Laborator 3:Constructori. Constructori de copiere. Destructori. Operatorii New si Delete. Mostenirea

Constructori

Constructorul unei clase este o functie speciala atribuita numai acelei clase care dupa cum sugereaza si numele, permite construirea unei instante a clasei respective. Constructorul unei clase are numele clasei respective si nu returneaza nimic in urma executiei de unde rezulta faptul ca in declaratia functiei nu trebuie specificat nici un tip returnat.

In interiorul constructorului sunt instantiate campurile unei clase. In lipsa definirii explicite a functiei constructor, se aloca un constructor implicit al clasei ce se doreste a fi definita, in cadrul careia se aloca memoria necesara variabilelor membru.

Functiile constructor sunt apelate in momentul in care obiectele unei clase “intra in existenta”, si anume, in momentul in care sunt declarate in interiorul programului.

Exemplu:

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

class MyClass
{
	int i;
public:
	MyClass()
	{
		i = 0;
		cout<<"S-a executat corpul constructorului."<<endl;
	}
	void set(int value)
	{
		i = value;
	}
	int get()
	{
		return i;
	}
};
int main()
{
	MyClass a; //se intra in apelul constructorului clasei MyClass
	cout<<a.get()<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
```

```
}
```

Ca orice alt tip de functie, constructorul poate primi parametri. Parametri pasati constructorului unei functii sunt folositi pentru a instantia variabilele membru ale unei clase cu o anumita valoare.

Exemplu:

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

class MyClass
{
	int i;
public:
	MyClass()
	{
		i = 0;
		cout<<"S-a executat corpul constructorului."<<endl;
	}
	MyClass(int value)
	{
		cout<<"S-a executat corpul constructorului cu parametru."<<endl;
		i = value;
	}
	void set(int value)
	{
		i = value;
	}
	int get()
	{
		return i;
	}
};
int main()
{
	MyClass a(50);//Se intra in apelul constructorului
	//cu parametru al clasei MyClass.
	//Acesta este modul de creare al obiectelor folosind constructorii cu
	//parametru.
	cout<<a.get()<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

In cazul in care constructorul este definit ca avand un singur parametru, se poate folosi urmatoarea sintaxa pentru a instantia obiecte din clasa respectiva folosind constructorul respectiv.

```
int main()
{
	MyClass a = 50;//Se intra in apelul constructorului
	//cu parametru al clasei MyClass.
	//Acesta este modul de creare al obiectelor folosind constructorii cu
	//parametru.
```

```
	cout<<a.get()<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

!! In situatia in care se doreste definirea explicita a constructorilor unei clase, trebuie tinut cont de faptul ca obiectele clasei respective pot fi instantiate folosind doar definitiile constructorilor din corpul clasei respective. Spre exemplu, daca in cazul clasei `MyClass`, definite mai sus, se defineste doar constructorul `MyClass(int value)`, atunci in `main` pot fi instantiate obiecte ale clasei folosind doar forma de declarare corespunzatoare constructorului respectiv.

Constructori de copiere

Un caz special de constructor este cel al **constructorilor de copiere**. Inainte de a da definitia acestora, ar trebui sa mentionam scopul pentru care sunt destinati acestia. Constructori de copiere sunt folositi in momentul in care se doreste instantierea unui obiect al unei clase pe baza unui alt obiect deja existent al clasei respective. Practic, in cadrul constructorlui de copiere, se executa o copie “bit cu bit” a obiectului deja existent.

Pentru a fi posibila definirea constructorului de copiere, trebuie pasat ca parametru o referinta constanta catre obiectul ce se doreste a fi copiat. Daca s-ar folosi pasarea parametrului prin valoare, ar fi necesara copierea obiectului pasat ca parametru la executarea constructorului. Acest lucru presupune apelul executiei constructorului de copiere, lucru care ar duce la o recursie infinita. Din acest motiv, obiectul ce se doreste a fi copiat se paseaza folosind pasarea parametrului prin referinta. Pentru a nu putea modifica obiectul respectiv in timpul executiei constructorului, este necesara pasarea valorii trimisa prin referinta ca o constanta.

Exemplu:

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

class MyClass
{
	int i;
public:
	MyClass()
	{
		i = 0;
		cout<<"S-a executat corpul constructorului."<<endl;
	}
	MyClass(int value)
	{
		cout<<"S-a executat corpul constructorului cu parametru."<<endl;
		i = value;
	}
```

```
	MyClass(const MyClass &obj)
	{
		cout<<"S-a executat corpul constructorului de copiere."<<endl;
		i = obj.i;
	}
	void set(int value)
	{
		i = value;
	}
	int get()
	{
		return i;
	}
};
int main()
{
	MyClass a = 50;
	MyClass b = a;
	cout<<a.get()<<endl;
	b.set(6);
	cout<<a.get()<<endl;
	cout<<b.get()<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Destructori

Din moment ce exista un tip special de functie ce permite “construirea” obiectelor unei clase, ar trebui sa existe un tip de functie si pentru “distrugerea” obiectelor unei clase. Acest lucru este posibil, si este realizabil prin intermediul functiilor de tip destructor. Aceste functii sunt similare in forma cu functiile constructor cu exceptia faptului ca numele functiei este precedat de caracterul ~. Destructorul unei clase este apelat automat in momentul in care functia in care a fost instantiat obiectul respectiv se termina.

Scopul principal al destructorului este de a elibera memoria folosita in cadrul obiectului sau de a inchide fisierele care au fost deschise in momentul creeri obiectului.

Exemplu:

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

class MyClass
{
	int i;
public:
	MyClass()
	{
		i = 0;
		cout<<"S-a executat corpul constructorului."<<endl;
	}
```

```
	MyClass(int value)
	{
		cout<<"S-a executat corpul constructorului cu parametru."<<endl;
		i = value;
	}

	MyClass(const MyClass &obj)
	{
		cout<<"S-a executat corpul constructorului de copiere."<<endl;
		i = obj.i;
	}

	~MyClass()
	{
		cout<<"S-a intrat in apelul destructorului clasei."<<endl;
	}
	void set(int value)
	{
		i = value;
	}
	int get()
	{
		return i;
	}
};

void func(MyClass x)
{
	cout<<"Esti in interioru functiei func."<<endl;
}

int main()
{
	MyClass a = 50;
	MyClass b = a;
	cout<<a.get()<<endl;
	b.set(6);
	cout<<b.get()<<endl;
	func(a);
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Operatorii new si delete

In limbajul C, alocarea dinamica a memorie se executa prin intermediul familiei de functii **malloc** si dezalocarea memoriei se realiza prin intermediul functiei **free**. In limbajul C++, alocarea dinamica a memoriei se realizeaza prin intermediul operatorilor **new** si **delete** care prezinta o sintaxa mult mai simplificata.

Principala diferenta dintre cele 2 abordari este ca in cazul operatorilor de C++ sunt folositi constructorii respectiv destructorii clasei respective pentru a realiza alocarea si dezalocarea memoriei. Operatorul

new intoarce un pointer catre zona de memorie alocata dinamic. In cazul in care nu exista suficienta memorie, se intoarce pointerul NULL. Operatorul **delete** elibereaza memoria catre care pointeaza operandul sau.

Exemplu:

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

class MyClass
{
	int i;
public:
	MyClass()
	{
		i = 0;
		cout<<"S-a executat corpul constructorului."<<endl;
	}
	MyClass(int value)
	{
		cout<<"S-a executat corpul constructorului cu parametru."<<endl;
		i = value;
	}

	MyClass(const MyClass &obj)
	{
		cout<<"S-a executat corpul constructorului de copiere."<<endl;
		i = obj.i;
	}

	~MyClass()
	{
		cout<<"S-a intrat in apelul destructorului clasei."<<endl;
	}
	void set(int value)
	{
		i = value;
	}
	int get()
	{
		return i;
	}
};
int main()
{
	MyClass *a;
	a = new MyClass;
	a->set(6);
	cout<<a->get()<<endl;
	delete a;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Mostenirea

Mostenirea este mecanismul tipic C++ ce permite reutilizarea codului. Mostenirea apare in urma creerii de noi clase prin operatia de **derivare**. **Derivarea** reprezinta definirea unei noi clase pe baza proprietatilor deja existente in clasa de baza. In C++ aveti posibilitatea sa realizati derivare simpla(pornind de la o singura clasa de baza) sau derivare multipla(pornind de la mai multe clase de baza).

In contextul operatiei de derivare, s-a introdus un nou specificator de acces pentru a putea largii principiul incapsularii datelor si in cadrul claselor derivate. Daca pana acum aveam de-a face doar cu specificatorii de acces **public** si **private** care sugereau drepturi de acces asupra membrilor unei clase, in contextul mostenirii, s-a introdus specificatorul de acces **protected**. Daca un membru al unei clase este specificat ca fiind **protected** atunci membrul respectiv poate fi folosit doar in cadrul metodelor si a functiilor friend ale clasei respective si in cadrul metodelor claselor derivate ce pornesc de la clasa respectiva.

Exemplu

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

class BaseClass
{
protected:
	int i;
public :
	BaseClass()
	{
		i = 0;
	}
};

class DerivedClass : public BaseClass
{
	int j;
public :
	DerivedClass()
	{
		j = 0;
	}
	int GetJ()
	{
		return j;
	}
	void SetJ(int k)
	{
		j = k;
	}
		int GetI()
	{
		return i;
	}
	void SetI(int value)
	{
		i = value;
```

```
	}
};
int main()
{
	BaseClass a;
	DerivedClass b;
	b.SetI(200);
	cout<<a.i<<endl; // eroare pentru ca membrul i este protected
	cout<<b.GetI()<<endl;
	cout<<b.i<<endl; // eroare pentru ca membrul i din clasa de baza este
protected si
	// nu poate fi accesat din afara clasei derivat sau a clasei de baza
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

In momentul in care se deriva o clasa de baza, se mentioneaza specificatorul de acces cu care sunt mosteniti membri clasei de baza in antetul operatiei de derivare.

```
class DerivedClass : <specificator de acces> BaseClass
{
…

};
```

Un caz aparte ce trebuie luat in considerare in cazul mostenirii este cel al instantierii in momentul creerii unui obiect derivat a membrilor clasei de baza cu anumite valori specificate de utilizator. Pentru a putea fi posibil acest lucru, trebuie apelat in constructorul clasei derivate, constructorul parametrizat al clasei de baza.

Exemplu

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

class BaseClass
{
protected:
	int i;
public :
	BaseClass()
	{
		i = 0;
	}
	BaseClass(int value)
	{
		i = value;
	}
};

class DerivedClass : public BaseClass
{
	int j;
public :
```

```
	DerivedClass(int value1, int value2) : BaseClass(value1)
	{
		j = value2;
	}
	int GetJ()
	{
		return j;
	}
	void SetJ(int k)
	{
		j = k;
	}
		int GetI()
	{
		return i;
	}
	void SetI(int value)
	{
		i = value;
	}
};
int main()
{
	BaseClass a;
	DerivedClass b(10, 20);
	cout<<b.GetI()<<" "<<b.GetJ()<<endl;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

In cazul mostenirii, ordinea in care sunt executati constructorii claselor este conform operatiunii de derivare, pornind de la clasele de baza spre clasele derivate. Astfel, sunt executati mai intai constructorii claselor de baza urmati de constructorii claselor derivate. In cazul destructorilor, ordinea de apelare este inversa, se apeleaza mai intai destructorul clasei derivate, urmat de destructorul claselor de baza.

Exemplu

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

class BaseClass
{
protected:
	int i;
public :
	BaseClass()
	{
		cout<<"S-a apelat constructorul clasei de baza."<<endl;
	}
	~BaseClass()
	{
		cout<<"S-a apelat destructorul clasei de baza."<<endl;
	}
};
```

```
class DerivedClass : public BaseClass
{
	int j;
public :
	DerivedClass()
	{
		cout<<"S-a apelat constructorul clasei derivate."<<endl;
	}
	~DerivedClass()
	{
		cout<<"S-a apelat destructorul clasei derivate."<<endl;
	}
};
int main()
{
	DerivedClass *a;
	a = new DerivedClass;
	delete a;
	system("PAUSE");
	return EXIT_SUCCESS;
}
```

Exercitii

1. Adaugati un constructor cu parametrii astfel incat sa fie posibila executia codului din `main`.

```
#include <iostream>
using namespace std;
class MyClass
{
    public:
    int i;
};
int main()
{
  MyClass a(3);
  cout<<a.i<<endl;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

2. Adaugati un constructor cu 2 parametrii in clasa `DerivedClass` pentru a fi posibila executia codului din `main`. Pentru initializarea campului *i* mostenit se va utiliza constructorul clasei de baza.

```
#include <iostream>
using namespace std;
class BaseClass
{
    protected:
    int i;
```

```
    public: BaseClass (int j)
    {i=j;}
            int GetI()
            {return i;}

};
class DerivedClass: public BaseClass
{int a;

};
int main()
{
  DerivedClass a(3,5);
  cout<<a.GetI()<<endl;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

3. Construiti urmatoarele clase:
    - Clasa `BaseClass` care are:
        1) **`a`**- membru privat de tip int;
        2) **`b`**-membru protected de tip int;
        3) **`c`**-membru public de tip int;
        4) constructor cu 3 parametrii pentru initializarea campurilor
    - Clasa `DerivedClass` care mostenste public `BaseClass` si are
        1) **`d`**-membru privat de tip int
        2) constructor cu 4 parametrii pentru initializarea campurilor si care foloseste constructorul clasei de baza pentru initializarea atributelor mostenite
        3) metoda publica **`Print()`** de afisare a tuturor atributelor (mostenite sau proprii)

Functia `main` va fi umatoarea:

```
int main()
{
  DerivedClass a(3,4,5,6);
  a.Print();
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

4. Pornind de la codul de mai jos, adaugati o noua clasa numita `A` care are:
    - un camp privat **`b`** de tipul pointer la `MyClass`
    - constructor cu parametru de tip int in care se face alocarea memoriei pentru **`b`** si initializarea acestuia cu valoarea primita ca parametru
    - destructor in care se realizeaza dezalocarea memoriei lui **`b`**
    - metoda publica `Print()` care afiseaza valoarea campului `i` (prin intermediul metodi `get()`) a lui **`b`**.

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

class MyClass
{     int  i;
      public:
      MyClass (int j) {i=j;}
      int get()
      {return i;
      }
};

int main()
{
  A a(3);
  a.Print();
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

5. Adaugati constructorul de copiere pentru clasa de mai jos astfel incat sa fie posibla executia codului din main.

```
#include <iostream>
using namespace std;
class array {
      int *p;
      int size;
      public:
      array(int sz)
      {
      p = new int[sz];
      size = sz;
      }
      ~array() { delete [] p; }

      void put(int i, int j)
      {
      if(i>=0 && i<size) p[i] = j;
      }
      int get(int i)
      {
      return p[i];
      }
};

int main()
{
array num(10);
int i;
for(i=0; i<10; i++) num.put(i, i);
cout << endl;
// create another array and initialize with num
array x(num); // invokes copy constructor
for(i=0; i<10; i++) cout << x.get(i);
cout<<endl;
```

```
system("PAUSE");
 return EXIT_SUCCESS;

}
```