TP1 - Introduction to C++ language

Table of contents

Part 1 - class HelloW

class HelloW

To test the C++ features, I wrote a small `HelloW` class :
Header file :

```
class HelloW {
    string* toPrint;

    public:
        HelloW();
        HelloW(const HelloW& h);
        ~HelloW();
        HelloW& operator=(HelloW& h);
        friend ostream& operator<< (ostream& os, HelloW& h);
};
```

Implementation file :

```
HelloW::HelloW() : toPrint(new std::string()){
     *toPrint = "HelloW - hello world!";
};

HelloW::HelloW(const HelloW& h) : toPrint(new string()){
    *toPrint = *h.toPrint;
}

HelloW::~HelloW(){
    delete toPrint;
}

HelloW& HelloW::operator=(HelloW& h){
    delete toPrint;
    toPrint = h.toPrint;
    return *this;
}

ostream& operator<< (ostream& os, HelloW& h){
    os << *h.toPrint;
    return os;
}
```

Client file :

```
#include "HelloW.hpp"
#include <cstdlib>

int main(int argc, char* argv[]) {
    HelloW hello;
    cout << hello << endl;
    return EXIT_SUCCESS;
}
```

Output :

```
HelloW - hello world!
```

class HelloA & HelloB

These 2 classes are really similar to the `HelloW` class. The only difference is they have another method :

```
void HelloA::affiche(){
    cout << *toPrint << endl;
}
```

```
void HelloB::affiche(){
    cout << *toPrint << endl;
}
```

We also change the Client file :

```
#include "HelloW.hpp"
#include "HelloA.hpp"
#include "HelloB.hpp"
#include <cstdlib>

int main(int argc, char* argv[]) {
    HelloW hello;
    HelloA helloA;
    HelloB helloB;
    cout << hello << endl;
    helloA.affiche();
    helloB.affiche();

    return EXIT_SUCCESS;

}
```

Output :

```
HelloW - hello world!
HelloA - hello world!
HelloB - hello world!
```

Part 2 - class Vecteur

Members of the class

We define 2 private members of this class :

- Member dim (an integer, the dimension of the vector)
- Member tab (an array of float, the elements of the vector)

```
class Vecteur
{
        private:
                int dim;
                float* tab;
                [...]
};
```

Constructors

I wrote the 2 necessary constructors for the Coplien form :

```
Vecteur::Vecteur():dim(0),tab(NULL){};

Vecteur::Vecteur(const Vecteur& v){
        dim = v.dim;
        tab = new float[dim];
        for(int i = 0; i < dim; ++i)
                tab[i] = v.tab[i];
};
```

Then, I wrote the constructor with the parameter :

```
Vecteur::Vecteur(int d):dim(d),tab(new float[dim]){};
```

Input/Output operators

For the moment, it is impossible to see nothing at all in the vectors because we did not set up the input/output operators. So, I wrote the operators :

```
ostream& operator<<(std::ostream& os, Vecteur& v) {
        os << "(";
        for(int i = 0; i < v.dim-1;++i)
                os << v.tab[i] << ", ";
        os << v.tab[v.dim-1] << ")";
        return os;
};

istream& operator>>(std::istream& is,Vecteur& v){
        cout << "Taille: ";
        is >> v.dim;
        v.tab = new float[v.dim];
        for(int i = 0; i < v.dim; ++i)
                is >> v.tab[i];
        return is;
};
```

Now we can see, the output of our empty vectors :
Output :

```
Erreur de segmentation (core dumped)
```

We can see the content of an empty vector, so we need to implement, first, the `[]` operator.

Acces to elements

```
float& Vecteur::operator[](int index){
        return this->tab[index];
};
```

Now, we can set up elements of tab. I made some tests in the main loop :

```
Vecteur v1(3);
Vecteur v2 = v1;

v1[0] = 2;
v1[1] = 4;
v1[2] = 6;

cout << "v1 : " << v1 << endl;
```

Output :

```
v1 :    (2, 4, 6)
```

Of cours, we cannot visualize v2 because it is an empty vector.

Destructor

To further complete the Coplien form, we must define the default destructor.

```
Vecteur::~Vecteur(){
        delete [] tab;
};
```

It will be called at the end of the main loop.

Affectation operator

```
Vecteur& Vecteur::operator=(const Vecteur& v){
        if(this != &v) {
                dim = v.dim;
                delete [] tab;
                tab = new float[dim];
                for(int i = 0;i<dim;++i)
                        tab[i] = v.tab[i];
        }
        return *this;
};

Vecteur& Vecteur::operator=(float i){
        for(int j = 0; j<dim;++j)
                tab[j] = i;
        return *this;
};
```

To finish the Coplien, we must define an affectation operator. The first affectation matches to the affectation with a vector element and the second matches to an affectation with a scalar.
Main loop :

```
Vecteur v1(3);
Vecteur v4(5),v5(3);

v1[0] = 2;
v1[1] = 4;
```

```
v1[2] = 6;

cout << "v1 : " << v1 << endl;

v4 = 4;
v5 = v1;

cout << "v4 : " << v4 << endl;
cout << "v5 : " << v5<< endl;
```

Output :

```
v1 :    (2, 4, 6)
v4 :   (4, 4, 4, 4, 4)
v5 :   (2, 4, 6)
```

Incrementation operators

```
Vecteur& Vecteur::operator++(void){
    for(int i = 0; i < this->dim; ++i){
        this->tab[i]++;
    }
    return *this;
}

Vecteur Vecteur::operator++(int p){
    Vecteur v = *this;
    +++this;
    return v;
}

Vecteur& Vecteur::operator--(void){
    for(int i = 0; i < this->dim; ++i){
        this->tab[i]--;
    }
    return *this;
}

Vecteur Vecteur::operator--(int p){
    Vecteur v = *this;
    --*this;
    return v;
}
```

We define 2 operators for incrementation because one matches with the `++v` operator, which increment and then return the incremented vector, and the second matches the `v++` operator, which return a copy of the vector before it was incremented.

Main loop :

```
v2 = v1;
cout << "v2 = v1 :       " << v2 << endl;

v2++;
cout << "v2++ : " << v2 << endl;
v2--;
cout << "v2-- : " << v2 << endl;
v3 = --v2;
cout << "v3   : " << v3 << " v2   : " << v2 << endl;
```

Output :

```
v2 = v1 :       (2, 4, 6)
v2++ :  (3, 5, 7)
v2-- :  (2, 4, 6)
v3   :  (1, 3, 5) v2   : (1, 3, 5)
```

Boolean operations

It is difficult to compare vectors. For this part, I decided that the vectors will be compare on their length.
Firstly, I wrote a method that compute the squared norm of the vector :

```
float Vecteur::longueurCarre(void){
    float squared_size = 0;
    for(int i = 0; i < dim; ++i)
        squared_size += tab[i]*tab[i];
    return squared_size;
}
```

And then, I just return the comparson between the two norms :

```
bool Vecteur::operator==(Vecteur& v){
    return longueurCarre() == v.longueurCarre();
}

bool Vecteur::operator<(Vecteur& v){
    return longueurCarre() < v.longueurCarre();
}

bool Vecteur::operator>(Vecteur& v){
    return longueurCarre() > v.longueurCarre();
}
```

```cpp
bool Vecteur::operator<=(Vecteur& v){
    return longueurCarre() <= v.longueurCarre();
}

bool Vecteur::operator>=(Vecteur& v){
    return longueurCarre() >= v.longueurCarre();
}
```

Main loop :

```cpp
v3[0] = 6;
v3[1] = 4;
v3[2] = 2;
cout << "v1 :   " << v1 << endl;
cout << "v2 :   " << v2 << endl;
cout << "v3 :   " << v3 << endl;

cout << "v1 == v3 :     " << (v1 == v3) << endl;
cout << "v1 > v2 :      " << (v1 > v2) << endl;
cout << "v1 < v2 :      " << (v1 < v2) << endl;
cout << "v1 <= v1 :     " << (v1 <= v1) << endl;
cout << "v3 >= v1 :     " << (v3 >= v1) << endl;
```

Output :

```
v1 :    (2, 4, 6)
v2 :    (1, 3, 5)
v3 :    (6, 4, 2)
v1 == v3 :      1
v1 > v2 :       1
v1 < v2 :       0
v1 <= v1 :      1
v3 >= v1 :      1
```

Binary operators

I added a simple addition and soustraction between 2 vectors :

```cpp
Vecteur Vecteur::operator+(const Vecteur& v){
    Vecteur v_f = *this;
    for(int i = 0; i < v.dim; ++i){
        v_f.tab[i] += v.tab[i];
    }
    return v_f;
};
```

```
Vecteur Vecteur::operator-(const Vecteur& v){
    Vecteur v_f = *this;
    for(int i = 0; i < v.dim; ++i){
        v_f.tab[i] -= v.tab[i];
    }
    return v_f;
}
```

Main loop :

```
cout << "v1 :   " << v1 << endl;
cout << "v2 :   " << v2 << endl;
cout << "v3 :   " << v3 << endl;
v3 = v1 + v2;
cout << "v3 = v1 + v2 : " << v3 << endl;
v1 = v3 - v2 + v1;
cout << "v1 = v3 - v2 + v1 : " << v1 << endl;
```

Output :

```
v1 :    (2, 4, 6)
v2 :    (1, 3, 5)
v3 :    (6, 4, 2)
v3 = v1 + v2 :  (3, 7, 11)
v1 = v3 - v2 + v1 : (4, 8, 12)
```

Scalar product

Definition of the a classic scalar product :

```
float Vecteur::operator*(const Vecteur& v){
    float prod = 0;
    for(int i = 0; i < this->dim; ++i){
        prod += this->tab[i] * v.tab[i];
    }
    return prod;
}
```

Main loop :

```
cout << "v1 :   " << v1 << endl;
cout << "v2 :   " << v2 << endl;
cout << "v3 :   " << v3 << endl;
float scalar_product = v1 * v2;
```

```
cout << "Scalar product :       " << scalar_product << endl;
```

Output :

```
v1 :    (2, 4, 6)
v2 :    (1, 3, 5)
v3 :    (6, 4, 2)
Scalar product :        44
```

Product by a scalar

We define this product with 2 method : the first is the operation `v * f` and the second is `f * v`. This second method is a friend function.

```
Vecteur Vecteur::operator*(float f){
    Vecteur v_f = *this;
    for(int i = 0; i < this->dim; ++i){
        v_f.tab[i] *= f;
    }
    return v_f;
}

Vecteur operator*(float f,Vecteur& v){
    return v * f;
}
```

Main loop :

```
cout << "v3 :   " << v3 << endl;
cout << "v5 :   " << v5 << endl;

v5 = v3 * 2;
cout << "v3 :   " << v3 << endl;
cout << "v5 :   " << v5 << endl;

v3 = 2 * v3;
cout << "v3 :   " << v3 << endl;
cout << "v5 :   " << v5 << endl;
```

Output :

```
v3 :    (6, 4, 2)
v5 :    (0, 0, 0)
```

```
v3 :    (6, 4, 2)
v5 :    (12, 8, 4)

v3 :    (12, 8, 4)
v5 :    (12, 8, 4)
```

Auto-addition & auto-soustraction operators

Implementation :

```
Vecteur& Vecteur::operator+=(const Vecteur& v){
    for(int i = 0; i < v.dim; ++i){
        this->tab[i] += v.tab[i];
    }
    return *this;
}

Vecteur& Vecteur::operator-=(const Vecteur& v){
    for(int i = 0; i < v.dim; ++i){
        this->tab[i] -= v.tab[i];
    }
    return *this;
}
```

Main loop :

```
cout << "v3 :   " << v3 << endl;
cout << "v5 :   " << v5 << endl;

v5 += v3;
cout << "v5 :   " << v5 << endl;
v5 += v3;
cout << "v5 :   " << v5 << endl;
v3 -= v5;
cout << "v3 :   " << v3 << endl;
```

Output :

```
v3 :    (6, 4, 2)
v5 :    (0, 0, 0)

v5 :    (6, 4, 2)

v5 :    (12, 8, 4)
```

```
v3 :    (-6, -4, -2)
```

Call of constructors

We can determine the number of number of times a constructor is called by defining a static counter (an integer) which will be incremented each time a Vecteur object is created (the static members are common to every object of the class) :

Header file :

```
class Vecteur {
        private:
                [...]
        public:
                static int appels_constructeur_par_defaut;
                static int appels_constructeur_par_copie;
                static int appels_constructeur_avec_dim;
                [...]
```

Initialization in .cc file :

```
int Vecteur::appels_constructeur_par_defaut = 0;
int Vecteur::appels_constructeur_par_copie = 0;
int Vecteur::appels_constructeur_avec_dim = 0;
```

New constructors :

```
Vecteur::Vecteur():dim(0),tab(NULL){
    appels_constructeur_par_defaut++;
};

Vecteur::Vecteur(const Vecteur& v){
    dim = v.dim;
    tab = new float[dim];
    for(int i = 0; i < dim; ++i){
        tab[i] = v.tab[i];
    }
    appels_constructeur_par_copie++;
};

Vecteur::Vecteur(int d):dim(d),tab(new float[dim]){
    appels_constructeur_avec_dim++;
};
```

Main loop :

```
Vecteur v1(3);
Vecteur v2 = v1;
Vecteur v3;
Vecteur v4(5),v5(3);
v1 = v2;

cout << "Appels constructeur par défaut : " <<
Vecteur::appels_constructeur_par_defaut << endl;
cout << "Appels constructeur par copie : " <<
Vecteur::appels_constructeur_par_copie << endl;
cout << "Appels constructeur avec dimension : " <<
Vecteur::appels_constructeur_avec_dim << endl;

v1[0] = 2;
v1[1] = 4;
v1[2] = 6;

v2++;
cout << "v2++ : " << v2 << endl;
v2--;
cout << "v2-- : " << v2 << endl;

cout << "Appels constructeur par défaut : " <<
Vecteur::appels_constructeur_par_defaut << endl;
cout << "Appels constructeur par copie : " <<
Vecteur::appels_constructeur_par_copie << endl;
cout << "Appels constructeur avec dimension : " <<
Vecteur::appels_constructeur_avec_dim << endl;
```

Output :

```
Appels constructeur par défaut : 1
Appels constructeur par copie : 1
Appels constructeur avec dimension : 3
v2++ :  (2, 4, 6)
v2-- :  (1, 3, 5)
Appels constructeur par défaut : 1
Appels constructeur par copie : 3
Appels constructeur avec dimension : 3
```

The interesting thing to note is that when we increment or decrement a vector, the constructor by copy is called. Actually, we call this constructor by returning `*this` in these functions. So we would have the same result with every other methods that return `*this`.

Annexes

Hello World

Makefile

```
# differents types de fichiers
.SUFFIXES:.o.cpp.ln


#---------------------------------
# INITIALISATION DES VARIABLES
#---------------------------------

# Indiquer le compilateur
CCC = g++

# Les chemins ou se trouvent les fichiers a inclure
INCLUDES= -I/usr/openwin/include

# Options de compilation.
CCFLAGS=  ${INCLUDES} -c

# Options pour le linker.
LFLAGS=  -o

# Les librairies avec lesquelle on va effectueller l'edition de liens
LIBS=

# Les fichiers sources de l'application
FILES= HelloW.cpp ClientHW.cpp HelloA.cpp HelloB.cpp

#-----------
# LES CIBLES
#-----------

tp:  $(FILES:.cpp=.o)
        $(CCC) -o helloW $(FILES:.cpp=.o) ${LIBS}

clean:
        /bin/rm $(FILES:.cpp=.o) helloW

#-------------------------------------------------------------------------
----
# LES REGLES DE DEPENDANCE. Certaines sont implicites mais je recommande
d'en
# mettre une par fichier source.
#-------------------------------------------------------------------------
----

ClientHW.o:ClientHW.cpp HelloW.cpp HelloA.cpp HelloB.cpp HelloW.hpp
HelloA.hpp HelloB.hpp
```

```
%.o:%.cpp %.hpp

#---------------------------------
# REGLES DE COMPILATION IMPLICITES
#---------------------------------

.cpp.o:; ${CCC} ${CCFLAGS} $*.cpp
```

HelloW class

Header file

```
#include <iostream>
#include <string>

using namespace std;

class HelloW {
    string* toPrint;

    public:
        HelloW();
        HelloW(const HelloW& h);
        ~HelloW();
        HelloW& operator=(HelloW& h);
        friend ostream& operator<< (ostream& os, HelloW& h);
};
```

Implementation file

```
#include "HelloW.hpp"

HelloW::HelloW() : toPrint(new std::string()){
     *toPrint = "HelloW - hello world!";
};

HelloW::HelloW(const HelloW& h) : toPrint(new string()){
    *toPrint = *h.toPrint;
}

HelloW::~HelloW(){
    delete toPrint;
}

HelloW& HelloW::operator=(HelloW& h){
    delete toPrint;
    toPrint = h.toPrint;
    return *this;
```

```
}

ostream& operator<< (ostream& os, HelloW& h){
    os << *h.toPrint;
    return os;
}
```

HelloA class

Header file

```
#include <iostream>
#include <string>

using namespace std;

class HelloA {
    string* toPrint;

    public:
        HelloA();
        HelloA(const HelloA& h);
        ~HelloA();
        HelloA& operator=(HelloA& h);
        void affiche();
};
```

Implementation file

```
#include "HelloA.hpp"
#include <ostream>
#include <string>

HelloA::HelloA() : toPrint(new std::string()){
     *toPrint = "HelloA - hello world!";
};

HelloA::HelloA(const HelloA& h) : toPrint(new string()){
    *toPrint = *h.toPrint;
}

HelloA::~HelloA(){
    delete toPrint;
}

HelloA& HelloA::operator=(HelloA& h){
    delete toPrint;
    toPrint = h.toPrint;
```

```
    return *this;
}

void HelloA::affiche(){
    cout << *toPrint << endl;
}
```

HelloB class

Header file

```
#include <iostream>
#include <string>

using namespace std;

class HelloB {
    string* toPrint;

    public:
        HelloB();
        HelloB(const HelloB& h);
        ~HelloB();
        HelloB& operator=(HelloB& h);
        void affiche();
};
```

Implementation file

```
#include "HelloB.hpp"
#include <ostream>
#include <string>

HelloB::HelloB() : toPrint(new std::string()){
     *toPrint = "HelloB - hello world!";
};

HelloB::HelloB(const HelloB& h) : toPrint(new string()){
    *toPrint = *h.toPrint;
}

HelloB::~HelloB(){
    delete toPrint;
}

HelloB& HelloB::operator=(HelloB& h){
    delete toPrint;
    toPrint = h.toPrint;
```

```
    return *this;
}

void HelloB::affiche(){
    cout << *toPrint << endl;
}
```

ClientHW file

```
#include "HelloW.hpp"
#include "HelloA.hpp"
#include "HelloB.hpp"
#include <cstdlib>

int main(int argc, char* argv[]) {
    HelloW hello;
    HelloA helloA;
    HelloB helloB;
    cout << hello << endl;
    helloA.affiche();
    helloB.affiche();

    return EXIT_SUCCESS;

}
```

Vecteur class

Makefile

```
# differents types de fichiers
.SUFFIXES:.o.cc.ln


#---------------------------------
# INITIALISATION DES VARIABLES
#---------------------------------

# Indiquer le compilateur
CCC = g++

# Les chemins ou se trouvent les fichiers a inclure
INCLUDES= -I/usr/openwin/include

# Options de compilation.
CCFLAGS=  ${INCLUDES} -c
```

```
# Options pour le linker.
LFLAGS=  -o

# Les librairies avec lesquelle on va effectueller l'edition de liens
LIBS=

# Les fichiers sources de l'application
FILES= Client.cc Vecteur.cc

#-----------
# LES CIBLES
#-----------

tp:  $(FILES:.cc=.o)
        $(CCC) -o tp $(FILES:.cc=.o) ${LIBS}

clean:
        /bin/rm $(FILES:.cc=.o) tp

#-------------------------------------------------------------------------
----
# LES REGLES DE DEPENDANCE. Certaines sont implicites mais je recommande
d'en
# mettre une par fichier source.
#-------------------------------------------------------------------------
----

Client.o:Client.cc Vecteur.cc Vecteur.h
Vecteur.o:Vecteur.cc Vecteur.h

#---------------------------------
# REGLES DE COMPILATION IMPLICITES
#---------------------------------

.cc.o:; ${CCC} ${CCFLAGS} $*.cc
```

Header file

```
#ifndef VECTEUR_H
#define VECTEUR_H
//
// Type vecteur
//
#include <iostream>

using namespace std;

//
// Type vecteur de float
//
```

```
class Vecteur
{
private:
    int dim;
    float* tab;
public:
    static int appels_constructeur_par_defaut;
    static int appels_constructeur_par_copie;
    static int appels_constructeur_avec_dim;

// Constructeurs par defaut(void) , avec la taille du vect.(int), par
copie(vecteur&)
Vecteur(); // Constructeur par défaut
Vecteur(const Vecteur& v); // Constructeur par copie
Vecteur(int d); // Constructeur avec taille

// Destructeur
~Vecteur();

// Acces a la taille du vecteur
int dimension(void);

// Access à la longueur au carré du vecteur
float longueurCarre(void);

// acces au element []
float& operator[](int index);

// affectation : =(vecteur), =(int)
Vecteur& operator=(const Vecteur& v);
Vecteur& operator=(float i);

// incrementation/decr. : ++(void), ++(int), --(void)
Vecteur& operator++(void);
Vecteur operator++(int p);

Vecteur& operator--(void);
Vecteur operator--(int p);


// op. booleens : ==(vecteur&), <, >, <=, >=
bool operator==(Vecteur& v);
bool operator<(Vecteur& v);
bool operator>(Vecteur& v);
bool operator<=(Vecteur& v);
bool operator>=(Vecteur& v);

// op. binaires : +(vecteur&), -
Vecteur operator+(const Vecteur& v);
Vecteur operator-(const Vecteur& v);

// produit scalaire: *(vecteur&)
```

```
    float operator*(const Vecteur& v);

    // produit par un scalaire: v * n, n * v
    Vecteur operator*(float f);
    friend Vecteur operator*(float f,Vecteur& v);

    // auto-adition : +=(vecteur&), -=
    Vecteur& operator+=(const Vecteur& v);
    Vecteur& operator-=(const Vecteur& v);

    // Operateurs d'entree/sortie: <<, >>
    friend ostream& operator<< (ostream&, Vecteur&);
    friend istream& operator>> (istream&, Vecteur&);

};
#endif // VECTEUR_H
```

Implementation file

```
// Type vecteur
#include "Vecteur.h"
#include <sys/types.h>

// Initialisation du membre static
int Vecteur::appels_constructeur_par_defaut = 0;
int Vecteur::appels_constructeur_par_copie = 0;
int Vecteur::appels_constructeur_avec_dim = 0;

// Constructeurs
Vecteur::Vecteur():dim(0),tab(NULL){
    appels_constructeur_par_defaut++;
};

Vecteur::Vecteur(const Vecteur& v){
    dim = v.dim;
    tab = new float[dim];
    for(int i = 0; i < dim; ++i){
        tab[i] = v.tab[i];
    }
    appels_constructeur_par_copie++;
};

Vecteur::Vecteur(int d):dim(d),tab(new float[dim]){
    appels_constructeur_avec_dim++;
};

// Destructeur
Vecteur::~Vecteur(){
    delete [] tab;
};
```

```cpp
// Nombre d'elements
int Vecteur::dimension(void){
        return dim;
};

// Longueur au carré
float Vecteur::longueurCarre(void){
    float squared_size = 0;
    for(int i = 0; i < dim; ++i)
        squared_size += tab[i]*tab[i];
    return squared_size;
}

// Acces aux elements
float& Vecteur::operator[](int index){
    return this->tab[index];
}

// affectation : =(vecteur), =(int)
Vecteur& Vecteur::operator=(const Vecteur& v){
    if(this != &v) {
        dim = v.dim;
        delete [] tab;
        tab = new float[dim];
        for(int i = 0;i<dim;++i) {
            tab[i] = v.tab[i];
        }
    }
    return *this;
};

Vecteur& Vecteur::operator=(float i){
        for(int j = 0; j<dim;++j)
                tab[j] = i;
        return *this;
}

// incrementation/decr. : ++, ++(int), --
Vecteur& Vecteur::operator++(void){
    for(int i = 0; i < this->dim; ++i){
        this->tab[i]++;
    }
    return *this;
}

Vecteur Vecteur::operator++(int p){
    Vecteur v = *this;
    +++this;
    return v;
}
```

```
Vecteur& Vecteur::operator--(void){
    for(int i = 0; i < this->dim; ++i){
        this->tab[i]--;
    }
    return *this;
}

Vecteur Vecteur::operator--(int p){
    Vecteur v = *this;
    --*this;
    return v;
}

// op. booleens : ==, <, >, <=, >=
bool Vecteur::operator==(Vecteur& v){
    return longueurCarre() == v.longueurCarre();
}

bool Vecteur::operator<(Vecteur& v){
    return longueurCarre() < v.longueurCarre();
}

bool Vecteur::operator>(Vecteur& v){
    return longueurCarre() > v.longueurCarre();
}

bool Vecteur::operator<=(Vecteur& v){
    return longueurCarre() <= v.longueurCarre();
}

bool Vecteur::operator>=(Vecteur& v){
    return longueurCarre() >= v.longueurCarre();
}

// op. binaires : +, -
Vecteur Vecteur::operator+(const Vecteur& v){
    Vecteur v_f = *this;
    for(int i = 0; i < v.dim; ++i){
        v_f.tab[i] += v.tab[i];
    }
    return v_f;
};

Vecteur Vecteur::operator-(const Vecteur& v){
    Vecteur v_f = *this;
    for(int i = 0; i < v.dim; ++i){
        v_f.tab[i] -= v.tab[i];
    }
    return v_f;
}

// produit scalaire : *
```

```
float Vecteur::operator*(const Vecteur& v){
    float prod = 0;
    for(int i = 0; i < this->dim; ++i){
        prod += this->tab[i] * v.tab[i];
    }
    return prod;
}

// produit par un scalaire: n * v, v * n
Vecteur Vecteur::operator*(float f){
    Vecteur v_f = *this;
    for(int i = 0; i < this->dim; ++i){
        v_f.tab[i] *= f;
    }
    return v_f;
}

Vecteur operator*(float f,Vecteur& v){
    return v * f;
}

// auto-adition : +=, -=
Vecteur& Vecteur::operator+=(const Vecteur& v){
    for(int i = 0; i < v.dim; ++i){
        this->tab[i] += v.tab[i];
    }
    return *this;
}

Vecteur& Vecteur::operator-=(const Vecteur& v){
    for(int i = 0; i < v.dim; ++i){
        this->tab[i] -= v.tab[i];
    }
    return *this;
}

// Operateurs d'entree/sortie: <<, >>
ostream& operator<<(ostream& os, Vecteur& v) {
    os << "(";
    for(int i = 0; i < v.dim-1;++i) {
        os << v.tab[i] << ", ";
    }
    os << v.tab[v.dim-1] << ")";
    return os;
}

istream& operator>>(istream& is,Vecteur& v){
    cout << "Taille: ";
    is >> v.dim;
    v.tab = new float[v.dim];
    for(int i = 0; i < v.dim; ++i){
        is >>  v.tab[i];
```

```
    }
    return is;
}
```

Client file

```
// Include de l'application
#include "Vecteur.h"

int
main(int argc, char **argv)
{

        cout << "PROGRAMME DE TEST" << endl;

        // Constructeurs
        Vecteur v1(3);
        Vecteur v2 = v1;
        Vecteur v3;
        Vecteur v4(5),v5(3);
        v1 = v2;

        cout << "Appels constructeur par défaut : " <<
Vecteur::appels_constructeur_par_defaut << endl;
        cout << "Appels constructeur par copie : " <<
Vecteur::appels_constructeur_par_copie << endl;
        cout << "Appels constructeur avec dimension : " <<
Vecteur::appels_constructeur_avec_dim << endl;

        v1[0] = 2;
        v1[1] = 4;
        v1[2] = 6;
        cout << "v1 :   " << v1 << endl;

        // v4 = 4;
        // v5 = v1;
        // cout << "v4 :   " << v4 << endl;
        // cout << "v5 :   " << v5<< endl;

        cout << "v1 :   " << v1 << endl;


        v2 = v1;
        cout << "v2 = v1 :      " << v2 << endl;

        v2++;
        cout << "v2++ : " << v2 << endl;
        v2--;
        cout << "v2-- : " << v2 << endl;
        v3 = --v2;
```

```
        cout << "v3   : " << v3 << " v2   : " << v2 << endl;

        // Comparaisons
        v3[0] = 6;
        v3[1] = 4;
        v3[2] = 2;

        //cout << "v1 : " << v1 << endl;
        //cout << "v2 : " << v2 << endl;
        cout << "v3 :   " << v3 << endl;
        cout << "v5 :   " << v5 << endl;

        /*
        cout << "v1 == v3 :     " << (v1 == v3) << endl;
        cout << "v1 > v2 :      " << (v1 > v2) << endl;
        cout << "v1 < v2 :      " << (v1 < v2) << endl;
        cout << "v1 <= v1 :     " << (v1 <= v1) << endl;
        cout << "v3 >= v1 :     " << (v3 >= v1) << endl;
        */


        v3 = v1 + v2;
        cout << "v3 = v1 + v2 : " << v3 << endl;
        //v1 = v3 - v2 + v1;
        //cout << "v1 = v3 - v2 + v1 : " << v1 << endl;


        /*
        float scalar_product = v1 * v2;
        cout << "Scalar product :       " << scalar_product << endl;
        */

        /*
        v5 = v3 * 2;
        cout << "v3 :   " << v3 << endl;
        cout << "v5 :   " << v5 << endl;
        v3 = 2* v3;
        cout << "v3 :   " << v3 << endl;
        cout << "v5 :   " << v5 << endl;
        */

        /*
        v5 += v3;
        cout << "v5 :   " << v5 << endl;
        v5 += v3;
        cout << "v5 :   " << v5 << endl;
        v3 -= v5;
        cout << "v3 :   " << v3 << endl;
        */

//      v3 = v1 + v2;
//      cout << "v3 = v1 + v2 : " << v3 << endl;
```

```
        cout << "Appels constructeur par défaut : " <<
Vecteur::appels_constructeur_par_defaut << endl;
        cout << "Appels constructeur par copie : " <<
Vecteur::appels_constructeur_par_copie << endl;
        cout << "Appels constructeur avec dimension : " <<
Vecteur::appels_constructeur_avec_dim << endl;

        cout << "SORTIE TEST" << endl;
}
```