Programare II
Programare Orientată Obiect

Curs 5

Curs anterior

- Supraîncărcarea operatorilor

Curs curent

- Supraîncărcarea operatorilor unari
- Supraîncărcarea operatorilor
  - Conversii de tip
- Excepţii
  - Aruncarea excepţiilor
  - Prinderea excepţiilor
  - Constructor

Supraîncărcarea operatorilor unari

- Operatorii unari
  - ++, --, !, -, +
  - Funcţii membre non-statice, fără argumente
  - Funcţii non-membre, cu un argument
    - Argumentul trebuie să fie o referinţă la obiectul clasei

Supraîncărcarea operatorilor unari

```
class Punct
{
private:
   double m_x, m_y;
public:
   Punct (double x=0.0, double y=0.0):
      m_x(x), m_y(y)
   {
   }
   // Convertește un punct în echivalen
    tul lui avânt coordonatele negativ
   Punct operator- () const;

   // Întoarce adevărat dacă punctul est
    e setat în origine
   bool operator! () const;
};
```

```
PunctPunct::operator- () const
{
   return Punct(-m_x, -m_y);
}
bool Punct::operator! () const
{
   return (m_x == 0.0 && m_y == 0.0);
}

int main(){
  Punct p;
  if (!p)
    cout << “Punctul nu este in origine”;
  else
   cout << “Punctul este in origine”;
  cout << -p;
}
```

Curs curent

- Supraîncărcarea operatoriilor unari
- Supraîncărcarea operatorilor
  - Conversii de tip
- Excepţii
  - Aruncarea excepţiilor
  - Prinderea excepţiilor
  - Constructor

Conversii de tip

- Reguli
  - La evaluarea unei expresii se efectuează conversii sistematice de la tipurile întregi scurte la tipul int sau unsigned int. Apoi, pentru fiecare operand, dacă operanzii au tipuri diferite, se efectuează conversia necesară pentru a obţine ambii operanzi de acelaşi tip.
  - La atribuire se efectuează conversia valorii atribuite la tipul operandului care primeşte valoarea
  - La transferul parametrilor se efectuează conversia valori fiecărui parametru efectiv la tipul parametrului formal

Conversii de tip

- Conversii posibile
  - De la tip de bază la tip clasă
  - De la tip clasă la tip de bază
  - De la tip clasă la tip clasă
- Cum se pot realiza?
  - Supraîncărcarea operatorului de cast
  - Prin intermediul constructorilor

Prin intermediul constructorului

- Conversia de a un tip de dată (predefinit sau altă clasă definită de utilizator) la clasă
- Prototip
  - X(tipDeData)
- Apelat
  - Declaraţii de tipul X x=valoare
  - Apelul funcţiilor dacă este nevoie de o conversie la tipul de date

Prin intermediul constructorului

- Exemplu

```
class Complex {
    int re, im;

 public:
    Complex(float f){
        re = (int) f;
    }
    Complex(char *);
};
```

```
void f(){
    Complex c= 4.8f;
    Complex c1= “2+6i”;
}
void fct(Complex c) {
…
}

int main() {
  f();
  fct(“2+6i”);
}
```

Operatorul de cast

- Conversie de la tip de clasă la tip de bază/altă clasă
- Sintaxă
  - operator tipDeData ();
- Observaţii
  - Operatorul este întotdeauna funcţie membre ale clasei
  - Tipul de return este implicit tipul operatorului
  - Nu are parametrii deoarece primeşte implicit adresa obiectului curent

Operatorul de cast

- Exemplu

```
class Complex {
    int re, im;
 public:
    operator double(){
        return re;
     }
};
```

```
void f(){
    Complex c(5,6), e(9,3);
    double d = c;
    double dd = c + e;
}
```

Operatorul de cast

- Supraîncărcarea operatorului de cast nu poate realiza conversii dintr-un tip fundamental în tip clasă
- Operatorul de cast este unar
- Definirea constructorilor nu permite conversia unui tip clasă la un tip de bază

Supraîncărcarea operatorilor - polimorfism

- Polimorfismul este abilitatea de a folosi o metodă în moduri diferite
  - Ad-hoc polimorfism
  - Operatorii prin supraîncărcare au comportament diferit în funcţie de tipul argumentelor
- Supraîncărcarea operatoriilor face o expresie mai uşor de citit (exprimat) – syntactic sugar

Curs curent

- Supraîncărcarea operatoriilor unari
- Supraîncărcarea operatorilor
  - Conversii de tip
- Excepţii
  - Aruncarea excepţiilor
  - Prinderea excepţiilor
  - Constructor

Tratarea excepțiilor

- Definiţie
  - O excepţie este o eroare apare în momentul execuţiei unui program (run-time).

- Exemple
  - Memorie insuficientă
  - Un fişier nu poate fi deschis
  - Un obiect invalid pentru o operaţie
  - etc

Tratarea excepţiilor

- Ce facem când apare o excepţie?
  - Terminarea programului → soluţie inacceptabilă
  - Returnarea unei valori reprezentative pentru eroare → Care ar fi un cod de eroare reprezentativ? Trebuie ca funcţia apelantă să verifice codul întors de funcţia apelată
  - Întoarce o valoare legală şi lasă programul într-o stare ilegală (errno) → apelantul trebuie să testeze starea variabilei astfel încât să nu realizeze un apel invalid
  - Apelarea unei funcţii care va trebui apelată în caz de eroare → nu există control asupra codului apelantului
- Combinarea codului uzual cu codul de tratare a excepţiilor → programe greu de citit şi întreţinut

Tratarea excepţiilor

- Ce se întâmplă dacă nu folosim un mecanism de tratare a excepţiilor?
  - Când apare o excepţie, controlul este predat sistemului de operare, de obicei
    - Un mesaj de eroare este afişat
    - Programul se termină
- Ce se întâmplă dacă folosim un mecanism de tratare a excepţiilor?
  - Programele pot înlănţui excepţiile
  - Există posibilitatea să le tratăm şi să continuăm execuţia programului

Tratarea excepțiilor. Abordare

- Codul care detectează o eroare
  - O propagă mai departe throw
  - O tratează try … catch

Tratarea excepţiilor. Abordare

Propagare

```
void operation()
{
    if(daca_un_caz_de_erore_exista)
    {
     // aruncă un obiect de
     // tip excepţie
     throw TipExceptie();
    }
     ...
}
```

Tratare

```
void f()
{
    try {
       // … cod care ar putea
       // arunca o excepţie
       …
    } catch (TipExceptie1 e1) {
        // tratează o excepţie de tip1
    } catch (TipExceptie2 e2) {
        // tratează o excepţie de tip2
    }
}
```

Tratarea excepțiilor.Exemplu

- Clasa Stiva

```
class Stiva{
   int *tab; //tablou care contine elementele listei
   int dim;
   int index; //indexul elementului curent din lista
 public:

   void adauga(int x){ /**functia adauga elementul x in stiva*/
      tab[index] = x;
      index++;
   }
   int scoate(){ /**functia scoate un element din stiva*/
      return tab[index--];
   }
};
```

Tratarea excepțiilor.Exemplu

- Există cazuri când pot apărea probleme la funcţiile
  - adauga()
  - scoate()

Tratarea excepţiilor.Exemplu

- Există cazuri când pot apărea probleme la funcţiile
  - adauga()
  - scoate()
- Probleme
  - Stiva plină
  - Stiva goală
- Cum rezolvăm?
  - Definim o clasă se excepţii
  - Obs: putem utiliza tipurile de date existente

Tratarea excepțiilor.Exemplu

Propagarea excepţiilor

```
class StivaGoala { ... };
class Stiva{
    ...
    void adauga(int x){ /**functia adauga elementul x in stiva*/
        if (index ==dim) throw “Stiva Plina!”;
        tab[index] = x;
        index++;
     }
     int scoate(){ /**functia scoate un element din stiva*/
         if (index ==0) throw StivaGoala();
         return tab[index--];
     }
};
```

Tratarea excepțiilor.Exemplu

```
int main() {
    try {
        Stiva s(1);
        s.scoate();
        s.adauga(4);
        s.adauga(5);
     } catch (StivaGoala) {
          cout << “Stiva este goala!” <<< “Exceptia prinsa: ”
    << a << endl;
      }
}
```

Codul care aruncă excepţia este încadrat într-un bloc try{…}catch{…}

Tratarea excepțiilor

- Observații
  - Un bloc try poate avea atașat unul sau mai multe blocuri catch
  - Un bloc catch poate avea următoarele forme
    - catch(tipExceptie)
      - Prinde orice exceptie de tipul specificat (tipExceptie) fară a permite regăsirea datelor stocate în excepție
    - catch(tipExceptie variabila)
      - Prinde orice exceptie de tipul specificat (tipExceptie), permite regăsirea datelor stocate în excepție
    - catch(...)
      - Prinde orice excepție fară a permite identificarea tipului excepției sau datelor stocate în ea
  - În interorul unei funcții pot exista mai multe blocuri try-catch, blocurile try-catch pot fi imbricate

Tratarea excepțiilor. Observații

- În acest caz

```
void fct(int a){
    if (a<0) throw “Eroare valoare parametru”;
    a=a+5;
}
```

Dacă se intră pe ramura de *then* a instrucțiuni *if* execuția instrucțiuni *throw* este echivalentă cu apelul instrucțiuni *return* adică întrerupe execuția funcției și codul de după instrucțiunea *throw* nu se mai execută

Tratarea excepțiilor. Observații

- În acest caz

```
void fct(int a){
    if (a<0) throw “Eroare valoare parametru”;
    a=a+5;
}
int main(){
   cout << “Inainte apel\n”;
   fct(-3);
   cout << “Dupa apel\n”
}
```

În acest caz se va afișa pe ecran doar mesajul “Inainte apel” deoarece apelul funcției *fct()* aruncă o eroare care duce la întreruperea execuției medodei *main()*.

Tratarea excepțiilor. Observații

- În acest caz

```
void fct(int a){
    if (a<0) throw “Eroare valoare parametru”;
    a=a+5;
}
int main(){
     cout << “Inainte apel\n”;
  try{
     fct(-3);
     cout << “Dupa apel\n”
  }catch (const char *ex) { cout <<“Eexeptie: ” <<ex << endl; }
  cout << “Dupa try\n”;
}
```

În acest caz se vor afișa pe ecran doar mesajele: “Inainte apel”, “Eexeptie : Eroare valoare parametru” si “Dupa try”;deoarece apelul funcției *fct()* aruncă o eroare si codul care urmeaza acestei linii pana la intalnirea unui bloc try care trateaza acesta eroare nu se mai executa

Gruparea excepțiilor

- Excepţiile de obicei sunt grupate în familii → se poate folosi moştenirea

```
class MathException {
     public: virtual string getDescription() {
        cout << “Math ex”;
    }
};
 class Overflow : public MathException {
     public: string getDescription() {
        cout << “Overflow ex”;
     }
};
class Underflow : public MathException {
 };
```

Gruparea excepțiilor

- Excepţiile de obicei sunt grupate în familii → se poate folosi moştenirea

```
void f() {
    try {
        // ... folosirea functiilor matematice
    } catch(Overflow) {
        // tratarea exceptiei Overflow
    } catch(MathException &e) {
        // tratarea oricărei excepștii MathExceptions
        // care nu este Overflow
        cout << e.getDescription();
        // apelul fucției MathException::getDescription()
    }
}
```

Gruparea excepţiilor

- Observaţii
  - Argumentul din cauza de catch nu este obligatoriu
  - Ierarhiile de excepţii măresc robusteţea abordării
  - Se poate folosi şi moştenire multiplă
  - Este bine să prindem prima dată excepţiile de tip subclasă şi apoi cele de tip supraclasă

Specificarea excepţiilor

- Lista se excepţii aruncate de o funcţie se poate specifica ca parte a declarării funcţiilor
- Sintaxă
  - Tip nume_functie (lista_argumente) throw (listă_de_excepţii)
- Exemplu
  - void adunare(Marice &m1, Matrice &m2) throw (ExceptieMatematica, AlocareIncorecta)
- Dacă lista de excepţii lipseşte o funcţie poate arunca orice excepţie
- void f() throw(); - nu aruncă nici o excepţie

Specificarea excepţiilor

- Observaţii
  - Specificarea excepţiilor trebuie inclusă atât în declaraţia cât şi în definiţia funcţiei
  - O funcţie virtuală poate fi supradefinită doar de o funcţie care este cel puţin la fel de restrictivă ca funcţie iniţială
  - Dacă o altă excepţie este aruncată în afara celor specificate în clauza throw → apel std::unexped(), care apelează std::terminate() care apelează funcţie abrout()
  - Se poate supradefini comportamentelor funcţiilor std::unexped(), std::terminate() prin suprascrierea handlerelor set_unexpected respectiv set_terminate

Excepții neprinse

- Dacă o excepţie este aruncată dar nu este prinsă, funcţia std::terminate va fi apelată
- Pentru a prinde toate excepţiile netratate se poate utiliza următorul bloc

```
int main() {
    try {
        // cod care poate arunca excepții
    } catch(...) {
        // tratează orice excepție neprinsă până acum
    }
}
```

- Aruncarea unei excepţii este mai puţin costisitoare decât apelul unei funcţii
- STL – biblioteca standard a C++ propune o ierarhie de excepţii

Re-transmiterea excepţiilor

- După ce s-a realizat o corecţie a datelor într-o clauză catch, excepţia poate fi aruncată pentru a fi procesată mai departe

```
void func() {
    try{
        //cod care aruncă o excepţie bad_alloc
    } catch (bad_alloc e) {
        // cod de abordare a excepţiei
        throw;
    }
}
```

Imbricarea blocurilor try

- Excepţiile sunt întotdeauna tratate de cel mai apropiat handler

```
try {
    try {
        throw 5;
    } catch (int x) {
        cout << x << endl;
        // excepţia va fi prinsă aici
    }
} catch (int x) {
    cout << x-5 << endl;
}
```

Constructori și excepții

- Cum se raportează erorile în constructor?

```
class Vector {
    public:
        class BadSize
        {
        } ;

        Vector(int sz)
        {
            if(sz<MAX_SIZE) throw BadSize();
           // codul
        }
};
```

Constructori și excepții

- Cum se trateaza exceptiile aruncate de constructori?

```
void f(int size) {
    try {
        Vector v(size);
        cout << v;
    } catch(Vector::BadSize& bs) {
    }
    cout << “Return successfully.”;
    return;
};
```

Constructori și excepții

- Cum se trateaza exceptiile aruncate de constructori?

```
class Map{
    public:
        Vector val, key;
        Map(int sz)
            try {
                : val(sz),key(sz)
                 {
                 }
             } catch(Vector::BadSize& bs) {
            }
};
```

Intrebare

- Fie clasa Coordonate care are două atribute:
  - Longitudinea care este o valoare în intervalul [-90, 90]
  - Latitudine care este o valoare în intervalul [-180, 180]

- Propuneţi două funcţii de setare/validare a valorilor atributelor clasei Coordonate
- Propuneţi un constructor pentru clasa Coordonate

Curs viitor

- Exemplu compozitie