```cpp
//**********************************//
//RAPPEL DE C++
//**********************************//


//1. Classes et struct

//une struct peut être vue comme une classe dont tout est publique.
//notez qu'une struct peut, tout comme les classes avoir un constructeur
//ainsi que la surcharge des opérateurs.

//exemple de struct
struct mon_vecteur
{
    //constructeur
    mon_vecteur(double _x,double _y,double _z)
        :x(_x),y(_y),z(_z)//rappel: ceci est l'initialisation pour le
constructeur
    {}

    //paramètres
    double x,y,z;
};

//exemple de classe:
class mon_vecteur2
{
public:
    mon_vecteur2(double _x,double _y,double _z)
        :x(_x),y(_y),z(_z){}

private:
    //ici on ne pourra pas accéder de l'exterieur aux paramètres private
    double x,y,z;
};

//utilisation
int main(int argc,char *argv[])
{
    mon_vecteur(1,2,3);
    mon_vecteur2(1,2,3);
    return 0;
}
```



```cpp
//implementation
mon_vecteur mon_vecteur::addition(const mon_vecteur& vec) const
{return mon_vecteur(x+vec.get_x(),y+vec.get_y(),z+vec.get_z());}
mon_vecteur::mon_vecteur(double _x,double _y,double _z)
    :x(_x),y(_y),z(_z){}
double mon_vecteur::get_x()const{return x;}
double mon_vecteur::get_y()const{return y;}
double mon_vecteur::get_z()const{return z;}
double& mon_vecteur::get_x() {return x;}
double& mon_vecteur::get_y() {return y;}
double& mon_vecteur::get_z() {return z;}
void mon_vecteur::get_values(double *_x,double *_y,double *_z)
{if(_x!=0) *_x=x; if(_y!=0) *_y=y; if(_z!=0) *_z=z;}



//exemple de classe non correct car défini un comportement non standard
class mon_vecteur2 : public mon_vecteur //par heritage, les méthodes get_x sont
accessibles
{
private:
    double x,y,z;
public:
    mon_vecteur2(double _x,double _y,double _z)
            :mon_vecteur(_x,_y,_z){}

    //ici l'appel à addition va modifier le vecteur courant!
    mon_vecteur2 addition(mon_vecteur2& vec)
    {
        vec.get_x()+=x;vec.get_y()+=y;vec.get_z()+=z;
        return vec;
    }
};

//utilisation
int main(int argc,char *argv[])
{
    mon_vecteur a(1,2,3);
    mon_vecteur b(4,5,6);
    mon_vecteur c=a.addition(b); //c=(5,7,9), (a et b) inchangés
    c.get_y()=8; //c=(5,8,9)

    mon_vecteur2 a2(1,2,3);
    mon_vecteur2 b2(4,5,6);
    mon_vecteur2 c2=a2.addition(b2); //c=(5,7,9) mais b2=(5,7,9) aussi!!!

    return 0;
}
```



```cpp
//**********************************//
//RAPPEL DE C++
//**********************************//


//2. Passage d'arguments

//En C et C++, sauf mention explicite les arguments sont passés par copies.
//Ce type de passage est à réserver pour les variables de bases: entiers,
float,double
//Il ne faut pas passer de struct, classes, vecteurs par copie: perte d'espace
mémoire + perte de temps
//En C, on passe ce type d'arguments par pointeurs.
//En C++, on peut les passer par pointeurs (maClass*) ou référence (maClass&)
//Le pointeur nécessite un déférencement (*mon_objet), alors que la référence
permet
// de manipuler l'objet de manière classique.
// Règle standard en C++:
//   * Lorsque l'argument est modifié, on le passe par pointeur.
//   * Lorsque l'argment n'est pas modifié, on le passe par référence constante.
//Modifier un argument passé par référence n'est pas une programmation propre
// et génère des actions inatendues!
//

class mon_vecteur
{

public:
    mon_vecteur(double _x,double _y,double _z);

    //additionne un vecteur à celui courant (notez le passage d'argument en
reference constante)
    //Le mot clé const en fin de méthode indique que cette fonction ne modifie
pas
    // les paramètres internes (x,y,z) de la classe courante.
    mon_vecteur addition(const mon_vecteur& vec) const;

    //retourne les valeurs de (x,y,z) sans modifier la classe courante
    double get_x()const;
    double get_y()const;
    double get_z()const;

    //permet d'acceder et d'affecter les valeurs de x,y,z car elles sont
    // passés par références.
    // ex. mon_vecteur a(4,5,6);
    //     a.get_x()=78; //on a alors a=(78,5,6);
    // Notez que le mot clé const n'apparait pas en fin de déclaration
    // Le compilateur sait différencier l'appel à get_x()const et get_x() en
fonction du contexte.
    double& get_x();
    double& get_y();
    double& get_z();

    // retourne dans les pointeurs les valeurs de x,y,z
    // Notez que l'on prend garde de ne pas écrire sur le pointeur NULL
    void get_values(double *_x,double *_y,double *_z);

private:
    double x,y,z;

};
```



```cpp
#include <iostream>

//**********************************//
//RAPPEL DE C++
//**********************************//


//3. Surcharge d'opérateurs

//En C++, les classes et structs peuvent voir leur opérateurs être surchargés.
//Ce la permet, entre autres, de définir des opérations ressemblant à des
formules de maths.
//On utilisera cependant ces opérateurs uniquement pour des opérations non
ambigues.
// ex. vecteur1 x vecteur2 est ambigue: produit scalaire? vectoriel? terme à
terme?
//     vecteur1<<vecteur2 ? concatenation, remplacement ?
//     Notez que le signe + est parfois utilisé pour la somme, parfois pour la
//  concatenation, ce qui peut également prêter à des ambiguités.

class mon_vecteur
{

public:
    mon_vecteur(double _x,double _y,double _z);

    double get_x() const;
    double get_y() const;
    double get_z() const;


    //************************************
    //opérateurs "mathématiques"
    //************************************

    //operateurs internes:

    //surchage de l'opérateur += avec un autre vecteur
    //(notez que l'on retourne une référence de l'objet courant,
    // car on modifie la classe courant, mais pas le paramètre)
    mon_vecteur& operator+=(const mon_vecteur& vec);
    //surchage de l'opérateur += avec un scalaire
    mon_vecteur& operator+=(const double& valeur);
    //sucharge de l'opérateur -= avec un autre vecteur
    mon_vecteur& operator-=(const mon_vecteur& vec);
    //sucharge de l'opérateur -= avec un scalaire
    mon_vecteur& operator-=(double a);
    //surcharge de l'opérateur *= avec un scalaire
    mon_vecteur& operator*=(const double& value);
    //surcharge de l'opérateur /= avec un scalaire
    mon_vecteur& operator/=(const double& value);

    //operateur de négation
    //(on ne modifie pas la classe, mais on retourne une copie)
    mon_vecteur operator-() const;

    //operateurs "friend" (les friends ne sont pas hérités par les classes
dérivées)

    //surchage de vec_a+vec_b: ici on ne modifie plus la classe
    //Il ne s'agit pas d'une opération de la classe à proprement parler, on le
défini en friend
```

```
    friend mon_vecteur operator+(const mon_vecteur& vec_a,const mon_vecteur&
vec_b);
    //surchage de scalaire+vec (nécessite une définition par friend!)
    friend mon_vecteur operator+(double a,const mon_vecteur& vec);
    //surchage de vec+scalaire
    friend mon_vecteur operator+(const mon_vecteur& vec,double a);
    //on peut faire de même avec l'operator- ...
    friend mon_vecteur operator-(const mon_vecteur& vec_a,const mon_vecteur&
vec_b);
    friend mon_vecteur operator-(double a,const mon_vecteur& vec);
    friend mon_vecteur operator-(const mon_vecteur& vec,double a);

    //surchage de scalaire*vec
    friend mon_vecteur operator*(double a,const mon_vecteur& vec);
    friend mon_vecteur operator*(const mon_vecteur& vec,double a);
    //surchage de vec/scalaire (attention, pas de scalaire/vec !)
    friend mon_vecteur operator/(const mon_vecteur& vec,double a);


    //*************************************
    //opérateurs crochet et parenthèses
    //*************************************

    //On va écrire un vecteur tel que
    // "vec[0]=vec(0)=x"
    // "vec[1]=vec(1)=y"
    // "vec[2]=vec(2)=z"

    //operateur crochet: permet d'écrire: double z=vec[2];
    // Notons que l'on ne modifie pas la classe, on peut donc appeler la méthode
sur un objet constant
    double operator[](unsigned int k) const;
    //Note: on aurait pu écrire aussi: const double& operator[](unsigned int k)
const;

    //idem pour la parenthèse: permet d'écrire: double z=vec(2);
    double operator()(unsigned int k) const;

    //operateur crochet d'affectation: permet d'écrire vec[2]=4; (sur un objet
non constant)
    double& operator[](unsigned int k); //pas de const à la fin
    //idem pour la parenthèse
    double& operator()(unsigned int k);

    //*************************************
    //opérateurs de sortie standard
    //*************************************

    // on écrit ici les données dans un flux, classiquement pour écrire avec
std::cout<<
    // Ici, on a un exemple d'écriture sur une référence non constante passée en
paramètre (le flux)
    friend std::ostream& operator<<(std::ostream& flux,const mon_vecteur& vec);



private:
    double x,y,z;

};
```



```
//implementation

mon_vecteur::mon_vecteur(double _x,double _y,double _z):x(_x),y(_y),z(_z){}
double mon_vecteur::get_x() const{return x;}
double mon_vecteur::get_y() const{return y;}
double mon_vecteur::get_z() const{return z;}

mon_vecteur& mon_vecteur::operator+=(const mon_vecteur& vec)
{
    x+=vec.get_x();y+=vec.get_y();z+=vec.get_z();
    return *this; //on retourne l'objet courant
}

mon_vecteur& mon_vecteur::operator+=(const double& valeur)
{x+=valeur;y+=valeur;z+=valeur;return *this;}
mon_vecteur& mon_vecteur::operator-=(const mon_vecteur& vec)
{x-=vec.get_x();y-=vec.get_y();z-=vec.get_z();return *this;}
mon_vecteur& mon_vecteur::operator-=(double a)
{return (*this)+=-a;} //on peut se servir de ce que l'on a déjà codé (simplifie
la maintenance future)
mon_vecteur& mon_vecteur::operator*=(const double& value)
{x*=value;y*=value;z*=value;return *this;}
mon_vecteur& mon_vecteur::operator/=(const double& value)
{x/=value;y/=value;z/=value;return *this;}
mon_vecteur mon_vecteur::operator-() const
{return mon_vecteur(-x,-y,-z);}//notez que l'on duplique l'objet

//notez que pour les méthodes "friend", on a pas l'opérateur de résolution de
portée
// dans l'implémentation (mon_vecteur::)
mon_vecteur operator+(const mon_vecteur& vec_a,const mon_vecteur& vec_b)
{
    mon_vecteur temp=vec_a; //on duplique l'objet (car on ne modifie pas l'objet
courant)
    temp+=vec_b; //on se sert généralement des opérateurs "internes"
    return temp; //on retourne l'objet modifié
}

mon_vecteur operator+(double a,const mon_vecteur& vec)
{mon_vecteur temp=vec;temp+=a;return temp;}
mon_vecteur operator+(const mon_vecteur& vec,double a)
{return a+vec;}//on se sert de ce que l'on a déjà codé
mon_vecteur operator-(const mon_vecteur& vec_a,const mon_vecteur& vec_b)
{mon_vecteur temp=vec_a;temp-=vec_b;return temp;}

mon_vecteur operator-(double a,const mon_vecteur& vec)
{mon_vecteur temp=-vec;temp+=a;return temp;}//on se sert de la négation
mon_vecteur operator-(const mon_vecteur& vec,double a)
{mon_vecteur temp=vec;temp-=a;return temp;}
mon_vecteur operator*(double a,const mon_vecteur& vec)
{mon_vecteur temp=vec;temp*=a;return temp;}
mon_vecteur operator*(const mon_vecteur& vec,double a)
{return a*vec;}
mon_vecteur operator/(const mon_vecteur& vec,double a)
{return vec*(1/a);}
double mon_vecteur::operator[](unsigned int k) const
{
    if(k==0)
        return x;
    if(k==1)
        return y;
    if(k==2)
```



```
        return z;
    std::cout<<"Erreur, k="<<k<<std::endl;
    throw std::exception(); //lance une exception pouvant etre rattrapé
ultérieurement
    // sans faire cracher le programme
}
double mon_vecteur::operator()(unsigned int k) const
{
    return (*this)[k];
}
double& mon_vecteur::operator[](unsigned int k)
{
    //implémentation identique à la version non const
    // (mais peut être différent dans le cas de structures dynamique
    //   ex. vecteur de taille s'adaptant automatiquement)
    if(k==0)
        return x;
    if(k==1)
        return y;
    if(k==2)
        return z;
    std::cout<<"Erreur, k="<<k<<std::endl;
    throw std::exception();
}
double& mon_vecteur::operator()(unsigned int k)
{
    return (*this)[k];
}

std::ostream& operator<<(std::ostream& flux,const mon_vecteur& vec)
{
    flux<<"("<<vec.get_x()<<","<<vec.get_y()<<","<<vec.get_z()<<")";
    return flux;
}


//utilisation
int main(int argc,char *argv[])
{
    mon_vecteur a(1,2,3);
    mon_vecteur b(4,5,6);
    mon_vecteur c=a+b; //c=(5,7,9)

    mon_vecteur d=10-c; //c=(5,3,1)
    std::cout<<d<<std::endl; //affiche c
    mon_vecteur e=d; //e=(5,3,1)
    e[0]=b(1); //e.x=b.y => e=(5,3,1);
    e+=mon_vecteur(0,1,1); //e=(5,4,2);

    std::cout<<"voici mon vecteur final: "<<e<<std::endl;

    return 0;
}
```



```
#include <vector>
#include <list>
#include <set>
#include <map>
#include <string>
#include <iostream>


//*********************************//
//RAPPEL DE C++
//*********************************//


//4. Conteneurs de la STL (Standard Library)

//En C++, on utilisera tant que possible les conteurs de la STL.
//Plus générique, plus sûr que les pointeurs. Plus performant et générique
// que les conteneurs définis manuellement.



class mon_vecteur
{
private:
    double x_interne,y_interne,z_interne;
public:
    mon_vecteur():x_interne(0),y_interne(0),z_interne(0){}
    mon_vecteur(double _x,double _y,double
_z):x_interne(_x),y_interne(_y),z_interne(_z){}

    double x() const{return x_interne;}
    double y() const{return y_interne;}
    double z() const{return z_interne;}

    friend std::ostream& operator<<(std::ostream& flux,const mon_vecteur& vec)
    {flux<<"("<<vec.x()<<","<<vec.y()<<","<<vec.z()<<")";return flux;}
};


//classe d'une date de création que l'on utilisera comme clé dans une map
class date
{
   public:
    int jour;
    int mois;
    int annee;


    date():jour(0),mois(0),annee(0){}
    date(int _jour,int _mois,int _annee):jour(_jour),mois(_mois),annee(_annee){}

    friend std::ostream& operator<<(std::ostream& flux,const date& vec)
    {flux<<vec.jour<<"/"<<vec.mois<<"/"<<vec.annee;return flux;}
};

//classement de deux date par un foncteur (classe définissant l'opérateur())
class date_less//classe définissant si date1<date2 ou non par l'appel à
l'opérateur parenthèse
{
public:
```

```
    //est ce que date1<date2?
    bool operator()(const date& date1,const date& date2)
    {
        if(date1.annee<date2.annee)
            return true;
        if(date1.annee==date2.annee)
        {
            if(date1.mois<date2.mois)
                return true;
            if(date1.mois==date2.mois)
                if(date1.jour<date2.jour)
                    return true;
        }
        return false;
    }
};

//utilisation
int main(int argc,char *argv[])
{


    // ************************* //
    // Vecteur
    // ************************* //

    //un vecteur d'entier
    std::vector<int> vec(10); //idem que: vec int[10];
    //parcours
    for(int k=0,N=vec.size();k<N;++k) //ici N=10 (le conteneur connait sa taille)
        vec[k]=k+8; //acces classique aléatoire comme avec un tableau.

    //par iterateur (commun avec les listes, graphes, ... etc)
    std::vector<int>::iterator it;
    for(it=vec.begin();it!=vec.end();++it)
        *it=vec[1]+1; // it agit comme un pointeur: on peut le déférencer

    //parcours par iterateur constant et remplissage de taille dynamique:
    std::vector<int> vec2;
    for(std::vector<int>::const_iterator it=vec.begin();it!=vec.end();++it)
        vec2.push_back(*it+1); //vec2 augmente sa taille dynamiquement et vaut
"vec+1"

    //un vecteur d'un type quelconque
    std::vector<mon_vecteur> V;
    V.push_back(mon_vecteur(4,7,8));
    V.push_back(mon_vecteur(0,1,2));
    //V contient ici 2 classes
    V[1]=V[0];//on affecte directement la valeur de l'entrée 1

    //Dans certains cas, il est nécessaire d'accéder directement aux pointeurs.
    // Le vecteur est stocké de manière contigue, on peut donc revenir à la
notation C:
    mon_vecteur* pointeur=&V[0];
    std::cout<<*pointeur<<" ; "<<*(pointeur+1)<<std::endl;
    pointeur[1]=mon_vecteur(-1,-1,-2);

    //A tout moment, on peut dynamiquement changer la taille du vecteur
    V.resize(15);
    V[14]=mon_vecteur(8,8,9);

    //Rappel:
```



```
    // Vecteur= accès aléatoire en O(1)
    // Recherche en O(N)
    // Ajout et Suppression d'un élément en milieu de tableau en O(N)
(redimentionnement, copie, ...)




    // ************************* //
    // Listes
    // ************************* //

    //les listes permettent des ajouts et des suppressions d'un élément en O(1)
peut importe leur positionnement
    //exemple pour une liste de chaine de characteres

    std::list<std::string> ma_liste;
    ma_liste.push_back(" world");
    ma_liste.push_front("hello");
    ma_liste.push_back(" !");

    //affiche "hello world !"
    for(std::list<std::string>::const_iterator
it=ma_liste.begin();it!=ma_liste.end();++it)
        std::cout<<*it;
    std::cout<<std::endl;

    //modification d'un mot
    std::list<std::string>::iterator it_modification=ma_liste.begin();
    *it_modification="Good morning"; //modifi le premier mot

    //insere un element en après le mot hello
    std::list<std::string>::iterator it_insertion=ma_liste.begin();
    ++it_insertion; //on se place après "hello"
    ma_liste.insert(it_insertion,std::string(" my very ") + "wonderful");
//concatenation de chaine de caractere

    //suppression de "hello";
    ma_liste.erase(it_insertion); //attention it_insertion est à mettre à jour
    //insertion d'un autre mot
    it_insertion=ma_liste.end();--it_insertion;
    ma_liste.insert(it_insertion," life");

    //affichage de la nouvelle phrase
    for(std::list<std::string>::const_iterator
it=ma_liste.begin();it!=ma_liste.end();++it)
        std::cout<<*it;
    std::cout<<std::endl;

    //Rappel:
    // Recherche et accès (aléatoire) en O(N)
    // Ajout et Suppression d'un élément en milieu de tableau en O(1)




    // ************************* //
    // Set est un graphe binaire qui classe automatiquement les éléments du plus
petit au plus grand
    // ************************* //
```



```
    //exemple d'un arbre d'entier
    std::set<int> mon_arbre;
    mon_arbre.insert(4);
    mon_arbre.insert(7);
    mon_arbre.insert(-8);
    mon_arbre.insert(9);
    mon_arbre.insert(-8);//doublon non ajouté

    //affichage dans l'ordre
    for(std::set<int>::const_iterator it=mon_arbre.begin();it!=mon_arbre.end();
++it)
        std::cout<<*it<<" ; ";
    std::cout<<std::endl;

    //suppression d'un élément pointé
    std::set<int>::iterator it_set=mon_arbre.begin();
    ++it_set;++it_set;mon_arbre.erase(it_set);

    //suppression d'un élément donné par la clé (recherche en log(N))
    mon_arbre.erase(4);
    //recherche d'un élément
    std::set<int>::iterator it_set_find=mon_arbre.find(4);//cherche element 4
    if(it_set_find==mon_arbre.end())
        std::cout<<"element 4 non trouve dans le std::set"<<std::endl;

    //affichage dans l'ordre
    for(std::set<int>::const_iterator it=mon_arbre.begin();it!=mon_arbre.end();
++it)
        std::cout<<*it<<" ; ";
    std::cout<<std::endl;



    // ************************* //
    // Map est un type particulier de "std::set" contenant une clé pour
l'ordonnancement, et un objet associé
    // ************************* //

    //exemple de coordonnées associées à une clée d'ordonnancement (ici une
chaine de caractère)
    std::map<std::string,mon_vecteur> ma_map;

    //on insere les element sous forme de paire (clé,objet)
    ma_map.insert(std::make_pair("vecteur_01",mon_vecteur(1,2,3)));
    ma_map.insert(std::make_pair("vecteur_00",mon_vecteur(4,7,8)));
    std::string chaine_1="vecteur_36";mon_vecteur v(4,7,8);
    ma_map.insert(std::make_pair(chaine_1,v));
    ma_map.insert(std::make_pair("vecteur_01",mon_vecteur(-1,-1,-1)));//doublon
de clé non ajouté
    ma_map.insert(std::make_pair("vecteur_42",mon_vecteur(7,7,7)));

    //affichage de la map
    std::map<std::string,mon_vecteur>::const_iterator it_map;
    for(it_map=ma_map.begin();it_map!=ma_map.end();++it_map)
        std::cout<<it_map->first<<" : "<<it_map->second<<std::endl;//acces par
une paire

    //suppresion d'un element désigné par la clé
    ma_map.erase("vecteur_00");
    //tentative de suppression d'un element non existant
    ma_map.erase("vecteur_420");
    //recherche d'un element
```



```
    if(ma_map.find("vecteur_42")!=ma_map.end()) //return .end() si non trouvé
        std::cout<<"element vecteur_42 existe dans la map"<<std::endl;


    //cas d'une clé définie soit même
    // ex. clé = date de création
    std::map<date,std::string,date_less> ma_map2; //notez le 3eme element utilisé
pour le classement

    ma_map2.insert(std::make_pair(date(1,1,1889),"construction tour eiffel"));
    ma_map2.insert(std::make_pair(date(18,05,2021),"sortie de Windows 12"));
    ma_map2.insert(std::make_pair(date(29,4,2011),"mariage prince william"));
    ma_map2.insert(std::make_pair(date(17,04,2666),"fin du monde"));
    ma_map2.insert(std::make_pair(date(1,1,-3500),"invention de la roue"));
    ma_map2.insert(std::make_pair(date(15,9,2011),"rentree des classes"));
    ma_map2.insert(std::make_pair(date(14,9,1515),"bataille de Marignan"));

    //suppression d'un element désigné par la clé
    ma_map2.erase(date(15,9,2011));

    //affichage de la map
    for(std::map<date,std::string,date_less>::const_iterator
it=ma_map2.begin(),it_end=ma_map2.end();it!=it_end;++it)
        std::cout<<it->first<<" => "<<it->second<<std::endl;



    //Rappel:
    // Recherche et accès (aléatoire) en O(log(N))
    // Ajout et Suppression d'un élément en milieu de tableau en O(log(N))


    return 0;
}
```

```cpp

#include <iostream>
#include <string>
#include <list>
#include <vector>


//*********************************//
//RAPPEL DE C++
//*********************************//


//5. En C++ il existe 2 approches de généricité
// L'orienté objet par polymorphisme = virtual.
// Les templates.
//
// Avantages/inconvenients principaux:
//  Polymorphisme:
//                  + adaptable dynamiquement
//                  + simple et transparent dans le code
//                  - surchage du temps de calcul à l'execution
//  Template:
//                  + Evalué à la compilation: aucune perte de temps dynamique
//                  - Non dynamique pendant l'execution
//                  - Complexifie le code et la compilation
//


// ********************************************************************* //
//1. Le polymorphisme permet de connaitre le type de classe de manière dynamique.
// ********************************************************************* //

class objet_geometrique
{
public:
    objet_geometrique(const std::string& _nom):nom(_nom){}

    virtual void affiche() const //la méthode virtuelle spécialisée dans chaque
cas
    {std::cout<<"Je ne sais pas qui je suis ("<<nom<<")"<<std::endl;}
protected:
    std::string nom;
};

class disque : public objet_geometrique
{
public:
    disque(const std::string& nom,double _centre_x,double _centre_y,double
_rayon)
        :objet_geometrique(nom),centre_x(_centre_x),centre_y(_centre_y),rayon(_ra
yon)
    {}

    virtual void affiche() const
    {
        std::cout<<"Je suis un disque ("<<nom<<") de rayon "<<rayon<<", et de
centre ("<<centre_x<<","<<centre_y<<")"<<std::endl;
        //fait son affichage propre...
    }
private:
    double centre_x,centre_y;
```



```cpp
    double rayon;
};
class rectangle: public objet_geometrique
{
public:
    rectangle(const std::string& nom,double _x0,double _y0,double _x1,double _y1)
        :objet_geometrique(nom),x0(_x0),y0(_y0),x1(_x1),y1(_y1)
    {}


    virtual void affiche() const
    {
        std::cout<<"Je suis un rectangle ("<<nom<<"), mes cotes sont situes en
("<<x0<<","<<y0<<") ; ("<<x1<<","<<y1<<")"<<std::endl;
        //fait sont affichage propre...
    }
private:
    double x0,y0;
    double x1,y1;
};



// ********************************************************************* //
// 2. Les templates permettent de definir des classes generiques
// ********************************************************************* //


//une seule declaration et implementation de ce vecteur permet de gérer des types
divers
template <typename MON_TYPE>
class vecteur_generique
{
private:
    MON_TYPE x,y,z;//ce vecteur contient 3 coordonnees de type quelconque

public:

    vecteur_generique(){}
    //notez que l'on passera par référence constante pour éviter la copie
d'élément non connu à l'avance
    vecteur_generique(const MON_TYPE& _x,const MON_TYPE& _y,const MON_TYPE& _z)
        :x(_x),y(_y),z(_z){}

    const MON_TYPE& get_x() const {return x;}
    const MON_TYPE& get_y() const {return y;}
    const MON_TYPE& get_z() const {return z;}

    MON_TYPE& get_x() {return x;}
    MON_TYPE& get_y() {return y;}
    MON_TYPE& get_z() {return z;}

    friend std::ostream& operator<<(std::ostream& flux,const vecteur_generique&
vec)
    {
        flux<<vec.get_x()<<","<<vec.get_y()<<","<<vec.get_z();
        return flux;
    }
};

//exemple de méthode templates à plusieurs arguments
//une matrice dont le type est générique et dont la taille est connue à la
```



```cpp
compilation
template <typename MON_TYPE,int TAILLE_X,int TAILLE_Y>
class matrice_generique
{
private:
    MON_TYPE data[TAILLE_X*TAILLE_Y];
public:

    matrice_generique()
    {
        for(unsigned int k=0;k<TAILLE_X*TAILLE_Y;++k)
            data[k]=0;
    }

    //definition de l'operateur parenthese pour acceder a l'element
    MON_TYPE& operator()(unsigned int kx,unsigned int ky)
    {
        if(kx<TAILLE_X && ky<TAILLE_Y)
            return data[kx+TAILLE_X*ky];
        else
            throw std::exception();
    }

    //dans le cas d'un contexte constant (exemple pour la multiplication)
    const MON_TYPE& operator() (unsigned int kx,unsigned int ky) const
    {
        if(kx<TAILLE_X && ky<TAILLE_Y)
            return data[kx+TAILLE_X*ky];
        else
            throw std::exception();
    }

    //mutiplication entre matrices
    template <int TAILLE_Y2>
    matrice_generique<MON_TYPE,TAILLE_X,TAILLE_Y2> multiply(const
matrice_generique<MON_TYPE,TAILLE_Y,TAILLE_Y2>& M)
    {
        matrice_generique<MON_TYPE,TAILLE_X,TAILLE_Y2> matrice_resultat;

        for(unsigned int kx=0;kx<TAILLE_X;++kx)
            for(unsigned int ky=0;ky<TAILLE_Y2;++ky)
                for(unsigned int k=0;k<TAILLE_Y;++k)
                    matrice_resultat(kx,ky) += (*this)(kx,k)*M(k,ky);
        return matrice_resultat;
    }

    friend std::ostream& operator<<(std::ostream& flux,const matrice_generique&
M)
    {
        for(unsigned int k=0;k<TAILLE_X;++k)
        {
            flux<<"(";
            for(unsigned int k2=0;k2<TAILLE_Y;++k2)
            {
                flux<<M(k,k2)<<" ";
            }
            flux<<")"<<std::endl;
        }
        return flux;
    }
};
```



```cpp
//un exemple de nombre complexe
class nombre_complex
{
private:
    double x;
    double y;
public:

    nombre_complex():x(0.0),y(0.0){}
    nombre_complex(double _x):x(_x),y(0.0){}
    nombre_complex(double _x,double _y):x(_x),y(_y){}

    double get_x() const{return x;}
    double get_y() const{return y;}

    nombre_complex& operator+=(const nombre_complex& z)
    {x+=z.get_x();y+=z.get_y();return *this;}
    nombre_complex& operator*=(const nombre_complex& z)
    {
        double temp_x=x*z.get_x()-y*z.get_y();
        double temp_y=x*z.get_y()+y*z.get_x();
        x=temp_x;y=temp_y;
        return *this;
    }

    friend nombre_complex operator+(const nombre_complex& z1,const
nombre_complex& z2)
    {nombre_complex z=z1;z+=z2;return z;}
    friend nombre_complex operator*(const nombre_complex& z1,const
nombre_complex& z2)
    {nombre_complex z=z1;z*=z2;return z;}

    friend std::ostream& operator<<(std::ostream& flux,const nombre_complex& z)
    {flux<<z.get_x()<<"+"<<z.get_y()<<"i";return flux;}

};

//utilisation
int main(int argc,char *argv[])
{

    // ************************************ //
    // Polymorphisme par virtual //
    // ************************************ //


    //creation d'objets specifiques
    rectangle *r0=new rectangle("mon rectangle 1",0,0,1,1);
    rectangle *r1=new rectangle("mon rectangle 2",2,2,5,5);
    disque *d0=new disque("mon disque a moi",0,0,1);
    disque *d1=new disque("mon deuxieme disque",5,5,6);


    //creation d'un conteneur generique
    std::list<objet_geometrique*> mon_conteneur;
    mon_conteneur.push_back(r0);
    mon_conteneur.push_back(d0);
    mon_conteneur.push_back(d1);
    mon_conteneur.push_back(r1);

    //affichage spécifique (notez que chaque objet se "souviens" de son type
propre)
```

```
    for(std::list<objet_geometrique*>::const_iterator
it=mon_conteneur.begin();it!=mon_conteneur.end();++it)
        (*it)->affiche();



    //On notera que le polymorphisme nécessite de stocker des pointeurs ou des
références.
    //si on avait std::list<objet_geometrique> mon_conteneur, le virtual ne
fonctionne plus!
    //ex.
    std::vector<objet_geometrique> mon_conteneur_marche_pas;
    mon_conteneur_marche_pas.push_back(*r0);
    mon_conteneur_marche_pas[0].affiche();//fonctionne pas

    objet_geometrique& ma_reference_a_moi=*r0;
    ma_reference_a_moi.affiche();//fonctionne (par contre on ne peut pas stocker
de reference dans un vecteur)

    //on efface la memoire car on a cree des pointeurs (on peut passer par le
tableau)
    for(std::list<objet_geometrique*>::const_iterator
it=mon_conteneur.begin();it!=mon_conteneur.end();++it)
        delete *it;

    //notons que sans le virtual, on aurait du definir un tableau pour chaque
type:
    // std::list<rectangle> liste_de_rectangle;
    // std::list<disque> liste_de_disque;
    // ce qui n'est pas gerable pour un grand nombre d'objets differents.




    // ************************************ //
    // Utilisation de templates
    // ************************************ //

    //divers types géré sans duplication de code
    vecteur_generique<double> mon_vecteur_de_double(4.5,5,8);
    vecteur_generique<float> mon_vecteur_de_float(4.5,5,8);
    vecteur_generique<unsigned int> mon_vecteur_de_uint(4.5,5,8);
    std::cout<<mon_vecteur_de_double<<std::endl;
    std::cout<<mon_vecteur_de_float<<std::endl;
    std::cout<<mon_vecteur_de_uint<<std::endl;

    //mais on peut egalement s'en servir avec ses propres types plus complexes
    vecteur_generique<vecteur_generique<double> > mon_vecteur_de_vecteur;
    mon_vecteur_de_vecteur.get_x()=vecteur_generique<double>(1,5,8);
    mon_vecteur_de_vecteur.get_y()=vecteur_generique<double>(7,8,9);
    mon_vecteur_de_vecteur.get_z()=vecteur_generique<double>(-1.5,0,12.1);
    std::cout<<mon_vecteur_de_vecteur<<std::endl;


    //exemple d'utilisation de la matrice de taille fixée à la compilation
    matrice_generique<nombre_complex,3,2> M1; //matrice de nombre complex de
taille (3,2)
    //on pourrait écrire de manière identique des matrices de doubles ou de int:
matrice_generique<double,...> matrice_generique<int,...> ...
    M1(0,0)=nombre_complex(0.5,1);M1(0,1)=nombre_complex(1,1);
    M1(1,0)=nombre_complex(-0.5,0);M1(1,1)=nombre_complex(2,-0.5);
```



```
    M1(2,0)=nombre_complex(-0.5,1.5);M1(2,1)=nombre_complex(-0.5,3);

    matrice_generique<nombre_complex,2,3> M2;
    for(unsigned int kx=0;kx<2;++kx)
        for(unsigned int ky=0;ky<3;++ky)
            M2(kx,ky)=nombre_complex(kx+ky+1,kx-ky+2);

    std::cout<<std::endl;
    matrice_generique<nombre_complex,3,3> M=M1.multiply(M2);//multiplication de
matrices
    //Notez que si la compilation réussi, on est assuré que les tailles sont
compatibles
    // Il n'y a plus besoin de le vérifier pendant l'execution du programme.
    //ex. matrice_generique<double,3,4> A;A*A ne compilera pas!

    std::cout<<M1;
    std::cout<<"*"<<std::endl;
    std::cout<<M2;
    std::cout<<"="<<std::endl;
    std::cout<<M<<std::endl;


    return 0;
}
```