Type conversions

The C++ compiler respects the conversions rules defined by the C compiler:

Type conversion can be made by:
- Overloading the cast operator (cannot perform conversions from one fundamental type to a class type)
- By means of constructors (not allowed conversion from one fundamental type to a class type)

Overloading the cast operator

Example:

```
class Rational
{
int x;
int y;          // private atributes of the class
public:
Rational(int=0, int=1); //explicit constructor with default
//values
Rational(const Rational&);  //copy constructor
~Rational();                //destructor
void print();  //method for printing
}; //end of the class

Rational::Rational(int x, int y)
{
this->x = x;
this->y = y;
cout <<"Constructor "; print(); cout<<endl;
}
Rational::~Rational()
{
cout <<"Destructor "; print(); cout<<endl;
}
Rational::Rational(const Rational& z){
this->x = z.x;
this->y = z.y;
cout << "Copy constructor: "; print(); cout<<endl;
}
void Rational::print(){
cout << x << "/" << y << " ";
}
```

Overloading the operator float

Prototype:

```
operator float(); //conversion Rationalnumber->float
```

Define the function:

```
Rational::operator float()
{
cout << "The call of float() "; print();
return x/(float)y;
}
```

Test functions:

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

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

f2 = r2; //default conversion
cout << "f2=" << f2 << endl;

functie(f1); //call without conversion

functie(r1); //conversion

f1 = f1 + r1;  //default conversion r1->float
cout << "f1=" << f1 << endl;

f2 = r1 + r2;  //default conversion r1,r2->float
cout << "f2=" << f2 << endl;

f1 = r2 + 4.55;  //default conversion r2 -> float -> double
cout << "f1=" << f1 << endl;
```

```
     return 0;
}
```

Conversions when using constructors

Test functions

```
void function1(Rational nr)
{
cout << "Call function1 ( "; nr.print(); cout << ")\n";
}

int main()
{
cout<<"---conversion using constructors----"<<endl;
Rational r3(1,2), r4(3,4);
float f3=10, f4=20;  // explicit conversion

r3 = Rational(f3);  //default conversion
cout << "r3="; r3.print(); cout <<endl;

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

functie1(f4);  //default conversie

return 0;
}
```

Overloading unary operators ++, --

There are two possibilities of overloading:

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

or

++Obj
ClassType operator++();

Prototype:

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

Define the functions:

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

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

The call:

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

Overloading short operators

Short operators are +=, -= and similar operators.

When one of these operators are overloaded, there is a combination of an operation with an assignment.

Prototype

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

Define the function:

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

The call:

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

Overloading the operators <<, >>

We overload the operators <<, >> in order to perform sunt supradefiniti I/O For the classes defined by the user.

The conditions for overloading are:
- The first argument has to be a reference to an object: istream for input >>, ostream << for the output operator <<.
- These cannot be member functions of the class for which these are overloaded, we have to declare them as friend functions.
- The returned result has to be a reference to the address of the stream object received as parameter.

Prototype:

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

Define the functions:

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

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

The call:

```
Rational r7;
cout << "Introduce a rational number: "; cin >> r7;
cout << "The number is: "; cout<< r7 << endl;
```

HOMEWORK

1. Modify problem 1 (with the class Rational) such that you overload all the necessary operators for printing the following results:

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. Write a program which draws on the screen rectangles, diamonds. For this define the classes Rectangle/Diamond and for drawing overload the operator <<

Extra homework:

3. Create the class String such that the following program will work:

```
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
```

A possible solution for problem 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;
}
```