Includerea claselor

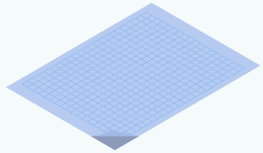
- Tot o relație de compunere, dar care presupune definirea unei clase (clasă copil) în interiorul altei clase (clasa părinte)
- Declarația este vizibilă doar în interiorul clasei părinte
- Accesul la clasa copil din exterior se va face prin utilizarea operatorului de rezoluție (::) aplicat pe clasa părinte (doar dacă clasa copil este declarată în zona publică a clasei părinte):

ClasaParinte::ClasaCopil obiectDeTipCopil;

Includerea claselor

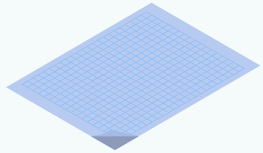
```
class TelefonMobil
{
     string producator;
     string model;
     class Baterie
     {
          //...
     } baterie;
     //...
};
```

Fișiere header



- Utilizate pentru a separa partea de definire a clasei de partea de implementare a ei și eventual de partea de utilizare
- În fișierele header nu sunt recomandate funcțiile inline
- Ajută la separarea diferitelor aspecte legate de clasă mai ales în proiectele mari pentru a face codul mai ușor de înțeles și depanat

Exemplul 1



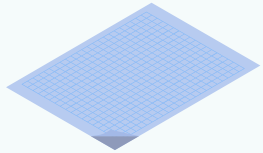
TelefonMobil.h

```
class TelefonMobil
{
public:
	float nivelBaterie;
	//...
	void incarca(float nivelIncarcare);
};
```

Source.cpp

```
#include "TelefonMobil.h"

void TelefonMobil::incarca(float nivelIncarcare)
{
	nivelBaterie += nivelIncarcare;
}

int main()
{
	TelefonMobil t1;
	t1.incarca(20);
}
```

Exemplul 2



TelefonMobil.h

```
class TelefonMobil
{
public:
	float nivelBaterie;
	//...
	void incarca(float);
};
```

TelefonMobil.cpp

```
#include "TelefonMobil.h"

void TelefonMobil::incarca(float nivel)
{
	nivelBaterie += nivel;
}
```

Source.cpp

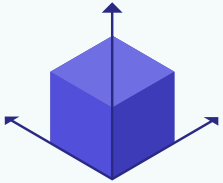
```
#include "TelefonMobil.h"

int main()
{
	TelefonMobil t1;
	t1.incarca();
}
```

Curs 06

SUPRAÎNCĂRCAREA

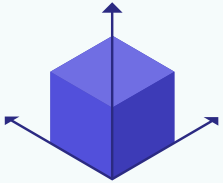
Supraîncărcarea funcțiilor și a operatorilor

Supraîncărcarea funcțiilor



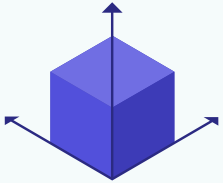
- Prima formă de polimorfism din Programarea Orientată Obiect
- Presupune existența a cel puțin două funcții (din același context) care au același nume, dar diferă prin numărul sau tipul parametrilor (prin signatură)
- Identificarea funcției corespunzătoare apelului se face la momentul compilării
- De aceea supraîncărcarea este considerată o formă de „legare timpurie” (early binding) sau un polimorfism slab

Exemplul 1



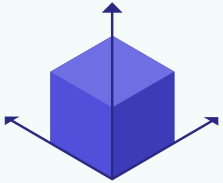
```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
     void incarca(int nivel);
     void incarca(int nivel, int nivelMaxim);
};
```

Exemplul 2



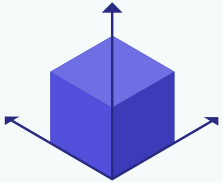
```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
     void incarca(int nivel);
     void incarca(string tipIncarcator);
};
```

Supraîncărcarea funcțiilor



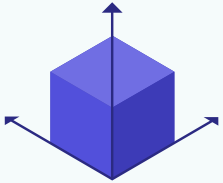
- Chiar dacă acest lucru nu a fost știut, am supraîncărcat până acum câteva metode
- Este vorba de constructorii din clase: funcții cu aceeași denumire (numele clasei, dar cu număr sau tip diferit de parametri)
- **Două supraîncărcări ale unei funcții nu pot diferi doar prin tipul returnat!**

Etape parcurse de compilator pentru alegerea corectă a funcției



1. Se caută o implementare a funcției fără a se realiza niciun fel de conversie
2. Se caută o implementare a funcției prin realizarea de conversii nedegradante (ex: de la int la long, de la char la int, etc.)
3. Se caută o implementare a funcției prin realizarea de conversii degradante (ex: de la long la int, de la float la int, etc.)
4. Se caută o implementare a funcției prin utilizarea operatorilor de cast supraîncărcați de către programator (conversii personalizate)

Etape parcurse de compilator pentru alegerea corectă a funcției



- Dacă în oricare dintre etapele precedente mai mult de o funcție corespunde apelului, atunci compilatorul va returna eroare de ambiguitate (nu știe ce variantă a funcției să aleagă pe baza apelului)
- Dacă după parcurgerea tuturor etapelor nicio funcție nu corespunde apelului, atunci se va returna eroare de link-editare (nu există o funcție definită care să corespundă apelului)

Supraîncărcarea operatorilor

- În limbajul C++ operatorii existenți (unari, binari, aritmetici, logici, relaționali) pot fi modificați astfel încât să funcționeze pentru tipurile de date definite de utilizator (ex: adunarea a două obiecte, compararea unui obiect cu un întreg, etc.)
- Modificarea lor se face prin funcții denumite operator<simbol_grafic> (unde <simbol_grafic> este simbolul grafic al acelui operator: + pentru adunare, ! pentru negație, = pentru atribuire, etc.)
- Lucrul acesta se numește supraîncărcarea operatorilor, deoarece, la fel ca la funcții, numele operatorului rămâne același și se schimbă doar tipul parametrului/parametrilor

Supraîncărcarea operatorilor

- În limbajul C++ există două modalități de supraîncărcare a operatorilor:
  - prin metodă (funcție membră), situație în care primul operand este obligatoriu de tipul clasei și va fi substituit de this (nu se primește drept parametru)
  - prin funcție globală, în special în cazurile când primul operand nu e de tipul clasei, situație când, pentru a accesa atributele private, fie vom anunța operatorul în clasa ca funcție friend, fie vom utiliza metode de acces

Metode și clase „prietene” (friend)

- În limbajul C++ se poate acorda acces anumitor funcții sau anumitor clase asupra tuturor membrilor clasei (fie ei privați sau protected)
- Aceste funcții sau clase se numesc prietene (friend) și trebuie anunțate în cadrul clasei în care se permite accesul în următorul mod:
  - pentru clase: **`friend class NumeClasa;`**
  - pentru funcții: **`friend tip_returnat NumeFunctie(tipParam1, tipParam2, etc);`**

Exemplul 1

```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
     void incarca(int nivel);
     void incarca(int nivel, int nivelMaxim);
     //clasa Persoana va avea acces asupra tuturor membrilor clasei TelefonMobil
     friend class Persoana;
};
```

Exemplul 2

```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
     void incarca(int nivel);
     void incarca(int nivel, int nivelMaxim);
     //functia doneazaTelefon va avea acces asupra tuturor membrilor clasei TelefonMobil
     friend void doneazaTelefon(TelefonMobil, Persoana);
};
```

Reguli de supraîncărcare a operatorilor

- Nu se pot supraîncărca decât operatori existenți
- Anumiți operatori sunt exceptați de la supraîncărcare (ex: `.` `*.` `::` `?:` `sizeof()`)
- Sensul unui operator nu se poate schimba prin supraîncărcare (+ va realiza adunare, - scădere, etc.)
- Comutativitatea, cardinalitatea și asociativitatea operatorilor se vor conserva prin supraîncărcare
- Operatorii supraîncărcați nu se compun automat (dacă supraîncărcăm operatorii + și =, nu înseamnă că am supraîncărcat și operatorul +=)

Excepții de la regulă

- Operatorii `()` `[]` `->` `=` se supraîncarcă doar prin funcție membră
- Operatorii **`new`** și **`delete`** se supraîncarcă doar prin funcție globală
- Comutativitatea nu este asigurată automat
- Formele pre și postfixate ale operatorilor de acest fel se supraîncarcă separat

Operatorii unari

- Pot fi supraîncărcați atât prin funcție membră (varianta recomandată), cât și prin funcție globală
- Dacă vor fi supraîncărcați prin funcție membră nu vor avea niciun parametru, iar ca funcție globală vor avea un parametru (un obiect - singurul operand)

Exemplu: supraîncărcarea operatorului ! prin funcție membră (metodă)

```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
public:
     bool operator!()
     {
          return nivelBaterie != 0;
     }
};
```

Exemplu: supraîncărcarea operatorului ! prin funcție globală

```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
     friend bool operator!(TelefonMobil);
};
bool operator!(TelefonMobil t)
{
     return t.nivelBaterie != 0;
}
```

Exemplu: utilizare operator!

```
int main()
{
	TelefonMobil telefonPersonal;
	//...
	bool poateFiFolosit = !telefonPersonal;
	//apel echivalent cu telefonPersonal.operator!() pentru functie membra
	//respectiv operator!(telefonPersonal) pentru functie globala
}
```

Operatorii unari de incrementare și decrementare

- Pot fi supraîncărcați atât prin funcție membră (varianta recomandată), cât și prin funcție globală
- Pentru a permite compilatorului să facă distincția între preincrementare (respectiv, predecrementare) și postincrementare (respectiv, postdecrementare), operatorii postfixați vor avea un parametru de tip int suplimentar (parametru ce nu va fi folosit efectiv)

Exemplu: supraîncărcarea operatorului de preincrementare prin funcție membră (metodă)

```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
public:
     TelefonMobil operator++()
     {
          nivelBaterie++;
          return *this;
     }
};
```

Exemplu: supraîncărcarea operatorului de preincrementare prin funcție globală

```
class TelefonMobil
{
     //...
     int nivelBaterie;
     //...
     friend TelefonMobil operator++(TelefonMobil&);
};
TelefonMobil operator++(TelefonMobil& t)
{
     t.nivelBaterie++;
     return t;
}
```

Exemplu: utilizare operator de preincrementare

```
int main()
{
     TelefonMobil telefonPersonal;
     //...
     TelefonMobil telefonDeServiciu = ++telefonPersonal;
     //apel echivalent cu telefonPersonal.operator++() pentru functie membra
     //respectiv operator++(telefonPersonal) pentru functie globala
}
```

Exemplu: supraîncărcarea operatorului de postincrementare prin funcție membră (metodă)

```
Class TelefonMobil
{
	//...
	int nivelBaterie;
	//...
public:
	TelefonMobil operator++(int i)
	{
		TelefonMobil copie = *this;
		nivelBaterie++;
		return copie;
	}
};
```

Exemplu: supraîncărcarea operatorului de postincrementare prin funcție globală

```
class TelefonMobil
{
	//...
	int nivelBaterie;
	//...
	friend TelefonMobil operator++(TelefonMobil&, int);
};
TelefonMobil operator++(TelefonMobil& t, int i)
{
	TelefonMobil copie = t;
	t.nivelBaterie++;
	return copie;
}
```

Exemplu: utilizare operator de postincrementare

```
int main()
{
     TelefonMobil telefonPersonal;
     //...
     TelefonMobil telefonDeServiciu = telefonPersonal++;
}
```

Operatorii binari

- Pot fi supraîncărcați atât prin funcție membră cât și prin funcție globală
- Varianta cu funcție globală este utilizată atunci când primul operand nu este de tipul clasei curente
- Dacă vor fi supraîncărcați prin funcție membră vor avea un parametru, iar ca funcție globală vor doi parametri (din care unul va fi de tipul clasei de interes)

Exemplu: supraîncărcarea operatorului + prin funcție membră (metodă)

```
Class TelefonMobil
{
	//...
	int nivelBaterie;
	//...
public:
	TelefonMobil operator+(TelefonMobil t)
	{
		TelefonMobil copie = *this;
		copie.nivelBaterie = nivelBaterie + t.nivelBaterie;
		return copie;
	}
};
```

Exemplu: supraîncărcarea operatorului + prin funcție globală

```
class TelefonMobil
{
	//...
	int nivelBaterie;
	//...
	friend TelefonMobil operator+(TelefonMobil, TelefonMobil);
};
TelefonMobil operator+(TelefonMobil t1, TelefonMobil t2)
{
	t1.nivelBaterie += t2.nivelBaterie;
	return t1;
}
```

Exemplu: utilizare operator +

```
int main()
{
	TelefonMobil telefonPersonal;
	TelefonMobil telefonDeServiciu;
	//...
	TelefonMobil telefon = telefonPersonal + telefonDeServiciu;
	//apel echivalent cu telefonPersonal.operator+(telefonDeServiciu) - functie membra
	//sau operator+(telefonPersonal, telefonDeServiciu) - functie globala
}
```

Exemplu: conservarea comutativității pentru operatorul +

```
Class TelefonMobil
{
	//...
	int nivelBaterie;
	//...
public:
	//momentan operatorul nu este comutativ
	TelefonMobil operator+(int x)
	{
		TelefonMobil copie = *this;
		copie.nivelBaterie = nivelBaterie + x;
		return copie;
	}
};
```

Exemplu: conservarea comutativității pentru operatorul +

```
class TelefonMobil
{
	//...
	int nivelBaterie;
	//... (inclusiv definitia de mai devreme)
	friend TelefonMobil operator+(int, TelefonMobil);
};
//acum operatorul+ este comutativ
TelefonMobil operator+(int x, TelefonMobil t2)
{
	t2.nivelBaterie += x;
	return t2;
}
```