Conversii de tip

Compilatorul de C++ respectă regurile de conversie definite de compilatorul de C:

- La evaluarea unei expresii se efectuează conversii sistematice de la tipurile întregi scurte la tipul int sau unsigned int. Apoi, pentru fiecare operand, dacă operanzii au tipuri diferite, se efectuează conversia necesară pentru a obține ambii operanzi de acelaşi tip.

- La atribuire se efectuează conversia valorii atribuite la tipul operandului care primeşte valoarea

- La transferul parametrilor se efectuează conversia valori fiecărui parametru efectiv la tipul parametrului formal

Conversile de tip se pot realiza prin:
- Supradefinirea operatorului de cast (nu poate realiza conversii dintr-un tip fundamental la un tip clasă)
- Prin intermediul constructorilor (nu permite conversia de la un tip fundamental la un tip clasă)

Supraîncărcarea operatorului de cast

Pentru exemplificare vom folosi clasa NumarRational. Vom supradefinii conversia la tipul float şi int.

```
class Rational
{
int x;
int y;         //atribute private ale clasei
public:
Rational(int=0, int=1); //constructor explicit cu parametri
impliciti
Rational(const Rational&);  //constructor de copiere
~Rational();               //destructor
void afisare();  //metoda pentru afisare
}; //terminarea clasei

Rational::Rational(int x, int y)
{
this->x = x;
```

```
this->y = y;
cout <<"Constructor "; afisare(); cout<<endl;
}
Rational::~Rational()
{
cout <<"Destructor "; afisare(); cout<<endl;
}
Rational::Rational(const Rational& z){
this->x = z.x;
this->y = z.y;
cout << "Constructor de copiere: "; afisare(); cout<<endl;
}
void Rational::afisare(){
cout << x << "/" << y << " ";
}
```

Supraincacarea operatorului float

Prototip:

```
operator float(); //conversie numarRational->float
```

Definire functie:

```
Rational::operator float()
{
cout << "Apelare float() "; afisare();
return x/(float)y;
}
```

Functii de test:

```
void functie(float r)
{ //test functie
cout << "Apel functie ( " << r << " )\n";
}

int main()
{
Rational r1(1,2), r2(3,4);
float f1, f2;
f1 = (float) r1;  //conversie explicita
cout << "f1=" << f1 << endl;
```

```
f2 = r2; //conversie implicita la atribuire
cout << "f2=" << f2 << endl;

functie(f1); //apel fara conversie

functie(r1); //conversie la transferul parametrilor

f1 = f1 + r1;  //conversie implicita r1->float
cout << "f1=" << f1 << endl;

f2 = r1 + r2;  //conversie implicita r1,r2->float
cout << "f2=" << f2 << endl;

f1 = r2 + 4.55;  //conversie implicita r2 -> float -> double
cout << "f1=" << f1 << endl;
     return 0;
}
```

Conversii folosind constructori

Functii de test

```
void functie1(Rational nr)
{
cout << "Apel functie ( "; nr.afisare(); cout << ")\n";
}

int main()
{
cout<<"---conversie folosind constructori----"<<endl;
Rational r3(1,2), r4(3,4);
float f3=10, f4=20;  //conversie explicita

r3 = Rational(f3);  //conversie implicita la atribuire
cout << "r3="; r3.afisare(); cout <<endl;

r4 = f4;
cout << "r4="; r4.afisare(); cout << endl;

functie1(f4);  //conversie implicita la transfer de parametri

return 0;
}
```

Supradefinirea operatorilor unari ++, --

In acest caz exista doua posibilități de suprascriere:

Obj++
TipClasa operator++(int x);

Sau

++Obj
TipClasa operator++();

Prototip:

```
Rational& operator++(); //postfix
Rational operator++(int); //prefix
```

Definire functii:

```
Rational& Rational::operator++()
{
cout<<"Apel operator ++ prefix ";
this->x++;
this->y++;
return *this;
}

Rational Rational::operator++(int a)
{
cout<<"Apel operator ++ sufix ";
this->x++;
this->y++;
return Rational(x,y);
}
```

Apel:

```
cout<<"Operator ++\n";
cout << "r1++ ="; r1++; r1.afisare(); cout << endl;
cout << "++r1 ="; ++r1; r1.afisare(); cout << endl;
```

Supradefinirea operatorilor prescurtati

Operatori prescurtați sunt +=, -= si restul de operatori care urmeaza acest Șablon (pattern).

Când se supraîncarcă unul din aceşti operatori se combina o operatie cu o atribuire.

Prototip

```
Rational& operator+=(Rational& b);
```

Definire functie:

```
Rational& Rational::operator+=(Rational& b)
{
cout<<"Apel operator += ";
this->x = b.x + x;
this->y = b.y + y;
return *this;
}
```

Apel:

```
cout << "r3 += r4 "; r3 += r4; r3.afisare(); cout << endl;
```

Supradefinirea operatorilor <<, >>

Operatori <<, >> sunt supradefiniti pentru a putea efectua operatii de I/O pentru clasele defite de utilizatori.

Conditiile pentru supradefinire sunt urmatoarele:
- Primul argument trebuie sa fie o referinta la un obiect: istream pentru operația de intrare >>, ostream << pentru operatorul de iesire <<.
- Nu pot fi funcții membre ale clasei pentru care supradefim, trebuie sa le declaram ca funcții prietene.
- Rezultatul întors trebuie sa fie o referință la adresa obiectului stream primit ca parametru.

Prototip:

```
friend istream& operator>>(istream&, Rational& nr);
friend ostream& operator<<(ostream&, Rational& nr);
```

Definire functii:

```
istream& operator>>(istream& intrare, Rational& nr)
{
float x, y;
char c;
intrare >> nr.x;
intrare.get(c);
intrare >> nr.y;
return intrare;
}

ostream& operator<<(ostream& iesire, Rational& nr)
{
iesire << nr.x;
if (nr.y > 0){
iesire << "/" << nr.y;
}
return iesire;
}
```

Apel:

```
Rational r7;
cout << "Introduceti numarul: "; cin >> r7;
cout << "Nr citit este: "; cout<< r7 << endl;
```

TEMA

1. Modificati problema 1 (clasa Rational) din laborator astfel incat sa supraincarcati toti operatorii necesari pentru afisarea rezultatelor urmatoare: 1/2+ 3/4 =5/4

2/5 – 3/4 = -7/20

3/4 * 16/15 = 4/5

2/5 / 7/4 = 8/35.

2. Realizati un program care deseneaza pe ecran dreptunghiuri, romburi. Pentru acesta definiți clasa Dreptunghi/Romb iar pentru desenare se supraîncarca operatorul <<

Suplimentar:

3. Creati clasa String pentru ca urmatorul program sa functioneze:

```
void f(String s) {
cout << s;
}
void g(String& s) {
cout << s;
}
int main(int, char*[]) {
String s1("This is a string");
String s2 = "This is another string";
String s3 = s2.concat(s1);
String s4, s5(32);
String s6=s2; // copy constructor
f(s2); // argument as value
g(s2);// arugument as reference
s4 = s2.substring(5,2); // substring starting at position 5
having length 2
} // destructors are called for all objects in reverse
order of declaration
```

Solutie posibila pentru problema 3:

```
#include <stdafx.h>
#include <iostream>
#include <string.h>
using namespace std;

class String
{
friend ostream& operator << (ostream&, const String&);
public:
// Default & user-defined constructor
String(int size = 15) {
s = new char [(sz=size)+1];
}
String(const char* str) {
init(str);
}
// Copy-constructor
String(const String& str) {
init(str.s);
}
// Destructor
~String() {
// free all resources acquired in constructors
if(s!=NULL)
delete [] s;
}
String operator=(const String& str);
String concat(const String& src);
String substring(int startPosition, int length);
private:
char *s;
int sz;
void init(const char* str);
};
void String::init(const char* str) {
s = new char [(sz=strlen(str))+1];
strcpy(s, str);
}
String String::operator=(const String& str) {
String::~String();
init(str.s);
return *this;
}
String String::concat(const String& src) {
char* old = s;
```

```
s = new char [sz+src.sz+1];
strcpy(s, old);
strcpy(s+sz-1, src.s);
sz += src.sz;
delete [] old;
return *this;
}
String String::substring(int startPosition, int length) {
String substr(length);
strncpy(substr.s, s+startPosition, length);
substr.s[length]=0;
return substr;
}
ostream& operator << (ostream& output, const String& s)
{
output << "[" <<&s.s<< "] " <<s.s<< endl;
return output;
}
void f(String s) {
cout << s;
}
void g(String& s) {
cout << s;
}
int main(int, char*[]) {
String s1("This is a string");
String s2 = "This is another string";
String s3 = s2.concat(s1);
String s4, s5(32);
String s6=s2; // copy constructor
s4 = s2.concat(s1);
f(s2); // argument as value
g(s2);// arugument as reference
// substring starting at position 5 having length 2
s4 = s2.substring(5,2);
return 0;
}
```