Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Anca Dobrovăț
Andrei Păun

An universitar 2024 – 2025
Semestrul I
Seria 26

Curs 7

Agenda cursului

1. Polimorfims la execuție prin funcții virtuale în C++ (recapitulare și completări la cursul 6)

Parametrizarea metodelor (polimorfism la execuție).
Funcții virtuale în C++.
Clase abstracte.
Overloading pe funcții virtuale
Destructori și virtualizare

2. Tratarea exceptiilor

Discutie SMART POINTERS

1. Moştenirea in C++

C++ permite moștenirea ceea ce înseamnă că putem deriva o clasă din altă clasă de bază sau din mai multe clase.

Sintaxa:

class Clasa_Derivată : [modificatori de acces] Clasa_de_Bază { .... } ;

sau

class Clasa_Derivată : [modificatori de acces] Clasa_de_Bază1, [modificatori de acces] Clasa_de_Bază2, [modificatori de acces] Clasa_de_Bază3 .....

Clasă de bază se mai numește **clasa părinte** sau **superclasă**, iar clasa derivată se mai numește **subclasa** sau **clasa copil**.

1. Moştenirea in C++

Inițializare de obiecte

Foarte important în C++: garantarea inițializării corecte => trebuie să fie asigurată și la compoziție și moștenire.
La crearea unui obiect, compilatorul trebuie să garanteze apelul TUTUROR subobiectelor.

Problema: - cazul subobiectelor care nu au constructori impliciți sau schimbarea valorii unui argument default în constructor.

De ce? - constructorul noii clase nu are permisiunea să acceseze datele private ale subobiectelor, deci nu le pot inițializa direct.

Rezolvare: - o sintaxă specială: ***listă de inițializare pentru constructori***.

1. Moştenirea in C++

Lista de iniţializare pentru constructori (utilitate in cazul Mostenirii)

```
class Alta_clasa { int a;
      public:
      Alta_clasa(int i) {a = i;}
 };
class Bar { int x;
      public:
         Bar(int i) {x = i;}
 };
class MyType2: public Bar {
      Alta_clasa m; // obiect m = subobiect in cadrul clasei MyType2
      public:
         MyType2(int);
};
```

```
MyType2 :: MyType2 (int i) : Bar (i), m(i+1) { … }
```

1. Moştenirea in C++

Constructorii clasei derivate

Pentru crearea unui obiect al unei clase derivate, **se creează iniţial un obiect al clasei de bază** prin apelul constructorului acesteia, apoi se adaugă elementele specifice clasei derivate prin apelul constructorului clasei derivate.

Declaraţia obiectului derivat trebuie să conţină valorile de iniţializare, **atât pentru elementele specifice, cât şi pentru obiectul clasei de bază**.

Această specificare se ataşează la antetul funcţiei constructor a clasei derivate.

În situaţia în care clasele de bază au definit **constructor implicit** sau **constructor cu parametri impliciţi**, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază.

1. Moştenirea in C++

Constructorii clasei derivate

Constructorul de copiere

Se pot distinge mai multe situaţii.

1) Dacă ambele clase, atât clasa derivată cât şi clasa de bază, nu au definit constructor de copiere, se apelează constructorul implicit creat de compilator. Copierea se face membru cu membru.

2) Dacă clasa de bază are constructorul de copiere definit, dar clasa derivată nu, pentru clasa derivată compilatorul creează un constructor implicit care apelează constructorul de copiere al clasei de bază. (poate fi considerata un caz particular al primei situații, deoarece și partea de bază poate fi privită ca un fel de membru, iar la copiere se apelează cc pentru fiecare membru).

3) Dacă se defineşte constructor de copiere pentru clasa derivată, acestuia îi revine în totalitate sarcina transferării valorilor corespunzătoare membrilor ce aparţin clasei de bază.

1. Moştenirea in C++

Ordinea chemării constructorilor și destructorilor

Constructorii sunt chemați în ordinea definirii obiectelor ca membri ai clasei și în ordinea moştenirii:

- la fiecare nivel se apelează întâi constructorul de la moştenire, apoi constructorii din obiectele membru în clasa respectivă (care sunt chemați în ordinea definirii) și la final constructorul propriu;

- se merge pe următorul nivel în ordinea moştenirii;

Destructorii sunt chemați în ordinea inversă a constructorilor

1. Moştenirea in C++

Redefinirea funcţiilor membre

Clasa derivată are acces la toţi membrii cu acces **protected** sau **public** ai clasei de bază.

Este permisă supradefinirea funcţiilor membre clasei de bază cu funcţii membre ale clasei derivate.

-2 modalităţi de a redefini o funcţie membră:

- **cu acelasi antet ca în clasa de bază** (“redefining” - în cazul funcţiilor oarecare / “overloading” - în cazul funcţiilor virtuale);
- **cu schimbarea listei de argumente sau a tipului returnat**.

1. Moştenirea in C++

Redefinirea funcţiilor membre

Obs:

Schimbarea interfeței clasei de bază prin modificarea tipului returnat sau a signaturii unei funcții, înseamnă, de fapt, utilizarea clasei în alt mod.

Scopul principal al moştenirii: polimorfismul.

Schimbarea signaturii sau a tipului returnat = schimbarea interfeței = contravine exact polimorfismului (un aspect esențial este păstrarea interfeței clasei de bază).

Moştenirea si funcţiile statice

Funcțiile membre statice se comportă exact ca și funcțiile membre nestatice: Se moștenesc în clasa derivată.

Redefinirea unei funcții membre statice duce la ascunderea celorlalte supraîncărcări.

Schimbarea signaturii unei funcții din clasa de bază duce la ascunderea celorlalte versiuni ale funcției.

Dar: O funcție membră statică nu poate fi virtuală.

1. Moştenirea in C++

Modificatorii de acces la moştenire

class A : ***public*** B { /* declaraţii */};

class A : ***protected*** B { /* declaraţii */};

class A : ***private*** B { /* declaraţii */};

Dacă modificatorul de acces la moştenire este **public**, membrii din clasa de bază își păstrează tipul de acces și în derivată.

Dacă modificatorul de acces la moştenire este **private**, toți membrii din clasa de bază vor avea tipul de acces “private” în derivată, indiferent de tipul avut în bază.

Dacă modificatorul de acces la moştenire este **protected**, membrii “publici” din clasa de bază devin “protected” în clasa derivată, restul nu se modifică.

1. Moştenirea in C++

Moştenirea cu specificatorul "private"

•inclusă în limbaj pentru completitudine;

•este mai bine a se utiliza compunerea în locul moştenirii private;

•toți membrii private din clasa de bază sunt ascunși în clasa derivată, deci inaccesibili;

•toți membrii public și protected devin private, dar sunt accesibile în clasa derivată;

•un obiect obținut printr-o astfel de derivare se tratează diferit față de cel din clasa de bază, e similar cu definirea unui obiect de tip bază în interiorul clasei noi (fără moştenire).

•dacă în clasa de bază o componentă era public, iar moştenirea se face cu specificatorul private, se poate reveni la public utilizând:

using Baza::nume_componenta

1. Moştenirea in C++

Moştenire multiplă (MM)

- putine limbaje au MM;

- moştenirea multiplă e complicată: ambiguitate LA MOSTENIREA IN ROMB / IN DIAMANT;

- nu e nevoie de MM (se simulează cu moştenire simplă);

- se moşteneste in acelaşi timp din mai multe clase;

Sintaxa:
class Clasa_Derivată : [modificatori de acces] Clasa_de_Bază1, [modificatori de acces] Clasa_de_Bază2, [modificatori de acces] Clasa_de_Bază3 .....

1. Moştenirea in C++

Moştenire multiplă (MM)

- dar dacă avem nevoie doar de o copie lui i?

- nu vrem să consumăm spaţiu în memorie;

- ***folosim moştenire virtuală***:

```
class base {  public:     int i; };
class derived1 : virtual public base {  public:     int j; };
class derived2 : virtual public base {  public:     int k; };
class derived3 : public derived1, public derived2 {public:    int sum; };
```

•Dacă avem moştenire de două sau mai multe ori dintr-o clasă de bază (fiecare moştenire trebuie să fie virtuală) atunci compilatorul alocă spaţiu pentru o singură copie;
•În clasele derived1 şi 2 moştenirea e la fel ca mai înainte (niciun efect pentru virtual în acel caz)

2. Polimorfismul la execuţie prin funcţii virtuale

Funcţii virtuale

Funcţiile virtuale şi felul lor de folosire: componentă IMPORTANTĂ a limbajului OOP.

Folosit pentru polimorfism la execuţie ---> cod mai bine organizat cu polimorfism.

Codul poate "creşte" fără schimbări semnificative: programe extensibile.

Funcţiile virtuale sunt definite în bază şi redefinite în clasa derivată.

Pointer de tip bază care arată către obiect de tip derivatşi cheamă o funcţie virtuala în bază şi redefinite în clasa derivată executa ***Funcţia din clasa derivată***.

Poate fi vazuta ca exemplu de separare dintre interfata si implementare.

2. Polimorfismul la execuţie prin funcţii virtuale

Decuplare in privinţa tipurilor

Upcasting - Tipul derivat poate lua locul tipului de bază (foarte important pentru procesarea mai multor tipuri prin acelaşi cod).

Funcţii virtuale: ne lasă să chemăm funcţiile pentru tipul derivat.

Problemă: apel la funcţie prin pointer (tipul pointerului ne da funcţia apelată).

2. Polimorfismul la execuţie prin funcţii virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {  public:
  void play(note) const {
    cout << "Instrument::play" << endl;  }
};

class Wind : public Instrument {
public:  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;  }
};

void tune(Instrument& i) {  i.play(middleC);  }

int main() {
  Wind flute;
  tune(flute); // Upcasting ===> se afiseaza Instrument::play
}
```

2. Polimorfismul la execuţie prin funcţii virtuale

In C ---> early binding la apel de funcţii - se face la compilare.

In C++ ---> putem defini late binding prin funcţii virtuale (late, dynamic, runtime binding) - se face apel de funcţie bazat pe tipul obiectului, la rulare (nu se poate face la compilare).

Late binding ===> prin pointeri!

Late binding pentru o funcţie: se scrie virtual inainte de definirea funcţiei.

Pentru clasa de bază: nu se schimbă nimic!

Pentru clasa derivată: late binding însemnă că un obiect derivat folosit în locul obiectului de bază îşi va folosi funcţia sa, nu cea din bază (din cauză de late binding).

Utilitate: putem extinde codul precedent fara schimbari in codul deja scris.

2. Polimorfismul la execuţie prin funcţii virtuale

Cum se face late binding

Tipul obiectului este ţinut în obiect pentru clasele cu funcţii virtuale.

Late binding se face (uzual) cu o tabelă de pointeri: vptr către funcţii.

În tabelă sunt adresele funcţiilor clasei respective (funcţiile virtuale sunt din clasa, celelalte pot fi moştenite, etc.).

Fiecare obiect din clasă are pointerul acesta în componenţă.

La apel de funcţie membru se merge la obiect, se apelează funcţia prin vptr.

Vptr este iniţializat în constructor (automat).

2. Polimorfismul la execuţie prin funcţii virtuale

Cum se face late binding

```
class Pet { public:
  virtual string speak() const { return " "; } };

class Dog : public Pet { public:
  string speak() const { return "Bark!"; } };

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding for both:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probably):
  cout << "p3.speak() = " << p3.speak() << endl;
}
```

2. Polimorfismul la execuţie prin funcţii virtuale

Clase abstracte și funcții virtuale pure

Clasă abstractă = clasă care are cel puțin o funcție virtuală PURĂ

Necesitate: clase care dau doar interfață (nu vrem obiecte din clasă abstractă ci upcasting la ea).

Eroare la instantierea unei clase abstracte (nu se pot defini obiecte de tipul respectiv).

Permisă utilizarea de pointeri și referințe către clasă abstractă (pentru upcasting).

Nu pot fi trimise către funcții (prin valoare).

2. Polimorfismul la execuţie prin funcţii virtuale

Funcţii virtuale pure

Sintaxa: **virtual** ***tip_returnat nume_funcţie(lista_parametri) =0;***

Ex: virtual int pura(int i)=0;

Obs: La moștenire, dacă în clasa derivată nu se definește funcția pură, clasa derivată este și ea clasă abstractă ---> nu trebuie definită funcție care nu se execută niciodată

UTILIZARE IMPORTANTĂ: prevenirea “object slicing”.

2. Polimorfismul la execuţie prin funcţii virtuale

Overload pe funcţii virtuale

Obs. Nu e posibil overload prin schimbarea tipului param. de întoarcere (e posibil pentru ne-virtuale)

De ce. Pentru că se vrea să se garanteze că se poate chema baza prin apelul respectiv.

Excepție: pointer către bază întors în bază, pointer către derivată în derivată

2. Polimorfismul la execuţie prin funcţii virtuale

Overload pe funcţii virtuale

```
class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";  return 1;   }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {public:
  void g() const {}
};

class Derived2 : public Base {public:
  // Overriding a virtual funcţion:
  int f() const {  cout << "Derived2::f()\n";
    return 2;  }
};
```

```
int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
}
```

2. Polimorfismul la execuţie prin funcţii virtuale

Overload pe funcţii virtuale

```
class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";  return 1;   }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived3 : public Base {public:
//! void f() const{ cout << "Derived3::f()\n";}};

class Derived4 : public Base {public:
  // Change argument list:
  int f(int) const
       { cout << “Derived4::f()\n"; return 4; }
};
```

```
int main() {
  string s("hello");
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // f() version hidden
//!  d4.f(s); // string version hidden
  Base& br = d4; // Upcast
//!  br.f(1); // Derived version
unavailable
  br.f();
  br.f(s); // Base version available
return 0;
}
```

2. Polimorfismul la execuţie prin funcţii virtuale

Constructori si virtualizare

Obs. NU putem avea constructori virtuali.

În general pentru funcțiile virtuale se utilizează late binding, dar în utilizarea funcțiilor virtuale în constructori, varianta locală este folosită (early binding)

De ce?

Pentru că funcția virtuală din clasa derivată ar putea crede că obiectul e inițializat deja

Pentru că la nivel de compilator în acel moment doar VPTR local este cunoscut

Cursul urmator detalii despre clone ("virtualizarea" constructorilor)

2. Polimorfismul la execuţie prin funcţii virtuale

Destructori si virtualizare

Este uzual să se întâlnească.

Se cheamă în ordine inversă decât constructorii.

Dacă vrem să eliminăm porțiuni alocate dinamic și pentru clasa derivată dar facem upcasting trebuie să folosim destructori virtuali.

2. Polimorfismul la execuţie prin funcţii virtuale

Destructori si virtualizare

```
class Base1 {public:  ~Base1() { cout << "~Base1()\n"; } };

class Derived1 : public Base1 {public:  ~Derived1() { cout << "~Derived1()\n"; } };

class Base2 {public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {public:  ~Derived2() { cout << "~Derived2()\n"; } };

int main() {
  Base1* bp = new Derived1;
  delete bp; // Afis: ~Base1()
  Base2* b2p = new Derived2;
  delete b2p; // Afis: ~Derived2()  ~Base2()
}
```

2. Polimorfismul la execuţie prin funcţii virtuale

Destructori virtuali puri

Utilizare: recomandat să fie utilizat dacă mai sunt și alte funcții virtuale.
Restricție: trebuiesc definiți în clasă (chiar dacă este abstractă).
La moștenire nu mai trebuiesc redefiniti (se construiește un destructor din oficiu)
De ce? Pentru a preveni instantierea clasei.
Obs. Nu are nici un efect dacă nu se face upcasting.

```
class AbstractBase {
public:
 virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?
int main() { Derived d; }
```

2. Polimorfismul la execuţie prin funcţii virtuale

Funcţii virtuale in destructori

La apel de funcție virtuală din funcții normale se apelează conform VPTR
În destructori se face early binding! (apeluri locale)
De ce? Pentru că acel apel poate să se bazeze pe porțiuni deja distruse din obiect

```
class Base { public:
  virtual ~Base() {  cout << "~Base1()\n";  this->f();   }
  virtual void f() { cout << "Base::f()\n"; }
};
class Derived : public Base { public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived;
  delete bp; // Afis: ~Derived() ~Base1() Base::f()
}
```

2. Polimorfismul la execuţie prin funcţii virtuale

Downcasting

Folosit in ierarhii polimorfice (cu funcţii virtuale).

Problema: upcasting e sigur pentru că respectivele funcţii trebuie să fie definite în bază, downcasting e problematic.

Explicit cast prin: **dynamic_cast**

Dacă ştim cu siguranţă tipul obiectului putem folosi "static_cast".

Static_cast întoarce pointer către obiectul care satisface cerinţele sau 0.

Foloseşte tabelele VTABLE pentru determinarea tipului.

2. Polimorfismul la execuţie prin funcţii virtuale

Downcasting

```
class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  Dog* d1 = dynamic_cast<Dog*>(b);  // Afis - 0; Try to cast it to Dog*:
  Cat* d2 = dynamic_cast<Cat*>(b);// Try to cast it to Cat*:
  // b si d2 retin aceeasi adresa
  cout << "d1 = " << d1 << endl;
  cout << "d2 = " << d2 << endl;
cout << "b  = " << b << endl;
}
```

2. Polimorfismul la execuţie prin funcţii virtuale

Downcasting

```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Upcast: normal and OK
  // More explicit but unnecessary:
  s = static_cast<Shape*>(&c);
  // (Since upcasting is such a safe and common
  // operation, the cast becomes cluttering)
  Circle* cp = 0;
  Square* sp = 0;
// Static Navigation of class hierarchies
requires extra type information:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Static navigation is ONLY an efficiency
hack;
  // dynamic_cast is always safer. However:
  // Other* op = static_cast<Other*>(s);
  // Conveniently gives an error message,
while
  Other* op2 = (Other*)s;
  // does not
}
```

3. Tratarea excepțiilor în C++

O excepție este o problemă care apare în timpul execuției unui program.

O excepție C++ este un răspuns la o circumstanță excepțională care apare în timpul rulării unui program, (probleme la alocare, încercare de împărțire la zero, etc.)

- automatizarea procesării erorilor

- try, catch, throw

- block try aruncă excepție cu throw care este prinsă cu catch

- după ce este prinsă se termină execuția din blocul catch şi se dă controlul "mai sus", nu se revine la locul unde s-a făcut throw (nu e apel de funcție).

3. Tratarea excepțiilor în C++

```
try {
  // try block
}
catch (type1 arg) {
  // catch block
}
catch (type2 arg) {
  // catch block
}
catch (type3 arg) {
  // catch block
}...
catch (typeN arg) {
  // catch block
}
```

tipul argumentului arg din catch arată care bloc catch este executat

dacă nu este generată excepție, nu se execută nici un bloc catch

instrucțiunile catch sunt verificate în ordinea în care sunt scrise, primul de tipul erorii este folosit

3. Tratarea excepțiilor în C++

Observații:

- dacă se face throw şi nu există un bloc try din care a fost aruncată excepția sau o funcție apelată dintr-un bloc try: eroare

- dacă nu există un catch care să fie asociat cu throw-ul respectiv (tipuri de date egale) atunci programul se termină prin terminate()

- terminate() poate să fie redefinită să facă altceva

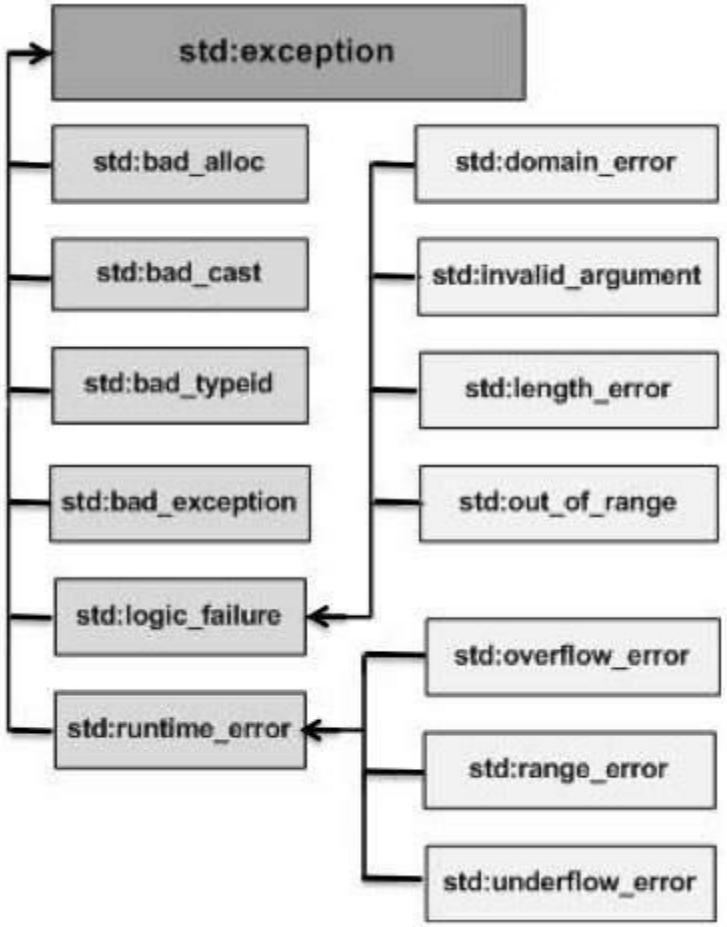
3. Tratarea excepțiilor în C++

Semnalarea unei posibile erori la alocarea de memorie: bad_alloc

```
class TestTry {
   int *v, n;
      public:
      TestTry(int a) {
           try {
               v = new int[a];
           }
           catch (bad_alloc
Nume_Var)        {
                cout << "Allocation Failure\n";
                exit(EXIT_FAILURE);
           }
          n = a;
       }
};
int main() {
TestTry T(4);
}
```

3. Tratarea excepțiilor în C++

Exceptii standard de biblioteca <exception>



Sursa: https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

3. Tratarea excepțiilor în C++

Tipul aruncat coincide cu tipul parametrului blocului catch

```
void Test_Throw_ok () {
     try {
         throw 10;
     }
     catch (int x)       {
         cout << "Exceptie 10\n";
     }
}

int main() {
   Test_Throw_ok();
}
```

Excepția este prinsă; se afișează expresia din blocul catch

```
void Test_Throw_ok () {
     try {
         throw 10;
     }
     catch (char x)       {
         cout << "Exceptie 10\n";
     }
}

int main() {
   Test_Throw_ok();
}
```

Excepția nu este prinsă

3. Tratarea excepțiilor în C++

Aruncarea unei excepții dintr-o funcție (throw în funcție)

```
class TestTry {
   int *v, n;
      public:
      TestTry(int a) { … }
      void Test_Throw_Functie() {
            try {
               Test(5);
               Test(200);
               Test(-300);
               Test(22);
            }
            catch (int x)        {
                cout << "Exceptie pe valoarea " << x <<"\n";
            }
      }
};
```

```
void Test(int x)
{
    cout << "In functie x = " << x <<"\n";
    if (x < 0) throw x;
}

int main() {
TestTry T(4);
T.Test_Throw_Functie();
}
```

In functie x = 5
In functie x = 200
In functie x = -300
Exceptie pe valoarea -300

3. Tratarea excepțiilor în C++

Try-catch local, în funcție, se continuă execuția programului

```
class TestTry {
   int *v, n;
      public:
      TestTry(int a) { … }
      void Test_Try_Local()
     {
        int x;
        x =  -25;
        Try_in_functie(x);
        x =  13;
        Try_in_functie(x);
        n = x;
        cout << n;
      }
};
```

```
void Try_in_functie(int x)
{
    try
   {
         if (x < 0) throw x;
    }
    catch(int x)
    {
   cout << “Exceptie pe valoarea “ << x
<<“\n“;
    }
}
int main() {
TestTry T(4);
T.Test_Try_Local();
}
```

Exceptie pe valoarea -25
13

3. Tratarea excepțiilor în C++

Excepții multiple; catch (...)

```
void Exceptii_multiple(int x){
    try{
       if (x < 0) throw x; //int
       if (x == 0) throw 'A'; //char
       if (x > 0) throw 12.34; //double
    }
    catch(...) {
      cout << "Catch macar una!\n";
    }
Catch(int){...}
}

int main(){
    Exceptii_multiple(-52);
    Exceptii_multiple(0);
    Exceptii_multiple(34);
}
```

3. Tratarea excepțiilor în C++

- aruncarea de erori din clase de bază şi derivate
- un catch pentru tipul de bază va fi executat pentru un obiect aruncat de tipul derivat
- să se pună catch-ul pe tipul derivat primul şi apoi catchul pe tipul de bază

```
class B { };
class D: public B { };
int main()
{
      D derived;
      try {     throw derived;   }

      catch(B b) {     cout << "Caught a base class.\n";   }

      catch(D d) {     cout << "This won't execute.\n";   }
return 0;
}
```

3. Tratarea excepțiilor în C++

Observații:

void Xhandler(int test) throw(int, char, double)

- se poate specifica ce excepții aruncă o funcție

- se restricționează tipurile de excepții care se pot arunca din funcție

- un alt tip nespecificat termină programul:
  - apel la unexpected() care apelează abort()
  - se poate redefini

- re-aruncarea unei excepții: throw; // fără excepție din catch

Tratarea excepțiilor în C++

La definiţia unei funcţii (metode), se poate preciza lista tipurilor de excepţii care pot fi generate în cadrul funcţiei.

void Functie (int test) throw(int, char)

- se poate specifica ce excepții aruncă o funcție

- se restricționează tipurile de excepții care se pot arunca din funcție

- un alt tip nespecificat termină programul:
  - apel la unexpected() care apelează abort()
  - se poate redefini

Tratarea excepțiilor în C++

Exemplu funcţie care precizeaza lista tipurilor de excepţii

```
void Functie(int x) throw (int,char)
{
    if (x < 0) throw x;
    if (x == 0) throw 'a';
    if (x > 0) throw 1.2;
}

int main()
{
    try
    {
//          Functie(-1);
//          Functie(0);
        Functie(1);
    }
    catch (int a){ cout << "int: " << a; }
    catch (char a){ cout << "char: " << a; }
    catch (double a){ cout << "double: " << a; }
    return 0;
}
```

Observaţie: lista cu tipurile de exceptii poate fi nulă, caz în care nu se acceptă nici o eroare:

Unele compilatoare pot da warninguri, altele termina executia abrubt: "terminate called after throwing an instance of 'double' "

Tratarea excepțiilor în C++

Rearuncarea unei exceptii

- re-aruncarea unei excepții: throw; // fără excepție din catch

```
void Rearuncare_exceptie(int x)
{
    try{
        if (x < 0) throw x;
        cout<<"A\n";
    }
    catch(int x) {
        cout<<"B\n";
        throw;
    }
    cout<<"C\n";
}

int main()
{
    try
    {
        Rearuncare_exceptie(-1);
    }
    catch (int a){ cout << "D\n"; }
    cout << "E\n";
}
```

Se afiseaza:
B
D
E

Tratarea excepțiilor în C++

Implementarea unei ierarhii de clase de excepţii pornind de la std::exception

Varianta C++98 – Detalii despre <exception> si toate functiile sale membre se pot gasi: https://www.cplusplus.com/reference/exception/exception/

```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
}
```

```
class MyException : public exception {
    public:
   const char * what () const throw () { return "Particularizat\n"; }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      cout << "Prins\n";
      cout << e.what() << "\n";
   } catch(std::exception& e) {
      cout << "Alte erori\n";
   }
   return 0;
}
```

Tratarea excepțiilor în C++

Implementarea unei ierarhii de clase de excepţii pornind de la std::exception

```
class MyException : public exception {
    public:
   const char * what () const throw () { return "Exceptie\n"; }
};

class Exceptie_matematica : public MyException
{
public:
   const char * what () const throw () { return "Exceptie matematica\n"; }
};

int main() {
   try {
      ///throw MyException();
      throw Exceptie_matematica();
   }
   catch(MyException& e) { cout << e.what() << "\n"; }
   catch(Exceptie_matematica& e) { cout << e.what() << "\n"; }
   return 0;
}
```

Se afiseaza Exceptie Matematica

3. Tratarea excepțiilor în C++

XVIII. Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afişează pentru o valoare întreagă citită egală cu 7, în caz negativ spuneți de ce nu este corect.

```
#include <iostream.h>
float f(int y)
{ try
  {  if (y%2) throw y/2;
  }
  catch (int i)
  {  if (i%2) throw;
     cout<<"Numarul "<<
  }
  return y/2;
}
int main()
{ int x;
  try
  {  cout<<"Da-mi un nu
     cin>>x;
     if (x) f(x);
     cout<<"Numarul "<<
  }
  catch (int i)
  {  cout<<"Numarul "<<
  }
  return 0;
}
```

```
Da-mi un numar intreg: -2   Numarul -2 nu e bun!
Da-mi un numar intreg: -1   Numarul 0 nu e bun!
Numarul -1 nu e bun!
Da-mi un numar intreg: 0   Numarul 0 nu e bun!
Da-mi un numar intreg: 1   Numarul 0 nu e bun!
Numarul 1 nu e bun!
Da-mi un numar intreg: 2   Numarul 2 nu e bun!
Da-mi un numar intreg: 3   Numarul 1 e bun!
Da-mi un numar intreg: 4   Numarul 4 nu e bun!
Da-mi un numar intreg: 5   Numarul 2 nu e bun!
Numarul 5 nu e bun!
Da-mi un numar intreg: 6   Numarul 6 nu e bun!
Da-mi un numar intreg: 7   Numarul 3 e bun!
Da-mi un numar intreg: 8   Numarul 8 nu e bun!
Da-mi un numar intreg: 9   Numarul 4 nu e bun!
Numarul 9 nu e bun!
Da-mi un numar intreg: 10   Numarul 10 nu e bun!
```

Perspective

Cursul 8:

Recapitulare finala Mostenire si Tratarea exceptiilor (+ toate kahoot-urile nedate)

Alte utilizari smart pointers
Copy constructor, operator =, destructor pt clasele cu atribute de tip pointer

Clone, copy&swap,

RAII (*Resource Acquisition Is Initialization*)