

C++

Création de classes
Opérateurs

private

vec2::vec2()

T& class::operator+=(T const& value);

Classe vec2

vec2.hpp

En têtes

```
#pragma once
#ifndef VEC2_HPP
#define VEC2_HPP

struct vec2
{
    float x;
    float y;

    vec2();
    vec2(float x_param, float y_param);
};

#endif
```

vec2.cpp

Implémentation

```
#include "vec2.hpp"

vec2::vec2()
    : x(0), y(0)
{
}

vec2::vec2(float x_param, float y_param)
    : x(x_param), y(y_param)
{
}
```

main.cpp

Appels à vec2

```
#include "vec2.hpp"
#include <iostream>

int main()
{
    vec2 v0(1.4, 5.5);
    std::cout << v0.x << std::endl;
    return 0;
}
```

En-tête de classe vec2

vec2.hpp

```
#pragma once

#ifndef VEC2_HPP
#define VEC2_HPP

struct vec2
{
    float x;
    float y;

    vec2();                                deux constructeurs prévus
    vec2(float x_param, float y_param);
};

#endif
```

Rem. Si on ne donne aucun constructeur, le compilateur génère un constructeur "vide" sans paramètre par défaut.

Mais si on définit au moins un constructeur, alors le constructeur par défaut n'est pas généré => Ne pas oublier de définir vec2().

Implémentation classe vec2

vec2.cpp

```
#include "vec2.hpp"
```

```
vec2::vec2()  
: x(0.0f), y(0.0f)
```

```
{
```

```
}
```

```
vec2::vec2(float x_param, float y_param)  
: x(x_param), y(y_param)
```

```
{
```

```
}
```

résolution de portée
indique que
l'on travaille
sur la classe vec2

liste d'initialisation

rien d'autre à faire

Encapsulation des attributs

vec2.hpp

```
#pragma once

class vec2
{
    float x_value;
    float y_value;
public:
    vec2();
    vec2(float x_param, float y_param);
};
```

*partie privée
non accessible
hors de la classe*

On ne peut plus accéder
directement à x et y

Accesseurs

Manipulation x,y façon Java

vec2.hpp

```
#pragma once

class vec2
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float get_x() const;
    float get_y() const;
    void set_x(float x);
    void set_y(float y);
};
```

get ne modifie
pas les attributs
de la classe
(~ final de Java)

Accesseurs

Manipulation x,y façon Java

vec2.hpp

```
#pragma once

class vec2
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float get_x() const;
    float get_y() const;
    void set_x(float x);
    void set_y(float y);
};
```

vec2.cpp

```
#include "vec2.hpp"

vec2::vec2()
    : x_value(0.0f), y_value(0.0f) {}

vec2::vec2(float x_param, float y_param)
    : x_value(x_param), y_value(y_param) {}

float vec2::get_x() const {return x_value;}
float vec2::get_y() const {return y_value;}

void vec2::set_x(float x) {x_value=x;}
void vec2::set_y(float y) {y_value=y;}
```

main.cpp

```
#include "vec2.hpp"
#include <iostream>

int main()
{
    vec2 v0(1.4f, 5.5f);
    v0.set_x(2.2f);
    v0.set_y(v0.get_x()+2.1f);
    std::cout<<v0.get_x()<<" "<<v0.get_y()<<std::endl;

    return 0;
}
```

pas très
agréable



Accesseurs

Manipulation x,y plus proche des math

vec2.hpp

```
#pragma once

class vec2
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float x() const;
    float y() const;
    float& x();
    float& y();
};
```

rôle du
set()

équivalent
d'un get()

main.cpp

```
#include "vec2.hpp"
#include <iostream>

int main()
{
    vec2 v0(1.4f, 5.5f);
    v0.x()=2.2f;
    v0.y()=v0.x()+2.1f;
    std::cout<<v0.x()<<" "<<v0.y()<<std::endl;
    return 0;
}
```

MAJ de
référence

simple
retour *const*

Accesseurs

Manipulation x,y plus proche des math

vec2.hpp

```
#pragma once

class vec2
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float x() const;
    float y() const;
    float& x();
    float& y();
};
```

main.cpp

```
#include "vec2.hpp"
#include <iostream>

int main()
{
    vec2 v0(1.4f, 5.5f);
    v0.x()=2.2f;
    v0.y()=v0.x()+2.1f;
    std::cout<<v0.x()<<"", "<<v0.y()<<std::endl;

    return 0;
}
```

vec2.cpp

```
#include "vec2.hpp"

vec2::vec2()
    :x_value(0.0f), y_value(0.0f) {}

vec2::vec2(float x_param, float y_param)
    :x_value(x_param), y_value(y_param) {}

float vec2::x() const {return x_value;}
float vec2::y() const {return y_value;}

float& vec2::x() {return x_value;}
float& vec2::y() {return y_value;}
```

Accesseurs

```
value& operator[] (int k);
```

En retournant une référence, la fonction externe retrouve l'accès à la variable

+ Manipulation x,y plus proche des math

- Perte d'encapsulation

est-elle pertinente?

Pour un `vec2`, accès direct aux attribut est OK

Contrairement aux "*standard Java*", l'accès aux attributs peut être toléré dans certains cas.

Fonction: produit scalaire

Deux approches:

1- Fonction membre (orientation Java)

vec2.hpp

```
#pragma once

class vec2
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float x() const;
    float y() const;
    float& x();
    float& y();

    float dot( vec2 const & v);
};
```

vec2.cpp

```
float vec2::dot( vec2 const & v)
{
    return x_value*v.x()+y_value*v.y();
}
```

main.cpp

```
vec2 v0(1.4f,5.5f);
vec2 v1(2.2f,4.8f);

float s=v0.dot(v1);

std::cout<<s<<std::endl;
```

Fonction: produit scalaire

Deux approches:

2- Fonction simple

Simple fonction
pas de **vec2::**

vec2.hpp

```
#pragma once

class vec2
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float x() const;
    float y() const;
    float& x();
    float& y();
};

float dot( vec2 const & v1, vec2 const & v2 );
```

vec2.cpp

```
float dot( vec2 const & v1, vec2 const & v2)
{
    return v1.x()*v2.x()+v1.y()*v2.y();
}
```

main.cpp

```
vec2 v0(1.4f, 5.5f);
vec2 v1(2.2f, 4.8f);

float s=dot(v0,v1);

std::cout<<s<<std::endl;
```

Fonction: produit scalaire

Deux approches: fonction membre VS fonction externe

Orienté Objet

```
float vec2::dot( vec2 const & v)
{
    return x_value*v.x()+y_value*v.y();
}
```

```
vec2 v0(1.4f,5.5f);
vec2 v1(2.2f,4.8f);

float s=v0.dot(v1);

std::cout<<s<<std::endl;
```

Avantage:

Objet contient ses fonctions

Inconvénient:

Non symétrique

Accès aux attributs privé de l'un
et pas de l'autre

Orienté Fonctionnel

```
float dot( vec2 const & v1, vec2 const & v2)
{
    return v1.x()*v2.x()+v1.y()*v2.y();
}
```

```
vec2 v0(1.4f,5.5f);
vec2 v1(2.2f,4.8f);

float s=dot(v0,v1);

std::cout<<s<<std::endl;
```

Avantage:

Symétrique

Encapsulation mieux préservée

Inconvénient:

Risque d'éparpillement des fonctions
IDE ne peut pas lister dot à partir de vec2

Effective C++: Prefer non-member non-friend functions
to member functions

Scott Meyers



Egalité entre vecteurs

Version fonction simple (pourrait être une fonction membre)

```
bool is_equal( vec2 const & v1, vec2 const & v2);
```

 vec2.hpp

```
bool is_equal( vec2 const & v1, vec2 const & v2)
{
    float epsilon=1e-6f;
    if(std::abs(v1.x()-v2.x())<epsilon &&
        std::abs(v1.y()-v2.y())<epsilon)
        return true;
    return false;
}
```

vec2.cpp



Ne jamais
comparer des float/double
avec ==

```
int main()
{
    vec2 v0(1.4f,5.5f);
    vec2 v1(2.2f,4.8f);
    vec2 v2(1.4f,5.5f);

    std::cout<<is_equal(v0,v1)<<std::endl;
    std::cout<<is_equal(v0,v2)<<std::endl;

    return 0;
}
```

main.cpp

std::abs() est générique
abs() ne l'est pas:
uniquement pour int

Opérateurs

On aimerait pouvoir écrire

`v1==v2`

plutôt que

`is_equal(v1,v2) //ou v1.is_equal(v2)`

- **Opérateur** est une *fonction* qui est appelée par un symbol particulier
- En C++, on peut définir l'opérateur `==`

Opérateurs ==

```
bool operator==(vec2 const & v1, vec2 const & v2);
```

vec2.hpp

```
bool operator==(vec2 const & v1, vec2 const & v2)
{
    float epsilon=1e-6f;
    if(std::abs(v1.x()-v2.x())<epsilon &&
        std::abs(v1.y()-v2.y())<epsilon)
        return true;
    return false;
}
```

vec2.cpp

```
int main()
{
    vec2 v0(1.4f,5.5f);
    vec2 v1(2.2f,4.8f);
    vec2 v2(1.4f,5.5f);

    if(v0==v1)
        std::cout<<"v0==v1"<<std::endl;
    if(v0==v2)
        std::cout<<"v0==v2"<<std::endl;

    return 0;
}
```

main.cpp

Appel de la fonction
operator==



Opérateurs !=

Très similaire à ==

```
bool operator==( vec2 const & v1, vec2 const & v2 );  
bool operator!=( vec2 const & v1, vec2 const & v2 );
```

vec2.hpp

```
bool operator!=( vec2 const & v1, vec2 const & v2 )  
{  
    return !(v1==v2);  
}
```

vec2.cpp

```
if(v0!=v1)  
    std::cout<<"v0!=v1"<<std::endl;
```

main.cpp

Réutiliser
ce que l'on a déjà codé
(surtout pas de copier-coller)



Opérateurs en fonctions membres

Au choix, les opérateurs peuvent être membres de la classe

```
class vec2                                     vec2.hpp
{
    float x_value;
    float y_value;
public:
    vec2();
    vec2(float x_param, float y_param);

    float x() const;
    float y() const;
    float& x();
    float& y();

    bool operator==(vec2 const & v) const;
    bool operator!=(vec2 const & v) const;
};
```

```
bool vec2::operator==(vec2 const & v) const;
{
    float epsilon=1e-6f;
    if(std::abs(x_value-v.x())<epsilon &&
        std::abs(y_value-v.y())<epsilon)
        return true;
    return false;
}

bool vec2::operator!=(vec2 const & v) const;
{
    return !(*this==v);
}                                              vec2.cpp
```

this = pointeur
sur l'objet courant

Aucune différence à l'utilisation

`v1==v2`

Liberté des paramètres

C++ = liberté du programmeur pour définir ses operateurs

```
bool operator==(vec2 const & v1, std::string const & v2);
```

vec2.hpp

```
bool operator==(vec2 const & v1, std::string const & v2)
{
    if(static_cast<unsigned int>(v1.x()+v1.y())==v2.length())
        return true;
    return false;
}
```

vec2.cpp

Les paramètres
peuvent être
de type différents

```
int main()
{
    vec2 v0(6.4f,4.5f);
    std::string s1="schtroumpf";
    std::string s2="maison";

    if(v0==s1)
        std::cout<<"v0==schtroumpf"<<std::endl;
    if(v0==s2)
        std::cout<<"v0==maison"<<std::endl;

    return 0;
}
```

main.cpp

Règle de bonne programmation

- Ne jamais utiliser les opérateur si le sens n'est pas trivial
- Ne pas sur-utiliser les opérateurs
rend rapidement le code illisible

`v1+v2 { v1.x-v2.x;}` : mauvaise idée

`v1=="schtroumpf"` : mauvaise idée

...

Opérateur +=

1er paramètre ref **non const** : doit être modifié

```
vec2& operator+=(vec2& v1, vec2 const & v2);
```

vec2.hpp

```
vec2& operator+=(vec2& v1, vec2 const & v2)
{
    v1.x()+=v2.x();
    v1.y()+=v2.y();

    return v1;
}
```

vec2.cpp

Traditionnellement
+= retourne une référence
vers la variable modifiée.

Permet d'imiter le comportement C
i +=3 += 9;

(Rarement utilisé, retour void est
également possible)

```
int main()
{
    vec2 v0(6.4f,4.5f);
    vec2 v1(1.2f,3.3f);

    v1+=v0;

    std::cout<<v1.x()<<" "<<v1.y()<<std::endl;

    return 0;
}
```

main.cpp

Opérateur += (version fonction membre)

un seul paramètre

```
vec2& operator+=( vec2 const & v2);
```

vec2.hpp

```
vec2& vec2::operator+=( vec2 const & v2)
{
    x_value+=v2.x();
    y_value+=v2.y();

    return *this;
}
```

vec2.cpp

```
int main()
{
    vec2 v0(6.4f,4.5f);
    vec2 v1(1.2f,3.3f);

    v1+=v0;

    std::cout<<v1.x()<<" "<<v1.y()<<std::endl;

    return 0;
}
```

main.cpp

Opérateur +

```
vec2 operator+( vec2 const & v1, vec2 const & v2);
```

 vec2.hpp

```
vec2 operator+( vec2 const & v1, vec2 const & v2)
```

 vec2.cpp

```
{  
    vec2 temp=v1;  
    temp+=v2;  
    return temp;  
}
```

← réutiliser ce qui a déjà été codé
(mieux que `vec2 (v1.x()+v2.x(), v1.y()+v2.y());`)

```
int main()
{
    vec2 v0(6.4f,4.5f);
    vec2 v1(1.2f,3.3f);
    vec2 v2=v0+v1;

    std::cout<<v2.x()<<" "<<v2.y()<<std::endl;

    return 0;
}
```

 main.cpp

Opérateur + (fonction membre)

```
vec2 operator+( vec2 const & v2) const; vec2.hpp
```

```
vec2 vec2::operator+( vec2 const & v2) const vec2.cpp
{
    vec2 temp=*this;
    temp+=v2;
    return temp;
}
```

```
int main() main.cpp
{
    vec2 v0(6.4f,4.5f);
    vec2 v1(1.2f,3.3f);
    vec2 v2=v0+v1;

    std::cout<<v2.x()<<" "<<v2.y()<<std::endl;

    return 0;
}
```

Opérateur -

Similaire à l'opérateur +

```
vec2& operator-=(vec2& v1, vec2 const & v2)
{
    v1.x()-=v2.x();
    v1.y()-=v2.y(); return v1;
}
vec2 operator-( vec2 const & v1, vec2 const & v2)
{
    vec2 temp=v1;
    temp-=v2;
    return temp;
}
```

Opérateur *= et *

vec2.hpp

```
vec2& operator*=(vec2& v, float s);  
vec2 operator*(vec2 const& v, float s);  
vec2 operator*(float s, vec2 const& v);
```

2 fonctions différentes
asymétrique

```
vec2& operator*=(vec2& v, float s)  
{  
    v.x()*=s;  
    v.y()*=s;  
    return v;  
}  
  
vec2 operator*(vec2 const& v, float s)  
{  
    vec2 temp=v;  
    temp*=s;  
    return temp;  
}  
  
vec2 operator*(float s, vec2 const& v)  
{  
    return v*s;  ← réutilisation  
}
```

vec2.cpp

main.cpp

```
vec2 v0(6.4f, 4.5f);  
vec2 v1=v0*2.0f;  
vec2 v2=2.0f*v0;
```

deux types d'appels possibles

Rem.

On ne peut pas définir
float * vec2 en tant que
fonction membre standard

Opérateur $\ast =$ et $\ast$

Doit-on définir:

```
vec2 operator*(const vec2& v1, const vec2& v2);
```

Que signifie ? $v1 \ast v2$?

- Produit composante à composante ?
- Produit scalaire ?
- Produit vectoriel ?

Dans le cas général, un doute subsiste

=> Meilleure solution: ne pas le définir

Evite de l'utiliser sans s'en rendre compte
(erreur silencieuse)

Opérateur /= et /

Similaire à l'opérateur *

Uniquement vec2/float, et pas float/vec2

Attention à ne pas diviser par 0

```
vec2& operator/=(vec2& v, float s)
{
    float epsilon=1e-6f;
    if(std::abs(s)<epsilon)
    {
        std::cout<<"Division par zero"<<std::endl;
        return v;
    }

    v.x()/=s;
    v.y()/=s;

    return v;
}

vec2 operator/( vec2 const & v, float s)
{
    vec2 temp=v;
    temp/=s;
    return temp;
}
```

Opérateur négation unaire

- Opérateur s'appliquant entre 2 paramètres: opérateur binaire
- Opérateur s'appliquant sur un seul paramètre: opérateur unaire

ex. `-vec2` : négation(`vec2`)

```
vec2 operator-( vec2 const & v);
```

`vec2.hpp`

```
vec2 operator-( vec2 const & v)  
{  
    return vec2(-v.x(), -v.y());  
}
```

`vec2.cpp`

```
vec2 v0(6.4f, 4.5f);  
vec2 v1=-v0;  
  
std::cout<<v1.x()<<" "<<v1.y()<<std::endl;
```

`main.cpp`

Opérateur négation unaire

Cas d'une fonction membre (pas de paramètres)

```
vec2 operator-() const; vec2.hpp
```

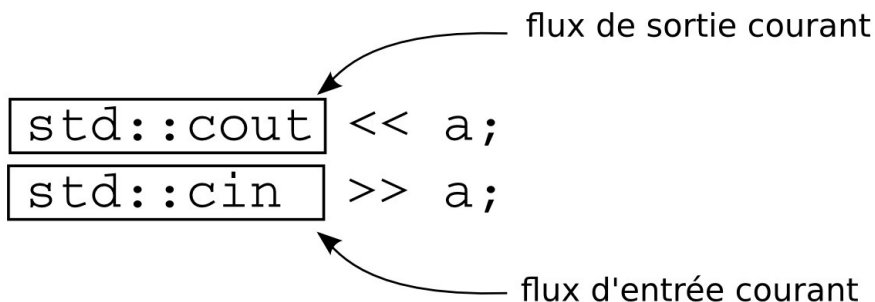
```
vec2 vec2::operator-() const vec2.cpp
{
    return vec2(-x_value, -y_value);
}
```

Opérateur flux de sortie

Les affichages, gestions de fichiers, mécanismes d'I/O fonctionnent par flux en C++

flux ~ buffer FIFO

ex.



<< : symbole/opérateur traditionnel d'entrée dans le flux

>> : symbole/opérateur traditionnel de sortie du flux

Opérateur flux de sortie <<

vec2.hpp

```
std::ostream& operator<<(std::ostream& s, vec2 const & v);
```

vec2.cpp

```
std::ostream& operator<<(std::ostream& s, vec2 const & v)
{
    s<<v.x()<<" "<<v.y();
    return s;
}
```

Envoie dans le flux de sortie

Retourne le flux

main.cpp

```
int main()
{
    vec2 v0(6.4f, 4.5f);
    std::cout<<v0<<std::endl;

    return 0;
}
```

std::ostream

flux de sortie(Output Stream)

affiche 6.4,4.5

Opérateur flux de sortie <<

Flux est générique:

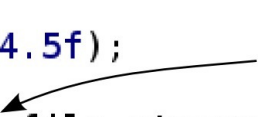
fonctionne directement pour des fichiers

```
#include "vec2.hpp"

#include <iostream>
#include <string>
#include <fstream>

int main()
{
    vec2 v0(6.4f, 4.5f);
    std::ofstream file_stream("mon_fichier.txt");
    file_stream<<v0;
    file_stream.close();

    return 0;
}
```



flux de sortie dans
un fichier

fichier mon_fichier.txt contient
6.4,4.5

Classe vec2

```
#pragma once

#include <iostream>

class vec2
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float x() const;
    float y() const;
    float& x();
    float& y();
};

vec2& operator+=(vec2& v1, vec2 const & v2);
vec2 operator+(vec2 const & v1, vec2 const & v2);

vec2 operator-(vec2 const & v);
vec2& operator-=(vec2& v1, vec2 const & v2);
vec2 operator-(vec2 const & v1, vec2 const & v2);

vec2& operator*=(vec2& v, float s);
vec2 operator*(vec2 const & v, float s);
vec2 operator*(float s, vec2 const & v);

vec2& operator/=(vec2& v, float s);
vec2 operator/(vec2 const & v, float s);

bool operator==(vec2 const & v1, vec2 const & v2);
bool operator!=(vec2 const & v1, vec2 const & v2);

std::ostream& operator<<(std::ostream& s, vec2 const & v);

float dot(vec2 const & v1, vec2 const & v2);
```

```
int main()
{
    vec2 v0(6.4f, 4.5f);
    vec2 v1(4.1f, 2.2f);
    vec2 v2(1.1f, -2.5f);

    std::cout<< 5.0f*(v0-2.0f*v1)-v2/1.9f <<std::endl;

    return 0;
}
```

Opérations mathématiques
ressemble à des maths

- Plus lisible et simple d'utilisation que C
- Plus lisible et plus rapide que Java
- Plus structuré que Matlab

Opérateur []

```
class vec2                                vec2.hpp
{
    float x_value;
    float y_value;

public:
    vec2();
    vec2(float x_param, float y_param);

    float x() const;
    float y() const;
    float& x();
    float& y();

    float operator[](int k) const;
};
```

```
int main()                                main.cpp
{
    vec2 v0(6.4f, 4.5f);

    std::cout<< v0[0] <<std::endl;
    std::cout<< v0[1] <<std::endl;
    std::cout<< v0[2] <<std::endl;

    return 0;
}
```

```
float vec2::operator[](int k) const        vec2.cpp
{
    switch(k)
    {
        case 0:
            return x_value;
        case 1:
            return y_value;
        default:
            std::cerr<<"Erreur ["<<k<<"]"<<std::endl;
            exit(1);
    }
}
```

comportement d'un tableau
(sécurité en +)



Opérateur []

Version référence non const

```
float& vec2::operator[](int k)  
{
    switch(k)
    {
        case 0:
            return x_value;
        case 1:
            return y_value;
        default:
            std::cerr<<"Erreur ["<<k<<"]"<<std::endl;
            exit(1);
    }
}
```

vec2.cpp

```
int main()
{
    vec2 v0(6.4f, 4.5f);
    v0[1]=12.4f;
    std::cout<<v0<<std::endl;

    return 0;
}
```

Manipulation comme un tableau



Opérateur ()

- L'opérateur [] ne peut prendre qu'un seul paramètre:
=> m[4,5] est impossible en C++
- L'opérateur () peut prendre plusieurs paramètres:
=> m(4,5)

```
float vec2::operator()(int k) const
{
    return (*this)[k];
}
float& vec2::operator()(int k)
{
    return (*this)[k];
}
```

vec2.cpp

```
float operator()(int k) const;
float& operator()(int k);
```

vec2.hpp

```
int main()
{
    vec2 v0(6.4f, 4.5f);
    v0(1)=12.4f;
    std::cout<<v0<<std::endl;

    return 0;
}
```

main.cpp

vecteur type Matlab



Fonctions génériques

Exemple d'utilisation de l'opérateur ()

Classe symbolisant une fonction

```
class gaussian
{
public:
    gaussian(float x0, float sigma);
    float operator()(float x) const;
private:
    float x0;
    float sigma;
};
```

```
gaussian::gaussian(float x0_param, float sigma_param)
    : x0(x0_param), sigma(sigma_param)
{}

float gaussian::operator()(float x) const
{
    return std::exp( -(x-x0)*(x-x0) / sigma );
}
```

```
gaussian g(1.0f, 0.5f);

std::cout<<g(0.8)<<std::endl;
std::cout<<g(0.9)<<std::endl;
std::cout<<g(1.0)<<std::endl;
std::cout<<g(1.1)<<std::endl;
std::cout<<g(1.2)<<std::endl;
```

Paramètres

Utilisation sous forme
 $y=f(x)$

Fonctions génériques

Utilisation d'un objet pour encapsuler une fonction
Passage en paramètre aisé

```
void affiche_valeur( gaussian const & g)
{
    std::cout<<g(0.8)<<std::endl;
    std::cout<<g(0.9)<<std::endl;
    std::cout<<g(1.0)<<std::endl;
    std::cout<<g(1.1)<<std::endl;
    std::cout<<g(1.2)<<std::endl;
}

int main()
{
    gaussian g1(1.0f,0.5f);
    gaussian g2(2.1f,7.8f);

    affiche_valeur(g1);
    affiche_valeur(g2);

    return 0;
}
```

Utilisation d'un objet comme
une fonction

=> Foncteur (/Functor)

Remplace avantageusement
les pointeurs de fonctions

- Plus lisible, plus simple
- Plus rapide!

Fonctions génériques

Exemple d'afficheur de fonction/courbe générique

```
void curve_drawer( generic_function const & f)
{
    int N=30;

    float x0=-3.0f;
    float x1=+3.0f;

    float dx=1.0f/N;

    for(int k=0;k<N;++k)
    {
        float x=x0+k*dx*(x1-x0);
        float y=f(x);

        //affichage d'un point
        std::cout<<x<<" "<<y<<std::endl;
    }
}
```

Fonctions génériques

Exemple d'afficheur de fonction/courbe générique

```
#pragma once

class generic_function      classe polymorphe
{                           (virtuelle pure)
public:
    virtual float operator()(float x) const =0;
    ~generic_function(){}

class cosinus : public generic_function
{
public:
    cosinus(float frequency);
    float operator()(float x) const;
private:
    float frequency;
};

class gaussian: public generic_function
{
public:
    gaussian(float x0, float sigma);
    float operator()(float x) const;
private:
    float x0;
    float sigma;
};
```

```
#include "function.hpp"

#include <cmath>
#include <iostream>

cosinus::cosinus(float frequency_param)
    :frequency(frequency_param)
{}

float cosinus::operator()(float x) const
{
    return std::cos(2.0f*M_PI*frequency*x);
}

gaussian::gaussian(float x0_param, float sigma_param)
    :x0(x0_param), sigma(sigma_param)
{}

float gaussian::operator()(float x) const
{
    return std::exp( -(x-x0)*(x-x0) / sigma );
}
```

```
int main()
{
    cosinus ma_fonction_1(1.24f);
    gaussian ma_fonction_2(1.0f, 0.5f);
    curve_drawer(ma_fonction_1);
    curve_drawer(ma_fonction_2);

    return 0;
}
```

Fonctions génériques

Exemple d'afficheur de fonction/courbe générique

Polymorphisme: lent

=> Remplacement avantageux par **Template**

```
#pragma once

class cosinus
{
public:
    cosinus(float frequency);
    float operator()(float x) const;
private:
    float frequency;
};

class gaussian
{
public:
    gaussian(float x0, float sigma);
    float operator()(float x) const;
private:
    float x0;
    float sigma;
};
```

+ Plus besoin d'héritage

Paramètre template

```
template <typename T>
void curve_drawer(T const& f)
{
    int N=30;

    float x0=-3.0f;
    float x1=+3.0f;

    float dx=1.0f/N;

    for(int k=0; k<N; ++k)
    {
        float x=x0+k*dx*(x1-x0);
        float y=f(x);

        //affichage d'un point
        std::cout<<x<<" "<<y<<std::endl;
    }
}
```

+ Paramètre connu à la compilation
=> plus rapide que le polymorphisme