Programozás II.

5. gyakorlat

Ismétlés múlt óráról

- Macska osztály
  - Konstruktor
  - nyávog()
  - példányosítás stack/heap
  - pointerek
  - default paraméter értékekkel
  - láthatóságok
  - std névtér
  - iostream

Copy konstructor

- Objektum másolásakor használjuk
  - Pont origo(0,0);
  - Pont p2 = origo;
  - Pont p3(origo);
- Fordító létrehoz egy copy konstruktort, ha mi nem csinálunk
  - ez miért nem jó mindig?

Paraméter átadási módok

- Háromféle módon adhatunk át paramétereket a függvényeinknek:
  - Érték szerint (C)

    ```
    void fuggveny( int a );
    ```
  - Cím szerint (C)

    ```
    void fuggveny( int* a );
    ```
  - Referencia szerint (Java, C++)

    ```
    void fuggveny( int& a );
    ```

Érték szerinti

- A paraméter értéke **lemásolódik** a **verembe** a függvény hívásakor.
- A függvény belsejében **ezzel** a változóval (objektummal) dolgozunk.
- A függvény végén ezek a változók (objektumok) automatikusan **felszabadulnak**.
- **Költségek**: létrehozás, másolás, felszabadítás

Példa

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

void fgvNovel( int a ) {
  a++;
}

int main( ) {
  int a = 5;
  fgvNovel( a );
  cout << a << endl;
}
```

Cím szerinti paraméterátadás

- A változó **nem** másolódik le
- A függvényen belül az **eredeti** változóval (objektummal) dolgozunk.
- A függvény végrehajtása után **nem** szabadulnak fel.
- Egy pointert állítunk a változóra és ezen keresztül érjük el azt.
- **Nincs** létrehozási, másolási, felszabadítási költség.

Példa

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

void fgvNovel( int* a ) {
  (*a)++;
}

int main( ) {
  int a = 5;
  fgvNovel( &a );
  cout << a << endl;
}
```

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

void fgvNovel( int a ) {
  a++;
}

int main( ) {
  int a = 5;
  fgvNovel( a );
  cout << a << endl;
}
```

Referencia szerinti paraméterátadás

- A pointerek nehézkes használata miatt vezették be
- Hasonló a pointerhez, de **itt nem kell képezni** a változó címét, közvetlenül a változóra mutat rá.
- Nem kell gondolkodni, hogy a pointer címét akarjuk, vagy a mutatott értéket, stb…
- **Megszorítások**:
  - a referenciát létrehozáskor inicializálni kell!
  - Nem lehet belőle tömböt képezni!
- **Referenciaképzés**: & jellel
  - int a = 5;
  - int& b = a;

Referencia szerinti paraméterátadás

- Nagyon hasonlít a pointer szerinti paraméter átadásra
- A C++ **nem tesz** különbséget referencia és érték paraméter között
  - nem tehetjük meg, hogy ugyanolyan névvel létrehozunk kettő függvényt, úgy, hogy az egyik paramétere egy referencia, a másiké egy érték.

Példa

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

void fgvNovel( int& a ) {
  a++;
}

int main( ) {
  int a = 5;
  fgvNovel( a );
  cout << a << endl;
}
```

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

void fgvNovel( int* a ) {
  (*a)++;
}

int main( ) {
  int a = 5;
  fgvNovel( &a );
  cout << a << endl;
}
```

Dinamikus tömbök

- Dinamikus tömböknek memóriát a new-val kell lefoglalni.

  int* arr = new int[size];

  SajatOsztaly* sajat = new SajatOsztaly[size];

  (ez csak akkor működik, ha van default konstruktor)

- Tömböket a delete[] operátorral törörüljük.

  delete[] arr;
  delete[] sajat;

Névterek - namespace

- Egy függvény vagy osztály írásakor könnyen előfordulhat, hogy az általunk választott **név ütközik** egy meglévővel.
- A névterek felhasználásával a különböző definíciók **névhierarchiába** szervezhetőek. Így **logikai csoportosítás is elérhető**.
- Névterekkel **tetszőleges mélységű névtér hierarchia** alakítható ki.
- Megkezdett névtér **folytatható**, akár külön másik forrás file-ban.
- Azonos névtérben **nem lehet** két egyforma definíció (osztály, függvény, változó).

Példa

```
namespace Math{
        class ComplexNum{ //… };
        class PositiveNum{ //… };
}
namespace House{
        class Kitchen{ //… };
        class Room{ //… };
}
```

Névterek elérése

- A névterek elérési útja programozható.
- Névtér feloldó operátor (**scope operátor**)
  ::
- Példa: std::cout
- A teljes elérési utat meg kell adni, ha az elérni kívánt tag több névtérbe van beágyazva:
- House::Room::Bed bed;

Névterek elérése

- Ahhoz, hogy ne kelljen minden alkalommal kiírni a teljes elérési útvonalat, a névteret használni kell.
- Ezután használhatjuk úgy az adott tagot, mintha a saját névterünkben lenne

```
using namespace Math;
ComplexNum a;

using namespace House::Room;
Bed bed;

using namespace std;
```

Öröklődés

Öröklődés

- Osztályok között értelmezett viszony
- Egy általánosabb típus felől (ős) haladunk egy speciálisabb típus felé (gyerek-, leszármazott-, utódosztály)
- Adatokat, műveleteket, viselkedésmódot örököl
  - kiegészíthetjük saját adatokkal, műveletekkel,
  - felülírhatunk bizonyos műveleteket
- A kód újrafelhasználásának egyik módja
- Egyszeres és többszörös öröklődés

Öröklődés

- Szintaxis:

  class ChildClass : visibility ParentClass{};

- Gyerekosztály konstruktorában meg kell hívni az ős osztály konstruktorát a következőképpen:

  public:

  ChildClass(int param) : Parent(param){}

Öröklődés

- Láthatóság használatával csak szigorítani lehet az öröklődést:

	public	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Többszörös öröklődés

- Nem egy, hanem több osztályból örököltetünk
- Szintaxis:

ChildClass: visibility Parent1, visibility Parent2, …

- Gyerek konstruktorában meg kell hívni az ősosztályok konstuktorát:

```
public:
  ChildClass( int param) : Parent1( param),
  Parent2( param), …
```

Ős adattagok elérése

- Az ősosztály minden adattagját elérjük (kivéve a private láthatóságúakat)
- Ha már felüldefiniáltunk egy metódust, és a metódust az osztályon belül bármilyen prefix nélkül használjuk, akkor a felülírt metódus fog meghívódni.
  - Viszont ha ilyen esetben az ős osztály metódusát szeretnénk használni, akkor meg kell adni az elérési útvonalat:
  - ParentClass::methodName(params);
  - miért?