Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Click to add text

Andrei Păun
Anca Dobrovăț

An universitar 2021 – 2022
Semestrul II
Seriile 13, 14 şi 15

Curs 8

Agenda cursului

1. Recapitulare – Polimorfism la executie
2. Operatorii de cast
3. Downcasting si upcasting
4. Tratarea exceptiilor

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.

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.

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).

Polimorfismul la execuţie prin funcţii virtuale

Cum se face late binding

```
class NoVirtual {  int a;
public:
  void x() const {}
  int i() const { return 1; } };

class OneVirtual {  int a;
public:
  virtual void x() const {}
  int i() const { return 1; } };

class TwoVirtuals {  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; } };
```

```
int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
}
```

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;
}
```

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).

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”.

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

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.

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()
}
```

Polimorfismul la execuţie prin funcţii virtuale

Destructori virtuali puri

Utilizare: recomandat să fie utilizat dacă mai sunt şi alte funcţii virtuale.
Restricţie: trebuiesc definiţi în clasă (chiar dacă este abstractă).
La moştenire nu mai trebuiesc redefiniti (se construieşte 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; }
```

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()
}
```

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).

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.

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;
}
```

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
}
```

Operatorii de cast

C++ are 5 operatori de cast:

1) operatorul traditional mostenit din C;

2) dynamic_cast;

3) static_cast;

4) const_cast;

5) reinterpret_cast.

Operatorii de cast

Dynamic_cast

- daca vrem sa schimbam tipul unui obiect la executie;
- **se verifica daca un downcast este posibil (si deci valid);**
- daca e valid, atunci se poate schimba tipul, altfel eroare.

Sintaxa:

dynamic_cast <target-type> (expr)

- target-type trebuie sa fie un pointer sau o referinta;

Dynamic_cast **schimba tipul unui pointer/referinte** intr-un alt tip **pointer/referinta**.

Operatorii de cast

Dynamic_cast

Scop: cast pe tipuri polimorfice;

Exemplu:
class B{virtual …};
class D:B {... };

- un pointer D* poate fi transformat oricand intr-un pointer B* (pentru ca un pointer catre baza poate oricand retine adresa unei derivate);
- invers este necesar operatorul dynamic_cast.

In general, dynamic_cast reuseste daca pointerul (sau referinta) de transformat este un pointer (referinta) catre un obiect de tip target-type, sau derivat din aceasta, altfel, incercarea de cast esueaza (dynamic_cast se evalueaza cu null in cazul pointerilor si cu bad_cast exception in cazul referintelor.

Operatorii de cast

Dynamic_cast

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // base pointer points to Derived object

dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK

bp = &b_ob; // base pointer points to Base object

dp = dynamic_cast<Derived *> (bp); // error
```

Operatorii de cast

Dynamic_cast

```
class Base { public:
    virtual void f()
    {
        cout << "Inside Base\n";
    }
};
class Derived : public Base {
public:
    void f()
    {
        cout << "Inside
Derived\n";
    }
};
```

```
int main() {
    Base *bp, b_ob;
    Derived *dp, d_ob;

    dp = dynamic_cast<Derived *> (&d_ob);
    if(dp) { cout << "from Derived * to Derived* OK.\n";
        dp->f();    }
    else cout << "Error\n";

    bp = dynamic_cast<Base *> (&d_ob);
    if(bp) { cout << "from Derived * to Base * OK.\n";
        bp->f();    }
    else cout << "Error\n";

    bp = dynamic_cast<Base *> (&b_ob);
    if(bp)    {  cout << "from Base * to Base * OK.\n";
        bp->f();    }
    else  cout << "Error\n";
```

Operatorii de cast

Dynamic_cast

```
class Base { public:
    virtual void f()
    {
        cout << "Inside Base\n";
    }
};
class Derived : public Base {
public:
    void f()
    {
        cout << "Inside
Derived\n";
    }
};
```

```
/*    Base *bp, b_ob;
    Derived *dp, d_ob; */

    dp = dynamic_cast<Derived *> (&b_ob);
    if(dp)  cout << "Error\n";
    else
cout << "Cast from Base* to Derived* not OK.\n";

      bp = &d_ob; // bp points to Derived object
    dp = dynamic_cast<Derived *> (bp);
    if(dp)  {
        cout << "Casting bp to a Derived * OK\n" <<
            "because bp is really pointing\n" <<
            "to a Derived object.\n";
        dp->f();
    }
    else cout << "Error\n";
```

Operatorii de cast

Dynamic_cast

```
class Base { public:
    virtual void f()
    {
        cout << "Inside Base\n";
    }
};
class Derived : public Base {
public:
    void f()
    {
        cout << "Inside
Derived\n";
    }
};
```

```
/* Base *bp, b_ob;
    Derived *dp, d_ob; */

    bp = &b_ob; // bp points to Base object
    dp = dynamic_cast<Derived *> (bp);
    if(dp)  cout << "Error";
    else    { cout << "Now casting bp to a Derived *\n
is not OK because bp is really \n pointing to a Base
object.\n";    }

    dp = &d_ob; // dp points to Derived object
    bp = dynamic_cast<Base *> (dp);
    if(bp)    { cout << "Casting dp to a Base * is OK.\n";
        bp->f();
    }
    else  cout << "Error\n";
    return 0;
}
```

Operatorii de cast

Dynamic_cast Afisare:

```
Cast from Derived * to Derived * OK.
Inside Derived
Cast from Derived * to Base * OK.
Inside Derived
Cast from Base * to Base * OK.
Inside Base
Cast from Base * to Derived * not OK.

Casting bp to a Derived * OK
because bp is really pointing
to a Derived object.
Inside Derived

Now casting bp to a Derived *
is not OK because bp is really
pointing to a Base object.
Casting dp to a Base * is OK.
Inside Derived
```

Operatorii de cast

Static_cast

- este un substitut pentru operatorul de cast clasic;
- lucreaza pe tipuri nepolimorfice;
- poate fi folosit pentru orice conversie standard;
- ne se fac verificari la executie (run-time);

Sintaxa: static_cast <type> (expr)

Expl:

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for(i=0; i<10; i++)
        cout << static_cast<double> (i) / 3 << " ";
    return 0;
}
```

Operatorii de cast

Const_cast

- folosit pentru a rescrie, explicit, proprietatea de const sau volatile intr-un cast (elimina proprietatea de a fi constant);

- tipul destinatie trebuie sa fie acelasi cu tipul sursa, cu exceptia atributelor const / volatile.

Sintaxa: const_cast <type> (expr)

Operatorii de cast

Const_cast Exemplu - pointer

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

void sqrval(const int *val) {
    int *p;
// cast away const-ness.
    p = const_cast<int *> (val);
    *p = *val **val; // now, modify object through v
}
int main()
{
    int x = 10;
    cout << "x before call: " << x << endl;
    sqrval(&x);
    cout << "x after call: " << x << endl;
    return 0;
}
```

Operatorii de cast

Const_cast Exemplu - referinta

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

void sqrval(const int &val) {
// cast away const on val
const_cast<int &> (val) = val * val;
}

int main()
{
int x = 10;
cout << "x before call: " << x << endl;
sqrval(x);
cout << "x after call: " << x << endl;
return 0;
}
```

Operatorii de cast

Reinterpret_cast

- converteste un tip intr-un alt tip fundamental diferit;

Sintaxa: reinterpret_cast <type> (expr)
Expl:

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

int main()
{
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
    return 0;
}
```

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).

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

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

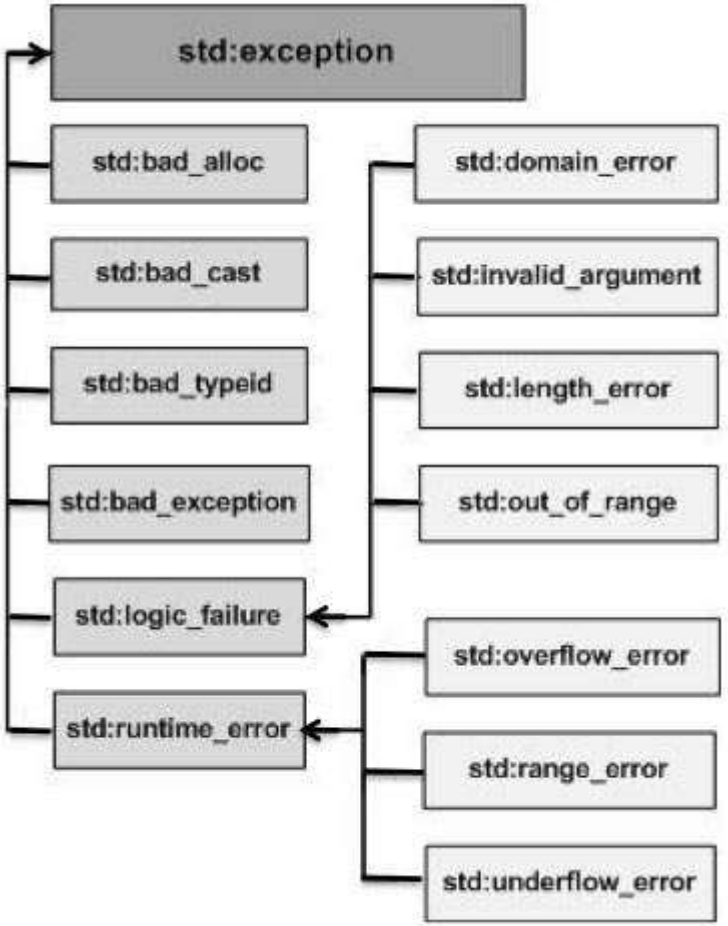
Tratarea excepțiilor în C++

Exemplu - 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); }
```

Tratarea excepțiilor în C++

Exceptii standard de biblioteca <exception>



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

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

Tratarea excepțiilor în C++

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

```
class TestTry {
public:
      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;
T.Test_Throw_Functie();
}
```

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

Facultatea de Matematica si Informatica
Universitatea din Bucuresti

Tratarea excepțiilor în C++

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

```
class TestTry {
  public:
      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;
T.Test_Try_Local();
}
```

Exceptie pe valoarea -25
13

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(int){…}
}
    catch(...) {
      cout << "Catch macar una!\n";
    }
int main(){
    Exceptii_multiple(-52);
    Exceptii_multiple(0);
    Exceptii_multiple(34);
}
```

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;
}
```

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++

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!
```

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/

```
1 class exception {
2 public:
3   exception () throw();
4   exception (const exception&) throw();
5   exception& operator= (const exception&) throw();
6   virtual ~exception() throw();
7   virtual const char* what() const throw();
8 }
```

```
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

Tratarea excepțiilor în C++

Functia terminate()

```
class A  { };
class B  { };

void f() { throw A(); }
void g() { throw B(); }

void my_terminate() {
    cout << "Terminate personalizat\n";
    abort();
}

void (*old_terminate)()
    = set_terminate(my_terminate);
```

```
int main()
{
    try
    {
        //f();
        g();
    }
    catch (A)
    {
        cout << "Exceptie A.\n";
    }
}
```

Se afiseaza Terminate personalizat si apoi se termina programul, incorect.

Tratarea excepțiilor în C++

Functia unexpected()

```
void g()
{
    throw "Surpriza.";
}

void f(int x) throw (int)
{
    if (x <= 0) throw 123;
    g();
}

void my_unexpected() {
  cout << "Exceptie neprevazuta.";
  exit(1);
}
```

```
int main()
{
    set_unexpected(my_unexpected);
    try {
        f(10);
    } catch (int) {
        cout << "Exceptie int" << endl;
    } catch (double)  {
        cout << "Exceptie double" << endl;
    }
    return 0;
}
```

Vedeti ce se intampla sub compilatorul g++! Surpriza!

Perspective

Cursul 9:

Templates/sabloane