Stream-uri standard - `cin`

- operatorul "`<<`" (*operator de inserție în stream*) se folosește pentru operațiile de scriere pe un stream (ieșire standard, fișier etc.).
- este un operator binar.
  - în partea sa stângă (operandul stâng) trebuie să avem un obiect de tip `ostream` (sau derivat din `ostream`).
  - în partea dreaptă (operandul drept) putem avea o expresie.
- operatorul este supraîncărcat pentru tipurile standard.
- pentru tipurile noi programatorul trebuie să supraîncarce.
- se pot înlănțui operațiile de inserție.
  - operatorul "`<<`" returnează o referință la stream
- pentru a scrie pe ieșirea standard (consolă) se folosește `cout`.

Stream-uri standard - `cout`

- operatorul "`>>`" (*operator de extracție din stream*) se folosește pentru operațiile de citire dintr-un stream (intrare standard, fișier etc.).
- este un operator binar.
  - în partea sa stângă (operandul stâng) trebuie să avem un obiect de tip `istream` (sau derivat din `istream`).
  - în partea dreaptă (operandul drept) putem avea o variabilă.
- operatorul este supraîncărcat pentru tipurile standard.
- pentru tipurile noi programatorul trebuie să supraîncarce.
- operațiile de extracție se pot înlănțui.
  - operatorul "`>>`" returnează o referință la stream
- pentru a prelua date de la intrarea standard (tastatură) se folosește `cin`.

Formatarea ieşirii

- Funcții pentru formatarea ieşirii:
  - `fmtflags setf(fmtflags fmtfl);` sau
  - `fmtflags setf(fmtflags fmtfl, fmtflags mask);` - realizează activarea unor indicatori de formatare.

Masca de biți (`mask`)	Valorile flagurilor (`fmtfl`)
`adjustfield`	`left`, `right` sau `internal`
`basefield`	`dec`, `oct` sau `hex`
`floatfield`	`scientific` sau `fixed`

Exemplu: `cout.setf(ios::hex, ios::basefield)`

- `setf(fmtfl)` este echivalentă cu `flags(fmtfl|flags())`
- `setf(fmtfl, mask)` este echivalentă cu `flags((fmtfl&mask)|(flags()&~mask))`

Formatarea ieşirii

- `void unsetf (fmtflags mask);` - dezactivarea unor indicatori

  `cout.unsetf(ios::fixed | ios::scientific);`
- `int width(int w);` – specifică numărul minim de caractere alocat câmpului în care se va efectua următoarea scriere.
  - dacă numărul de caractere necesar pentru scrierea unei valori este mai mic decât numărul specificat atunci spatiul rămas liber va fi completat cu un caracter specificat. Acesta este setat în mod implicit la un spatiu.
  - dacă scrierea unei valori necesită un număr de caractere mai mare decât cel specificat atunci se vor folosi atâtea caractere câte sunt necesare.
- În cazul în care valoarea afișată ocupă mai puține caractere decât lungimea, ea poate fi *aliniată la dreapta* `ios::right` (implicit) sau *la stânga* `ios::left` (pentru orice fel de date) sau *internal*, pentru date numerice (întregi sau reale).
- Alinierea internă separă semnul unui număr de cifrele sale. Semnul este aliniat la stânga, iar cifrele sunt aliniate la dreapta. Este folosit în programele de contabilitate.

Formatarea ieşirii

- `int precision(int p);` – stabileşte numărul de cifre folosite pentru afișarea valorilor reale.
  - în funcție de context, poate reprezenta numărul total de cifre sau numărul de cifre de după punctul zecimal.
- numerele reale sunt păstrate folosind formatul cu virgulă mobilă.
- avem trei formate de afişare: general (implicit), fix şi ştiințific.

Formatarea ieşirii

- Formatul *fix* (`ios::fixed`), are o parte întreagă, punct zecimal şi o parte fracționară. Precizia reprezintă numărul de cifre aflate după punctul zecimal.
- În formatul *ştiințific* (`ios::scientific`), un număr se reprezintă sub forma $F = \pm m \times 10^n$ sau $F = \pm mEn$, unde m este mantisa ($1 \leq m < 10$) iar n este exponentul (poate fi orice număr întreg pozitiv sau negativ). Precizia reprezintă numărul de zecimale ale mantisei.
- Formatul *implicit* afișează valoarea în forma fixă sau în forma științifică, în funcție de precizie. Fie n numărul de cifre ale părții întregi a valorii de afișat și fie p precizia curentă:
  - dacă $n \leq p$, se va folosi formatul fix de afișare, iar pentru partea zecimală se vor afișa $p-n$ zecimale, cu rotunjire.
  - dacă $n > p$, se va folosi formatul științific de afișare, cu mantisă și exponent, iar precizia reprezintă numărul de cifre ale mantisei.

Formatarea ieşirii

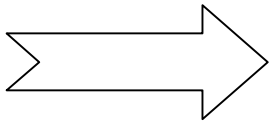
- Numerele în baza 10 (`ios::dec`) nu au prefix, numerele în baza 8 (`ios::oct`) sunt prefixate cu cifra 0, iar numerele în baza 16 (`ios::hex`) sunt prefixate cu `0x`.
- `char fill(char ch);` – stabileşte caracterul folosit pentru umplerea spațiilor rămase libere din cadrul câmpului de afişare.
- `ios::adjustfield` – stabileşte modul de aliniere.
- `ios::basefield` – stabileşte baza de numerație.
- `ios::floatfield` – stabileşte modul de reprezentare a numerelor reale la afişare.

Formatarea ieşirii

- Indicatori de formatare:
  - `ios::fixed` – numerele reale vor fi afişate folosind notația cu virgulă zecimală.
  - `ios::scientific` – numerele reale vor fi afişate folosind notația ştiințifică (cu exponent).
  - `ios::showpoint` – virgula zecimalelor va fi afişată întotdeauna la variabile de tip număr real chiar şi pentru valori întregi.
  - `ios::right` – specifică ca datele sa fie afişate aliniate la dreapta, atunci când este specificată lungimea câmpului de afişare

Exemplul 1

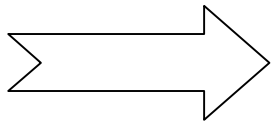
```
#include <iostream>

using namespace std;

int main(){
  cout << 128 << endl ;  // afişarea se face pe dimensiune implicita

  cout.unsetf(ios::dec); // se dezactiveaza afisarea in zecimal
  cout.setf(ios::hex);   // se activeaza afisare in hexazecimal
  cout.setf (ios::showbase); // se afiseaza baza de numeratie
  cout << 255 << endl;

  cout.setf(ios::scientific);
  cout << 123.456;

  return 0 ;
}
```



```
128
0xff
1.234560e+002
```

Exemplul 2

```
#include <iostream>

using namespace std;

int main(){
  cout << 123 << endl; // afisarea se face pe dimensiunea implicita

  cout.width(10);      // se modifica campul la dimensiunea 10
  cout << 123 << endl;
  cout.width(12);      // se modifica campul la dimensiunea 12
  cout.fill('*');      // se stabileste caracterul de umplere la '*'
  cout << 123 << endl;
  cout.width(12);      // se modifica campul la dimensiunea 12
  cout.precision(2);   // se specifica afisarea cu precizie 2
  cout.setf(ios::fixed); // este activata notatia cu virgula zecimala
  cout << 123.456 << endl;

  return 0;
}
```



```
123
       123
*********123
******123.46
```

Manipulatori

- Formatarea se mai poate realiza şi cu ajutorul **manipulatorilor**.
- Un **manipulator** este o funcție auxiliară având rolul de a modifica parametrii de formatare ai unui flux de intrare / ieşire.
- Un **manipulator** primeşte ca parametru şi întoarce ca rezultat, adresa fluxului asupra căruia trebuie să acționeze, astfel încât pot fi inserați în expresiile de intrare / ieşire.
- Exista manipulatori cu parametri şi fără parametri.
- Biblioteca standard ce include manipulatori cu parametri:

```
#include <iomanio.h>  // compilatoare vechi
```

sau

```
#include <iomanio>  // compilatoare noi
```

Manipulatori

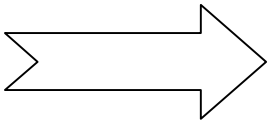
- Exemple de manipulatori definiți:
  - `dec` - activează bitul *dec* (la intrare / ieşire);
  - `hex` - activează bitul *hex* (la intrare / ieşire);
  - `oct` - activează bitul *oct* (la intrare / ieşire);
  - `endl` - inserează caracterul '\n' şi goleşte zona buffer (la ieşire);
    - Obs. Mai lent decât `cout << '\n';`
  - `flush` - goleşte zona buffer (la ieşire);
  - `setfill(int)` - stabileşte caracterul de umplere;
  - `setprecision(int)` - stabileşte precizia pentru numere în virgulă mobilă;
  - `setw(int)` - stabileşte dimensiunea câmpului folosit pentru următoarea operație de afişare.

Manipulatori

- Exemple de manipulatori definiți:
  - `setiosflags()` - un manipulator parametrizat îndeplinind aceeași sarcină ca funcția membru `setf()`;
  - `left` - activează alinirea la stânga (la ieşire);
  - `right` - activează alinierea la dreapta (la ieşire);
  - `internal` - activează alinierea internă (la ieşire);
  - `fixed` - activează afişarea numerelor reale în format fix;
  - `scientific` - activează afişarea numerelor reale în format ştiințific;
  - `showbase` - scrie şi baza de numerație la afişarea unui număr întreg;
  - `showpos` – afişează '+' în fața valorilor pozitive.
  - `skipws` – la citire se sare peste eventualele caracterele albe aflate înainte de valoarea de citit.

Exemplul 3

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
  cout << 123 << endl;

  cout << hex << showbase << 123 << endl;
  cout << oct << showbase << 123 << endl;
  cout << 1234.56 << endl;

  cout << setprecision(2) << setw(10) << setfill('*')
       << setiosflags(ios::fixed);
  cout << 1234.56 << endl;

  return 0;
}
```



```
123
0x7b
0173
1234.56
***1234.56
```

Formatarea intrării / ieşirii

- Pentru formatarea afişării se pot specifica
  - *lungimea* – numărul de caractere care vor fi utilizate pentru afișarea valorii dorite;
  - *alinierea*;
  - *caracterul de umplere*;
  - *precizia*;
  - *baza de numerație*;
  - *formatul de afișare al valorilor reale*.
- Pentru formatarea citirii se pot specifica
  - *baza de numerație* în care se consideră valoarea întreagă introdusă;
  - *modul de tratare* a caracterelor albe.

Operații de intrare / ieşire cu fişiere

- Operațiile de intrare / ieşire cu fişiere se realizează prin intermediul următoarelor clase:
  - `ifstream`
  - `ofstream`
  - `fstream`.

Operații de intrare / ieşire cu fişiere

- Constructori

```
ofstream::ofstream(const char* name,
                   ios::openmode = ios::out | ios::trunc);
ifstream::ifstream(const char* name,
                    ios::openmode = ios::in);
fstream::fstream(const char* name,
                  ios::openmode = ios::in | ios::out);
```

Operații de intrare / ieşire cu fişiere

- Moduri de deschidere a fişierelor:
  - *ios::in (input)* – permite operații de citire din flux;
  - *ios::out (output)* – permite operații de scriere în flux;
  - *ios::app (append)* – adaugă la sfârşitul fişierului;
  - *ios::ate (at end)* – poziționează pointer-ul la sfârşitul fişierului, însă informațiile pot fi scrise oriunde în cadrul fişierului;
  - *ios::trunc (truncate)* – este modul de deschidere implicit: vechiul conținut al fişierului este pierdut;
  - *ios::binary* – deschide fişierul în mod binar.
  - *ios::nocreate* (*deprecated*) – dacă fişierul nu există, atunci operația eşuează;
  - *ios::noreplace* (*deprecated*) – dacă fişierul deja există, atunci operația eşuează.

Operații de intrare / ieşire cu fişiere

- Rezultatul operațiilor de intrare/iesire poate fi testat prin intermediul a patru funcții membre:
  - `eof()` – întoarce *true* dacă s-a ajuns la sfârşitul fişierului;
  - `bad()` – întoarce *true* dacă o operație de citire sau scriere eşuează;
  - `fail()` – întoarce *true* dacă o operație de citire sau scriere eşuează sau dacă apare o eroare de format la citire;
  - `good()` – verifică dacă toate cele trei rezultate precedente sunt *false*.
- Închiderea unui fişier se face apelând funcția (metoda)
  - `close()`

Exemplul 4

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
    int n = 10, t;
    ofstream out("output.txt");

    for(int i = 0; i < n; i++) {
      out << i * i << endl;
    }
    out.close();
    ifstream in("output.txt");
    if (!in) {
      cerr << "Eroare deschidere";
      exit(0);
    }

    while (!in.eof()) {
     in >> t;
     if (in.good()) {
        cout << t << endl;
     }
    }

    in.close();

    return 0;
}
```

Operații de citire / scriere în mod binar

- Scrierea într-un fişier binar se poate face prin intermediul funcției:

```
ostream& write(const char* s, streamsize n);
```

- Citirea dintr-un fişier binar se poate face prin intermediul funcției:

```
istream& read(const char* s, streamsize n);
```

Exemplul 5

```
#include <iostream>
#include <fstream>

using namespace std;


int main() {
  int n = 10, t;
  ofstream out("output.bin", ios::binary);

  for(int i = 0; i < n; i++) {
  //  out.write(static_cast<char*>(&i),
  //            sizeof(i));
    out.write(reinterpret_cast<char*>(&i),
              sizeof(i));
  }

  out.close();
```

```
  ifstream in("output.bin", ios::binary);

  if (!in) {
    cerr << "Eroare deschidere fisier";
    exit(0);
  }

  while(!in.eof()) {
    // in.read(static_cast<char*>(&t),
    //         sizeof(t));
     in.read(reinterpret_cast<char*>(&t),
             sizeof(t));
    if (in.good()) {
      cout << t << endl;
    }
  }

  in.close();

  return 0;
}
```

Poziționarea într-un fişier

- Există doi pointeri ce indică locația (în cadrul fluxului de intrare / ieşire) unde se va efectua următoarea operație – scriere (*put*) sau citire (*get*).
- Obținerea valorii pointerului ce indică locația (poziția) unde se va efectua următoare operație (citire / scriere) într-un fişier, se face prin intermediul următoarelor metode:
  - `pos_type ostream::tellp();`
  - `pos_type istream::tellg();`
- `tellp()` – poziția pointerului pentru scriere.
- `tellg()` – poziția pointerului pentru citire.

Poziționarea într-un fişier

- Poziționarea pointerului (cursorului) ce indică locația unde se va realiza operația următoare (citire / scriere) într-un fişier, se poate face prin intermediul metodelor:
  - `ostream& ostream::seekp(off_type offset,`
    `ios::seekdir pos);`
  - `istream& istream::seekg(off_type offset,`
    `ios::seekdir pos);`

unde `pos` poate avea una din următoarele valori:

- `ios::beg` – specifică poziția de la începutul stream-ului;
- `ios::end` – specifică poziția de la sfârşitul stream-ului;
- `ios::cur` – specifică poziția curentă a stream-ului.

Referințe

- https://www.bogotobogo.com/cplusplus/dynamic_cast.php
- https://stackoverflow.com/questions/332030/when-should-static-cast-dynamic-cast-const-cast-and-reinterpret-cast-be-used
- http://faculty.cs.niu.edu/~mcmahon/CS241/c241man/node83.html
- http://www.cs.fsu.edu/~cop3014p/lectures/ch3/index.html
- https://www.tenouk.com/Module18a.html