

Curs 07

DERIVAREA

Derivarea (moștenirea) și tratarea excepțiilor



Derivarea / Moștenirea



- Permite crearea de clase noi prin reutilizarea de cod sursă
- Este cea de-a doua relație dintre clase după compunere
- Răspunde la întrebarea „is a” („este un” / „este o”) și surprinde o relație de specializare între clase
- Clasa existentă (numită și clasă de bază/părinte) este derivată/moștenită/extinsă de către o nouă clasă (clasă derivată/copil)
- Clasa derivată va moșteni toți membrii clasei de bază

Exemplu: Derivarea clasei TelefonMobil



```
class TelefonMobil
{
    //...
};
```

```
class TelefonMobilPliabil : public TelefonMobil
{
    //...
};
```

Tipuri de derivare



- Derivare publică:
 - toți membrii clasei de bază își păstrează vizibilitatea
- Derivare protected:
 - membrii publici devin protected în clasa derivată, ceilalți membri își păstrează vizibilitatea
- Derivarea private (implicită):
 - toți membrii clasei de bază devin privați în clasa derivată

Publicizarea (excepții de la tipul de derivare)



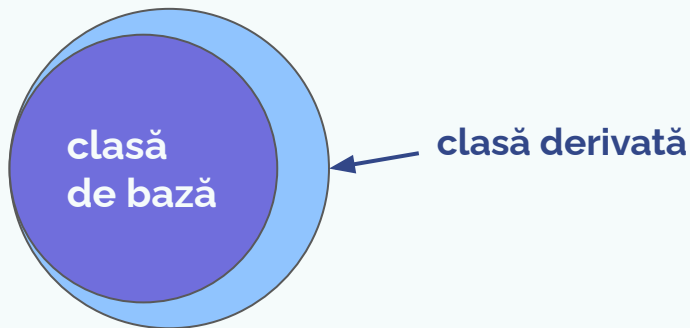
```
class TelefonMobil
{
public:
    int nivelBaterie
    //...
};

class TelefonMobilPliabil : private TelefonMobil
{
public:
    TelefonMobil::nivelBaterie;
    //...
};
```

Derivarea / Moștenirea



- Clasa derivată moștenește toate atributele și metodele clasei de bază (chiar dacă acestea sunt private)
- Atributele private nu pot fi accesate de clasa derivată, însă acestea tot se moștenesc



Derivarea / Moștenirea



- Nu toate metodele însă se moștenesc integral
- Nu se moștenesc integral:
 - constructorii (inclusiv cel de copiere)
 - destructorul
 - operatorul=
 - operatorii de cast

Apel constructor din clasa de bază



```
class TelefonMobil
{
private:
    int nivelBaterie;
    //...
public:
    TelefonMobil();
    TelefonMobil(int nivelBaterie);
};
```

```
class TelefonMobilPliabil : public TelefonMobil
{
private:
    int dimensiunePliat;
public:
    //apel implicit constructor din clasa de baza
    TelefonMobilPliabil();
    TelefonMobilPliabil(int nivelBaterie, int
    dimensiunePliat) : TelefonMobil(nivelBaterie);
    //apel explicit constructor din clasa de baza
};
```

Precizări



- Fiecare dintre constructori va fi responsabil de inițializarea părții lui din obiect
- Constructorul din clasa de bază va inițializa zona lui din obiect, iar cel din clasa derivată zona nou adăugată obiectului de bază
- **Ordinea de apel a constructorilor este bază-derivat (mai întâi se apelează constructorul din clasa de bază și mai apoi cel din clasa derivată)**

Precizări



- Fiecare dintre destructori va fi responsabil de dezalocarea zonei proprii din obiect
- Destructorul clasei de bază va dezaloca zona lui de obiect, iar cel din clasa derivată zona nou adăugată obiectului de bază
- **Ordinea de apel a destructorilor este derivat-bază (mai întâi se apelează destructorul din clasa derivată și mai apoi cel din clasa de bază)**

Upcasting



- Sau conversia derivat-bază
- Permite transformarea obiectelor de tip derivat în obiecte de tipul clasei de bază (toate telefoanele mobile pliabile sunt telefoane mobile până la urmă)
- Funcționează implicit dacă derivarea este publică
- Se aplică și la nivelul pointerilor (pointerii la TelefonMobilPliabil pot fi convertiți automat în pointeri la TelefonMobil)

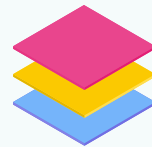
Exemplu upcasting



```
int main()
{
    TelefonMobil telefonDeServiciu;
    TelefonMobilPliabil telefonPersonal;

    telefonDeServiciu = telefonPersonal;
    //downcast-ingul nu functionaza in mod automat
    //linia de mai jos va returna eroare de compilare
    //telefonPersonal = telefonDeServiciu;
    TelefonMobil* ptm = &telefonDeServiciu;
    TelefonMobilPliabil* ptmp = &telefonPersonal;
    ptm = ptmp;
}
```

Alte precizări



- Clasa derivată poate defini membri cu același nume ca cei din clasa de bază, situație în care spunem că aceștia „ascund” membrii moșteniți
- Implicit vor fi utilizați cei din clasa derivată atunci când declarăm obiecte de acest tip, iar dacă vrem să îi folosim pe cei din clasa de bază e nevoie să utilizăm operatorul de rezoluție ::

Exemplu: Derivarea clasei TelefonMobil



```
class TelefonMobil
{
public:
    int nivelBaterie;
    void afiseazaDetalii();
};

class TelefonMobilPliabil : public TelefonMobil
{
public:
    int nivelBaterie;
    void afiseazaDetalii();
};
```

Exemplu



```
int main()
{
    TelefonMobil telefonDeServiciu;
    TelefonMobilPliabil telefonPersonal;
    //atributul este cel din clasa derivata
    telefonPersonal.nivelBaterie = 33;
    //utilizare atribut din clasa de baza
    telefonPersonal.TelefonMobil::nivelBaterie = 33;
}
```

Exemplu



```
int main()
{
    TelefonMobil telefonDeServiciu;
    TelefonMobilPliabil telefonPersonal;
    //la fel si pentru metode
    telefonPersonal.afiseazaDetalii();
    telefonPersonal.TelefonMobil::afiseazaDetalii();
}
```

Exemplu



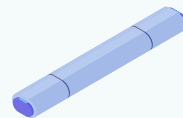
```
int main()
{
    TelefonMobil telefonDeServiciu;
    TelefonMobilPliabil telefonPersonal;
    //in cazul upcasting-ului, metodele sunt cele din clasa de baza
    telefonDeServiciu = telefonPersonal;
    //va apela metoda din clasa de baza
    //chiar daca obiectul este obtinut prin upcasting din TelefonMobilPliabil
    telefonDeServiciu.afiseazaDetalii();
}
```

Alte precizări



- Operatorii (cu excepția operator=) se moștenesc ca atare, dar vor funcționa doar pentru partea din obiect specifică clasei de bază
- Pentru a funcționa pentru noile obiecte, pot fi redefiniți
- Funcțiile friend din clasa de bază rămân friend și în clasa derivată
- Clasele friend ale clasei de bază nu sunt friend și pentru clasa derivată

Excepțiile



- În C++ avem anumite situații când datele de intrare nu sunt valide
- În aceste situații putem returna excepții din anumite funcții
- Aruncarea unei excepții înlocuiește returnarea unei valori din acea funcție
- Excepțiile, dacă nu sunt gestionate, întrerup funcționarea normală a programului și se comportă precum erorile de execuție
- În C++ o mare parte din funcțiile standard nu folosesc excepții

Exemplu: aruncarea unei excepții



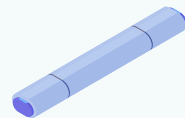
```
class TelefonMobil
{
    //...
    int* durateZilniceUtilizare;
    int nrZile;
    //permite acces citire/scriere
    int& operator[](int index)
    {
        if(index >=0 && index < nrZile)
        {
            return durateZilniceUtilizare[index];
        }
        throw exception("index invalid");
    }
};
```

Exemplu: gestionarea unei excepții



```
int main()
{
    TelefonMobil telefon;
    try
    {
        cout << telefon[2];
    }
    catch(exception e)
    {
        cout << e.what();
    }
}
```

Excepțiile



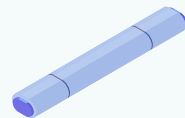
- În C++ putem returna nu doar excepții de tipul **exception** ci și alte tipuri de date (coduri de eroare de exemplu)
- Pot exista mai multe blocuri de tip catch pentru un bloc try (cel puțin unul este obligatoriu), situație în care vor fi evaluate de sus în jos
- **Întotdeauna un singur bloc catch este utilizat în cazul apariției unei excepții**

Exemplu: gestionarea unei excepții prin blocuri catch multiple



```
int main()
{
    TelefonMobil telefon;
    try
    {
        cout << telefon[2];
    }
    catch(int cod)
    {
        cout << cod;
    }
    catch(exception e)
    {
        cout << e.what();
    }
    catch(...)
    {
        cout << "exceptie necunoscuta!";
    }
}
```

Excepții personalizate



- Pot fi definite excepții personalizate prin derivarea clasei **exception**
- Dacă dorim să specificăm și mesaje de eroare, atunci putem să definim un constructor cu un parametru de tip `char*` pe care să îl trimitem mai departe către constructorul cu un parametru al clasei **exception**

Exemplu: definirea unei excepții personalizate



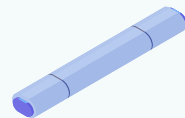
```
class index_exception : public exception
{
public:
    index_exception()
    {
    }
    index_exception(const char* message) : exception(message)
    {
    }
};
```

Exemplu: gestionarea unei excepții prin blocuri catch multiple



```
int main()
{
    TelefonMobil telefon;
    try
    {
        cout << telefon[2];
    }
    //va prinde doar erorile de tip index_exception
    catch(index_exception e)
    {
        cout << e.what();
    }
    //ordinea blocurilor catch trebuie sa fie de la cat mai specifica la cat mai general
    //daca acest bloc era primul atunci ar fi prins si exceptiile de tip index_exception
    catch(exception e)
    {
        cout << e.what();
    }
}
```

Precizări



- Excepțiile pot încetini execuția unui program și pot produce memory leaks dacă sunt generate de anumite metode (ex: constructor sau destructor)
- Nu sunt o alternativă pentru structura condițională
- Permit o modalitate standard și unitară de gestionare a situațiilor neprevăzute
- Nu sunt utilizate de către toate bibliotecile C++