# Programare orientată obiect

Cursul 14

rev 1.1/PP/20140108

# Sumar

- RTTI
- Extensii C++ 11
- .NET și C++
  - Exemplu
- Discuții finale

# RTTI – Run-Time Type Information

- Posibilitatea de obținere de informații despre tipurile de date la execuție
  - vs. Compilare
- Util atunci cînd nu se cunoaște tipul de dată decît la execuție
- Uzual, se aplică pe referințe și pointeri la o clasă de bază pentru a opera cu tipurile concrete

# RTTI

- Operatori
  - dynamic_cast
  - typeid
- Clase
  - type_info

# type_info

- Informații despre tipuri de date
- Obiecte returnate de typeid
- Include și:
  - Denumirea tipului
  - operatori de comparație (== și !=)

# typeid

- Identificare tipurilor obiectelor
- Returnează o referință la un obiect de tip type_info
- Se aplică pe:
  - Pointeri
  - Referințe
  - Tipuri fundamentale
  - Tipuri utilizator
- Dacă pointerul este NULL este lansată excepția
  - bad_typeid

# typeid

```
char *sir = "Test";
cout<<typeid(sir).name()<<endl;
```

Ieșire: `char const *`

```
class Patrat { /* */} patrat;
cout<<typeid(patrat).name()<<endl;
```

Ieșire: `class Patrat`

# dynamic_cast

- Conversie între tipuri polimorfice
- Conversie la execuție
- Conversie sigură:
  - Pointer/referință convertit
  - În caz de incompatibilitate între tipuri:
    - NULL/nullptr (pointeri)
    - sau excepție bad_cast (referințe)
- Scopul principal: asigurarea de conversii de tip *downcast* sigure
  - Pointer/Referință clasa de bază => Pointer/Referință clasa derivată
- **dynamic_cast**<TIP_C>(ob_de_convertit)

# dynamic_cast

## Conversie reușită

```
Figura *pf;  Patrat *pp;

pf = new Patrat();
pp = dynamic_cast<Patrat *>(pf);

if (pp != NULL)  {
        cout<<"conversie reusita!"<<endl;
        pp->getLatura() = 20;
}  else {
        cout<<"conversie nereusita!"<<endl;
}
delete pf;
```

## Conversie nereușită

```
Figura *pf;  Patrat *pp;

pf = new Figura();
pp = dynamic_cast<Patrat *>(pf);  /*vs.*/  // pp = (Patrat *)pf;

if (pp != NULL)  {
        cout<<"conversie ok!"<<endl;
        pp->getLatura() = 20;
}  else {
        cout<<"conversie nereusita!"<<endl;
}
delete pf;
```

# Alte conversii

- Operatorul de conversie C
  - (tip)Var
- static_cast
  - Conversie între tipuri verificabile la compilare
  - Portabil
- const_cast
  - Conversii între pointeri constanți/non-constanți și pointeri non-constanți/constanți
- reinterpret_cast
  - Conversii între tipuri diferite

# C++ 11

- Determinarea automată a tipului de date (**auto**)
- Constantă pentru pointeri nuli (**nullptr**)
- Funcții virtuale: **override** și **final**
- Constante enumerative puternic tipizate (**enum class**)
- Pointeri inteligenți
- Funcții anonime (expresii **Lambda**)
- Aserțiuni statice (**static_assert**)
- referințe rvalue

# C++ 11

- Funcții nemembre begin() și end()
- Containere noi
- Ciclări bazate pe intervale

# Determinarea automată a tipului de date (auto)

- Utilizat pentru tipuri de date
- Cuvînt cheie: **auto**
- Compilatorul determină tipul
- Poate fi utilizat la orice nivel (bloc, spațiu de nume, inițializare variabile contor etc.)
- Nu poate fi utilizat ca tip returnat de funcții

# Determinarea automată a tipului de date (auto)

```
auto x = 20;
auto y = 20.5;
auto sir = "sir de test";

cout<<x<<typeid(x).name()<<endl;
cout<<y<<typeid(y).name()<<endl;
cout<<sir<<typeid(sir).name()<<endl;
```

# Constantă pentru pointeri nuli (nullptr)

- Pointer nul
- Cuvînt cheie: **nullptr**
- De preferat față de NULL sau 0
- Exemple:
  - int *pi = nullptr;
  - if (pi != nullptr) //…
- Situații specifice
  - O funcție supraîncărcată
    - **void f(int)** și **void f(char *)**
  - Apelurile:
    - **f(NULL);**//prima variantă
    - **f(nullptr);**//a două variantă

# Funcții virtuale: override și final

- **override**
  - Metoda supradefinește o metodă virtuală din clasele de bază
  - Suport pentru semnalarea la compilare a prototipurilor diferite de funcții
- **final**
  - Metoda nu poate fi supradefinită în ierarhie

# Funcții virtuale: override și final

```
class Figura {
public:
        virtual char *  obtineNumeBaza() final
                {  return "Figura"; }
        virtual double aria() = 0;
};

class Patrat : public Figura {
protected:
        double l;
public:
        double aria() override { return l * l; }
};
```

```
//eroare: prototip diferit (ar fi supraîncărcare)
double aria(int lat) override
{
        return l * lat;
}
//eroare: funcție marcată ca final în clasa de bază
char * obtineNumeBaza()
{
        return "Figura";
}
```

# Constante enumerative puternic tipizate

- Constantele enumerative (**enum**)
  - Tratate ca valori întregi
  - Definite la nivel global
  - Nu poate apărea același nume
- Constante enumerative puternic tipizate (**enum class**)
  - Accesibile prin intermediul operatorului de rezoluție
    - Numele se poate repeta, numele clasei constantei fiind diferit
  - Posibilitatea precizării tipului de date

# Constante enumerative puternic tipizate

- Exemplu:

```
enum class Culoare {ROSU, VERDE, ALBASTRU};
Culoare culoare = Culoare :: VERDE;
if (culoare == Culoare:: ALBASTRU) //…
```

- Precizarea tipului
  - enum class Culoare **: char** {ROSU, VERDE, ALBASTRU };
  - enum class Culoare **: char** {ROSU = 1, VERDE = 2, ALBASTRU = 3 };

# Pointeri inteligenți

- Tipuri abstracte de date utilizate pentru gestiunea pointerilor
  - Alocare/eliberare de memorie
  - Verificare limite
  - Control copiere/mutare
- Implementează operatorii -> și *
- #include <memory>

# Pointeri inteligenți

- **unique_ptr**
  - Un singur proprietar
  - Nu se copiază sau partajează
- **shared_ptr**
  - Mai mulți proprietari (partajare)
  - Contorizarea referințelor
- **weak_ptr**
  - Pointeri partajați
  - Nu participă la contorizarea referințelor
  - Evitarea referințelor ciclice

# Pointeri inteligenți - Exemple

```
unique_ptr<Patrat> pPatrat(new Patrat());
pPatrat->aria();
//destructorul se apeleaza automat

shared_ptr<Patrat> pPatrat1(new Patrat());
shared_ptr<Patrat> pPatrat2 = pPatrat1;
pPatrat1->aria();
pPatrat2->aria();
```

# Funcții anonime (expresii Lambda)

- Utile pentru funcții simple care vor fi implementate la apel
- Utilizate frecvent în locul functorilor sau a predicatelor în algoritmi STL
  - În cazul functorilor nu mai este necesară implementarea unei clase
- Forme
  - Definire:
    - [clauză captare] (param)  {corpul_functiei}
    - [clauză captare] (param) **-> tip_returnat** {corpul_functiei}
  - Definire și apel direct
    - [clauză captare] (param)  {corpul_functiei} **(arg)**

# Funcții anonime (expresii Lambda)

- Clauză captare
  - Modul de transmitere a variabilelor din afara funcției lambda
  - [x] – variabila x prin valoare
  - [&x] – variabila x prin referință
  - [=] – orice variabilă locală prin valoare
  - [&] – orice variabilă locală prin referință
  - Pot fi combinate într-o listă: [&x, =]
    - x transmisă prin referință, celelalte variabile locale prin valoare

# Funcții anonime (expresii Lambda)

```
auto lambda = [] (int x, int y) { return x * y; };
cout<<lambda(10, 5);

vector<int> v;
//...
sort(v.begin(), v.end(), [](int x, int y) -> bool{ return (x > y);});
//sau
sort(v.begin(), v.end(), [](int x, int y) { return (x > y);});
```

# Funcții anonime (expresii Lambda)

```
int d = 4;
vector<int> v;
//...
int x = count_if(v.begin(), v.end(), [d](int x) { return (x % d) == 0; });
cout<<"Numere divizibile cu "<<d<<" in vector: " <<x <<endl;
```

# Aserțiuni statice (static_assert)

- Verificarea unor condiții la compilare
  - vs. assert
- Util pentru
  - depanarea claselor și funcțiilor generice
  - verificarea versiunilor, lungimilor tipurilor etc.
- static_assert(conditie, mesaj)
  - dacă valoarea condiției este *false* la compilare se afișează mesajul *mesaj*

# Aserțiuni statice (static_assert)

```
static_assert(sizeof(void *) == 4, "Configuratia trebuie sa fie Win32!");

template <typename T, size_t NrPuncte>
class Poligon
{
  static_assert(NrPuncte < 3, "Prea putine puncte!");
  T puncte[NrPuncte];
};
```

# referințe rvalue

- Referințe temporare pentru obiecte care urmează să fie distruse
- Tip && rvRef;
- Context:
  - rezultă un obiect temporar (cu alocare dinamică)
  - obiectul temporar nu va mai fi utilizat în continuare, fiind distrus imediat;
  - copierea conținutului va fi evitată (optimizare)
  - Exemplu
    - vect = vect1 + vect2;

# referințe rvalue

- Constructor de mutare (*move constructor*) și operator de atribuire cu mutare (*move assignment operator*)
  - Inițializează membrii sursei cu valori implicite (0, nullptr, NULL etc.)
  - Funcția std::move() returnează referința temporară
  - Clasa(Clasa &&sursa)
  - operator=(Clasa &&sursa)

# referințe rvalue

```
Produs (Produs &&s)
{
        //nu se face nici alocare, nici copiere
        nume = s.nume;
        //initializare cu valoare implicita în obiectul sursă
        s.nume = nullptr;

        cout<<"Constructor mutare"<<endl;
}
```

# Ciclări bazate pe intervale

```
vector<int> v;
v.push_back(10);
v.push_back(20);
v.push_back(30);

for (int elem : v)
{
   cout << elem<<endl;
}
```

```
for (int &elem : v)
{
   elem = elem *2;
}
```

# Funcții nemembre begin() și end()

- STL include funcțiile globale begin() și end()
- Permit obținerea unor iteratori valizi, inclusiv pentru structuri diferite de containere
- Exemplu

```
int v[] = {10, 20, 11, 25, 110, 22};
sort(begin(v), end(v));
```

# Containere STL noi

- Secvențiale
  - Listă simplu înlănțuită
    - **forward_list**<T>
  - Masiv unidimensional alocat static
    - **array**<T, N>
- Asociative
  - Mulțimi cu acces rapid la elemente
    - **unordered_set**<T>
    - **unordered_multiset**<T>
  - Tabele de dispersie
    - **unordered_map**<Cheie, T>
    - **unordered_multimap**<Cheie, T>

# Exemple de funcții STL noi

- shuffle
- is_sorted
- copy_if
- move
- find_if_not