```
+--------------------------------------------------------------------------+
|                              RESUME C++                                  |
|       Janvier                                                 2005       |
|                                                                          |
| wildcat                                            <wildcat@espix.org>   |
|                                                    http://www.espix.org  |
+--------------------------------------------------------------------------+
```

Ces questions sont la pour vous aider à répondre vous même.. bien sur il serais facile de les copier... ma foi vous pouvez pour ce que je m'en fou :) toutefois, il y a un grand manque a gagner si vous n'y répondez pas par vous meme.

Du reste je ne me suis pas relu donc il y a _certainement_ des imperfections, erreurs, des trucs que vous n'auriez pas fait pareil bref vous pouvez toujours m'envoyer le feedback par mail...

```
+--------------------------------------------------------------------------+
|                              QUESTION 01                                 |
+--------------------------------------------------------------------------+
```

Dans un programme à orientation graphique, on souhaite gérer au moyen des exceptions les erreurs suivantes :
- fenêtre principale invalide
- fenêtre non minimisable
- fenêtre non maximalisable
- fenêtre non réductible en icône
- graphiques impossibles à réaliser sur cette fenêtre

Imaginer la hiérarchie d'exceptions correspondante et écrire le handler d'exception correct correspondant.
Il faut éventuellement pouvoir gérer par une seule exception des erreurs de types similaires.

```
class Exception
{
        protected:
                string _error;
        public:
                Exception(const string &e = "Unknown error") : _error(e)
                {
                };
                ~Exception()
                {
                };
                string Erreur(void)
                {
                        return _error;
                };
};

class ExWindowSize : public Exception
{
        public:
                ExWindowSize(const string &e="Erreur, Taille de la fenetre inchangeable") : Ex
                {
                };
};
```

```
class ExWindowInvalid : public Exception
{
        public:
                ExWindowInvalid(const string &e="Erreur, Fenetre principale invalide.") : Exce
                {
                };
};


class ExNonMin : public ExWindowSize
{
        public:
                ExNonMin(const string &e="Erreur, Fenetre non minimisable.") : Exception(e)
                {
                };
};

class ExNonMax : public ExWindowSize
{
        public:
                ExNonMin(const string &e="Erreur, Fenetre non maximisable.") : Exception(e)
                {
                };
};


class ExNonIcon : public ExWindowSize
{
        public:
                ExNonMin(const string &e="Erreur, Fenetre non réductible en icône.") : Excepti
                {
                };
};

class ExNonGraph : public Exception
{
        public:
                ExNonMin() : Exception("Erreur, Graphiques impossibles à réaliser sur cette fe
                {
                };
};
```

Handler d’exception:

```
try
{
 // code a tester
}
catch (ExWindowSize &e)
{
        cout << e.Erreur() << endl;
}
catch (ExNonMax &e)
{
        cout << e.Erreur() << endl;
}
catch (Exception &e)
{
        cout << e.Erreur() << endl;
```

```
}
```

```
+-------------------------------------------------------------------------+
|                              QUESTION 02                                |
+-------------------------------------------------------------------------+
```

Dans un processus d'héritage, décrire précisément deux situations dans lesquelles un programmeur décidera de rendre virtuelle une méthode. Comment cette "méthode virtuelle" est-elle traitée par le compilateu ?

Décrire le mécanisme interne correspondant mis en place.

```
class TWagon
{
        protected:
                string Numero;
                string Description;
        public:
                virtual void affiche(void)
                {
                        cout << "Numero:        " << Numero << endl;
                        cout << "Description:   " << Description << endl;
                };
};

class TWagonVoy : public TWagon
{
        protected:
                int nbrvoy;
                int nbrvoymax;

        public:
                void affiche(void)
                {
                        cout << "Numero:        " << Numero << endl;
                        cout << "Description:   " << Description << endl;
                        cout << "Nombre voy:    " << nbrvoy << endl;
                        cout << "Nombre voy max:" << nbrvoymax << endl;
                };
};
```

Dans cet exemple, on décide de rendre "void affiche(void)" virtuelle, cela permet au programme lorsque l'on demande l'affichage des information d'un train, d'afficher non seulement les information génériques, mais aussi les informations supplémentaires (Conducteur dans ce cas).

L'appel a cette fonction est résolu, non pas a la compilation tel une méthode classique, mais a l'éxécution du programme. (dynamic binding).

Deuxieme exemple:

Ajoutons dans cette classe (TWagon) une méthode de modification de chargement:

```
        void setChargement(void)
        {
                affiche();
                modifie();
```

```
        }
```

Pour peu qu'affiche et modifie ont été déclarées en tant que méthodes virtuelles, et ont étés définie dans les classes filles, lors de l'appel de setChargement, la saisies des éléments des classes filles sera effectué automatiquement selon l'objet fourni.

Mécanisme de méthodes virtuelle:

Dès qu'une méthode est déclarée virtuelle, un pointeur (VPTR) est ajouté a la classe pointant sur une VTABLE commune aux instance de cette classe, la résolution dynamique se fesant en trois étapes:

- Localisation du VPTR correspondant a la méthode
- Localisation de la VTABLE grace a ce VPTR
- Localisation de la méthode a appeller grace au pointeur de fonction contenu dans cette vtable.

Il est a noter que ce mécanisme ne se met en place que lorsque les objets sont passés par référence, en effet, par valeur, le type de l'objet est déjà connu et donc la méthode a appeller aussi.

QUESTION 03

Décrire la classe ios et expliquer l'utilisation de ses divers membres.
En quoi sa structure reflète-t-elle le fonctionnement général des flux à deux couches du C++ ?

La classe ios est une classe abstraite servant de classe de base a toutes les classes de flux.

La classe ios contient un objet de type 'streambuf', de cette facon, la connexion E/S est réellement établie.
Ce pointeur peut etre obtenu en appellant la méthode "streambuf *rdbuf(void)".

Cette classe permet également un controle de flux au moyen de différentes méthodes renvoyant true/false, elles symbolisent ensemble l'état du flux, lui meme représenté au moyen d'une variable privée. (state)
Les différentes fonctions permettant de connaitre l'état de cette variable sont: "int bad()", "int fail()", "int eof()", "int good()", "int rdstate()"

Cette derniere fonction (rdstate), renvoie une variable de type io_state, membre de ios.

L'operateur ! de la classe ios a été lui aussi surchargé, ce qui permet de tester simplement l'état d'un flux, par exemple comme ceci:

```
if (!cout) traitement;
```

Egalement la possibilité d'écrire le code suivant:

```
if (cout << "test") traitement;
```

Ios permet également différentes possibilités de formattage des données, ces dernieres sont contenues dans une variable de type enum.
Cette variable est modifiée/consultée au moyen de deux méthode de Ios:

```
long setf(long flags)
```

```
        long unsetf(long flags)
 La taille du champ de sortie peut également etre fixée/lue au moyen de :

        int width(int largeur)
        int width() const
 Le charactere de remplissage est quand a lui manipulé via :

        char fill(char c)
        char fill() const
 De meme la précision des réels:
        int precision (int p)
        int precision () const

 Il est a noter que cette classe ne connais aucunement la destination
des données, elle s'occuppe simplement de leur formattage.


+-------------------------------------------------------------------------+
|                              QUESTION 04                                |
+-------------------------------------------------------------------------+


 Décrire les surcharges classiques des opérateurs =, +, << et ++.
 Utiliser une classe NombreComplexe qui représente la notion de nombres
complexes de type a+ib pour les exemples.

class NombreComplexe
{
        float _real;
        float _complex;

        public:
        NombreComplexe(float r=0., float c=0.) : _real(r), _complex(c)
        {
        };

        NombreComplexe &operator=(const NombreComplexe &n)
        {
                _a = n.getReal();
                _b = n.getComplex();
                return *this;
        };

        NombreComplexe &operator+(const NombreComplexe &n1)
        {
                NombreComplexe temp(*this);
                temp._real += n2._real;
                temp._complex += n1._complex;
                return temp;
        };

        friend ostream &operator<<(ostream &s, const NombreComplexe &n)
        {
                s << "Reel: " << n.getReal() << " Complx: " << getComplex();
                return s;
        };

        NombreComplexe &operator++() //préfixé
        {
                temp._real++;
```

```
                return *this;
        };

        NombreComplexe &operator++(int) //postfixé
        {
                NombreComplexe temp(*this)
                _real++;
                return temp;
        };

        float getReal (void)const       { return _real; };
        float getComplex (void)const    { return _complex; };
        void setReal(float c)           { _real = r; };
        void setComplex(float c)        { _complex = c; };

};
```

L'operateur + s'occuppe d'additionner les deux membres passé en parametres et de retourner l'addition de ceux ci en temps que nouvel objet.

L'operateur << recois un objet de classe ostream, qu'il remplis par les différents membres et informations sur la classe avant de le retourner par référence (on modifie bien l'objet ici et non sa copie..)

L'operateur ++ a pour seul but d'incrémenter un membre défini de la classe il n'est donc pas nécessaire de lui passer un parametre, toutefois, un parametre peut etre utilisé pour spécifier le type d'operateur ++ (préfixe ou suffixe).

```
};
```

QUESTION 05

Que signifie polymorphisme ? Pourquoi est-il possible en C++ et pas en C ? Donner un exemple d'utilisation du polymorphisme.

Le C représente la signature d'une fonction sur son nom, ainsi, si l'on veut décrire en C deux fonction effectuant le meme traitement, mais avec des types de données différents, il nous faudra définir deux fonction avec des noms différents.
En C++, une toute autre méthode est utilisée: la signature des fonctions n'est pas fondée uniquement sur le nom de celle-ci, mais bien sur son nom + le(s) type(s) de son(ses) parametre(s).

Exemple :

Si une classe A, necessite plusieurs constructeurs :

```
 class A
 {
        public:

                A();
                A(string);
                A(const A &);
 };
```

La surcharge du constructeur de la classe A, constitue un bel exemple de polymorphisme

QUESTION 06

Quelle la différence entre une variable/fonction membre ou statique ?
Comment fait-on pour accéder à une fonction selon qu'elle soit membre ou statique ?
Une fonction membre peut-elle accéder à une variable membre/statique ?
Une fonction statique peut-elle accéder à une variable membre/statique ?
Dans quels cas est-il intéressant d'avoir une variable membre statique ?
A quel endroit s'initialise une variable selon qu'elle est membre ou statique ?

- Une fonction statique permet d'etre appellée sans la nécéssité d'une instanciation de la classe, a l'inverse d'une variable membre, qui elle nécéssite cette instanciation.

- Acceder a une fonction statique de la classe A:

```
A::GetCpt();
```

  Acceder a une fonction membre de la classe A, avec a comme instance de cette classe:

```
a.GetCpt();
```

- Une méthode membre peut acceder a toute les variables membres/statiques.

- Une méthode statique ne peut acceder qu'a des membres statiques, étant donné qu'elle ne peut savoir sur quel instance de la classe elle doit agir. (instance peut etre inéxistant

- Il est interessant d'avoir une variable membre statique par exemple, si l'on désire compter le nombre d'instance d'une classe.
exemple:

On désire compter le nombre d'instance de la classe A

```
class A
{
        public:
                static int cpt;

                A() { A::cpt++; };
                ~A() { A::cpt--; };
};
```

En placant cette variable membre statique, il devient tres aisé de compter le nombre d'instance de cette classe. en ajoutant une incrémentation dans chaque constructeur, et une décrémentation dans le destructeur, cette tache deviens automatisée.
Le "lecteur attentif" aura conclut que la variable membre cpt n'est pas initialisée, il conviendra donc de le faire grâce a la ligne suivante:

```
A::cpt = 0;
```

QUESTION 07

Pourquoi le C++ introduit-il la notion de flux ?
Quels en sont les principes généraux de l'utilisation de ces flux dans un mécanisme d'E/S ?

Quelles sont, dans les grandes lignes, les hiérarchies qui sont utilisées et quels sont, brièvement, les rôles des principales classes ?

Les flux sont utilisés dans toute entrée/sortie, ils permettent une certaine souplesse d'utilisation grace aux operateur d'insertion et d'extraction, qui enlevent tout les problèmes liés au formatage des données, les conversions nécessaires étant automatiquement effectuées.

Les flux ajoute une couche représentant le flot de données entre source et destination.

ios, classe de base abstraite, commune a toutes les classe de flux, rassemble les caractéristiques communes aux flux formatés.

istream, ostream: fournissent les operation d'entrée (istream) et de sortie (ostream) sans buffer ni formattage particulier.
ces deux classes redéfinissent ensemble tout les operateurs << et >> pour les types de données standard C++.

iostream: né d'un héritage multiple, elle reprend simplement les éléments des deux classes précitées.

QUESTION 08

Expliquer et justifier le concept de container.
Décrire la structure hiérarchique que l'on peut classiquement mettre en place pour les classes containers classiques (classes abstraites, implémentations, etc).
Expliquer et justifier le concept d'itérateur.

Une classe container est une classe contenant une collection d'objet fournie avec les méthode d'ajout/suppression correspondant a cette collection.

Certains container ont la meme base (taille, insertion, extraction) meme si la définition de cette base differe, on peut donc creer une hierarchie de container ayant pour base une classe abstraite définisant l'interface commune des container.

Exemple de déclaration d'une hiérarchie de container (non entierement implémentée):

```
class Container
{
        protected:
                int *t;
                const int sz;
        public:
                Container() : sz(0), t(NULL) {};
                Container(int t) : sz(t) { t = new int [t]; };

                virtual ~Container() { delete t; };

                virtual int Taille() const { return sz; };

                virtual bool Insert(int) = 0;
                virtual bool Delete(int) = 0;
                virtual bool Affiche();
};
```

```
class Vecteur : public Container
{
        public:
                Vecteur(int n) : Container(n) {};
                ~Vecteur() { };

                void affiche(void)
                {
                        VecteurIterator it(*this);
                        cout << "Affichage du vecteur: " << endl;
                        while(!it.end())
                                cout << it++ << endl;
                }
                bool Insert(int);
                bool Delete(int);
};
```

Parfois, on doit pouvoir se balader dans les éléments d'un container selon un ordre défini. c'est la qu'interviennent les itérateurs.
Comme ceux ci doivent acceder aux données des container, il est souvent nécessaire de les déclarer en tant que classe amie de ceux ci, Ceci est effectué en ajoutant une ligne du type :

```
        friend class VecteurIterator;
```

a notre classe Vecteur.

L'itérateur devra au minimum posseder un operateur ++ permettant de se déplacer dans le container, ainsi qu'un operateur (int) qui permettra de renvoyer la valeur actuelle pointée. il est également possible de rendre ceci générique redéfinissant l'operateur unaire *.

```
class VecteurIterator
{
        private:
                Vecteur &v;
                int     *p;

        public:

                VecteurIterateur(Vecteur &i) : v(i), p(i.t)
                {
                };
                bool    end() const
                {
                        return ((p - v.t) >= v.sz)?true:false;
                };
                void reset()
                {
                        p = v.t;
                }
                bool operator++ ()
                {
                        if (!end())
                        {
                                p++;
                                return true;
                        }
                };
                void insert(int n)
```

```
                {
                        *p = n;
                };
                int operator int() const { return *p; };
};
```

+--+
| QUESTION 09 |
+--+

Expliquer les deux méthodes classiques d'accès à un fichier en C++.
Expliquer à quoi servent les différents types d'objets intervenants et comment font-ils pour remplir leurs fonctions ?
En quoi ces deux méthodes sont-elles comparables/différentes ?

Methode 1: filebuf && (i|o)stream

```
filebuf f("test.dat", ios::out);
ostream s(&f);

s << "Ecriture sur le fichier !" << endl;

f.close();
```

Explication:

Une classe de type filebuf effectue l'interface entre le flux et le fichier, mais n'est pas utilisable sans un (i|o)stream lui étant attaché.

Cette méthode nous montre très bien que les iostream ne s'occupe pas de la destination des données et qu'ils ont pour unique but de les formatter.

Methode 2: (i|o)fstream

```
fstream f("test.dat", ios::out);

f << "Ecriture sur le fichier !" << endl;

f.close();
```

Explication:

Cette méthode, a l'inverse de la premiere, est directement prévue pour une utilisation des fichier...
L'objet instancié de type (i|o)fstream contient en effet un objet de type filebufn pouvant etre accédée via rdbuf() (C.F. ios).

A noter qu'il est toutefois possible, comme dans la premiere méthode, de redéfinir le flux de sortie (fstream) via la méthode rdbuf

La méthode close est ici une méthode de la classe fstream redéfinie dans fstreambase.

+--+
| QUESTION 10 |
+--+

Ecrire les deux classes personnelHopital (un nom de type char* et un matricule de type char[]) et Medecin (héritée de personnelHopital, avec une spécialité en plus de type char*) en écrivant tous les constructeurs et destructeurs utiles.
Illustrer leur utilisation dans une fonction main().

```
class personnelHopital
{
        protected:

                char *_nom;
                char _mat[10];

        public:

                personnelHopital(char *n="", char *m="PER123456")
                {
                        strncpy(_mat, m, 10); // prevent BOF
                        _nom = new char [strlen(n)];
                        strcpy(_nom, n);
                };

                personnelHopital(const personnelHopital &p)
                {
                        _nom = new char [strlen(p.getNom())];
                        strcpy(_nom, p.getNom());
                        strcpy(_mat, p.getMat());
                };
                virtual ~personnelHopital()
                {
                        delete _nom;
                };

                const char *getNom(void) const { return _nom; };
                const char *getMat(void) const { return _mat; };
                void setNom(char *n)
                {
                        if (_nom)
                                delete [] _nom;
                        _nom = new char [strlen(n)];
                        strcpy(_nom, n);
                };
                void setMat(char *m)
                {
                        strncpy(_mat, m, 10); // prevent BOF
                };
};


class Medecin : public personnelHopital
{
        protected:

                char *_spec;

        public:

                Medecin(char *n="", char *m="MED123456", char *s="Inconue") : personnelHopital
```

```
                {
                        _spec = new char [strlen(s)];
                        strcpy(_spec, s);
                };

                Medecin(const Medecin &p)
                {
                        _nom = new char [strlen(p.getNom())];
                        strcpy(_nom, p.getNom());
                        strcpy(_mat, p.getMat());
                        _spec = new char [strlen(p.getSpec())];
                        strcpy(_spec, p.getSpec());
                };
                ~Medecin() { delete _spec; };

                char *getSpec(void) const { return _spec; };
                void setSpec(char *s)
                {
                        if (_spec)
                                delete [] _spec;
                        _spec = new char [strlen(s)];
                        strcpy(_spec, s);
                };
};


int main(void)
{
        Medecin m1("Dupont par son pere, pondu par sa mere", "MED123456", "Chirurgie");
        Medecin m1b(m1);
        Medecin m2;
        personnelHopital p1("Georgette", "MEN123456");

        return 1;
}
```

```
+-------------------------------------------------------------------------+
|                              QUESTION 11                                |
+-------------------------------------------------------------------------+
```

Expliquer à quoi servent le constructeur de copie et l'opérateur =.
Quel est leur syntaxe ?
Justifier chaque partie de la syntaxe (const ou non, référence ou non, etc ¿).
Dans quel cas sont-ils invoqués ?
Illustrer par des ex.

constructeur de copie:

Il est utilisé dans tout passage par valeur et également
lorsqu'un objet est renvoyé par une fonction.

syntaxe (pour une classe A):

```
        A(const A &o)
        {
                // assignement des membres de o -> this
        };
```

Ce constructeur ayant pour but de définir la copie de l'objet
il est évident qu'on ne PEUT PAS passer l'objet a copier par valeur..
(boucle infinie :))
Le mot clé const est utilisé car on ne modifie aucunement l'objet
passé au ctor.
Une grande utilité de la redéfinition de ce constructeur, est lorsque
la classe contient des pointeurs vers d'autres objet, ce constructeur
peut spécifier au besoin, qu'il faut copier ces membres et non pas
simplement leur adresse.

operateur = :

Lorsqu'une classe ne contient que des membres statiques, tout va bien (humpf),
en revanche, si cette meme classe contient des pointeurs, il conviens lors
d'une égalité (a=b;), de copier également le contenu pointé par ceux ci et non
pas simplement leur adresse. c'est ici que la redéfinition de l'operateur =
entre en jeu !

syntaxe (pour une classe A):

```
A &operator=(const A &o)
{
        // assignement des membres de o -> this
        return *this;
};
```

L'importance du renvois de donnée de type A &, garde toute son importance
dans une operation comme : a = b = c;
En effet, si l'on laisse void operator=(const A &o), ca risque de moins bien
marcher :D
Le passage par référence intervien de la meme facon que son retour,
en effet, si l'on écris a = b = c;, le parametre est passé et retourné
par référence. La donnée toujours a droite de l'operateur n'est pas modifiée
et donc passée en const;

QUESTION 12

Qu'est-ce qu'une classe abstraite ?
Donner un exemple illustrant chacun de ses rôles.
Quel est l'intérêt de créer de telles classes ?
Qu'apportent de plus les méthodes virtuelles pures ?

- Une classe abstraite est une classe contenant au moins une de
ses fonctions membres (ou fonction membres héritée) déclarée virtuelle
pure (=0), de ce fait une classe abstraite n'est jamais instanciable.

- bien que cette classe ne peut etre instanciée, elle fourni une
base commune a ses classes filles, également on pourra utiliser
cette classe comme pointeur vers des objet dériver et faire
ainsi intervenir le polymorphisme d'objet.

```
class Wagon
{
                int poid;
        public:
                Wagon();
                virtual void affiche() const =0;
                virtual void saisie() = 0;
};
class WVoyageur
{
                int nb_voyageur;
        public
                WVoyageur()
                void affiche();
                void saisie();
};
```

Dans cet exemple, les méthodes virtuelles pures affiche et saisie n'ont pour but que d'afficher les éléments spécifiques a chaque classe dérivée, et nont l'entiereté des spécification. utilisé en parallele avec un operator<< et >>, elles définirons une gestion simple des flux d'entrée et sortie.

- Elles permettent de prévoir un traitement dans la classe de base sans pour autant savoir ce qu'il fais, jusqu'au moment de la définition d'une classe fille.

QUESTION 13

Quels sont les différents types de passage de paramètres possibles ?
Un passage par const référence équivaut à quoi ?
Quand utilise-t-on tel ou tel type de passage de paramètres (dans le cas d'un type de donnée utilisateur ou un type prédéfini).
Expliquer et justifier vos réponses.

- Par valeur / Référence / Adresse
  - valeur: Le compilateur génere une copie de l'élément passé et le transmet a la fonction. les changements effectués a l'intérieur de cette fonction ne s'appliquent pas a l'objet de la fonction appellante. Attention en cas de polymorphisme d'objet, cette méthode de passage d'argument rend ce procédé totalement inéficace.
  - Adresse: Le contenue de l'objet n'est cette fois ci pas passé a la fonction appellée, mais simplement son adresse. Ceci permet a la fonction appellée de modifier l'objet passé si nécessaire, également, en cas de polymorphisme d'objet cette méthode garde intact le principe des méthodes virtuelles.
  - Référence: Cette méthode allie le point fort des deux méthode précédente, en effet, l'adresse de l'objet est passé, mais la syntaxe utilisée est celle du passage par valeur.

- Il signifie que l'on ne modifie pas l'objet passé en paramètre

- Par Référence:

Utilisé dans le cas d'une fonction devant modifier les paramètres qui lui sont fournis, sans pour autant changer l'écriture du programme principal.

```
Exemple: prototype:     void xchange(int &, int &);
         appel:         xchange(x, y);
```

Egalement si l'argument passé est de grande taille mais n'est pas modifié, un passage par référence (avec un mot clé const) est préférable pour en éviter la copie.

Par Valeur: Si on ne modifie pas l'argument passé.

Par adresse: Si le type modifié dans la fonction est prédéfini, ainsi l'utilisateur peut se rendre compte que sa variable va etre modifiée.

```
void fonction (int v1, int &v2, int *v3)
{
     v1++; // portée uniquement dans la fonction
     v2 += v1; // modifie v2
     *v3 = 42; // "
}
```

QUESTION 14

Pourquoi les concepteurs du C++ ont-ils créé un nouveau mécanisme de gestion des erreurs ? Expliquer ce mécanisme des exceptions et la manière correcte de l'utiliser dans la programmat En particulier, qu'entend-on par "classe d'exception" ?

- le mécanisme classique du C concernant la gestion des erreurs n'est pas adapté tout les usage car l'erreur peut etre ignorée (meme si celle ci est de la plus haute importance) ainsi qu'un résultat numérique qui ne représente pas toujours correctement l'erreur encontré

- Ce mécanisme permet de gerer les erreurs se produisant en se branchant sur un point du programme pour réaliser sa terminaison "propre".

- Le branchement s'effectue en lancant (throw) un objet d'une classe determinée, on "Attrappe" (catch) cet objet via un gestionnaire (handler) qui a reconnu son type d'objet.

L'implémentation générale des exception est défini comme suit:

```
void fonction(void)
{
       if (traitement != ok)
              throw (new int(42))
}
```

dans ce cas ci, on se contente d'un int comme objet lancé.

la reception de l'exception se fait comme suit:

```
try
{
       fonction();
}
catch (int &e)
{
       // gerer l'erreur e représentée par un int
}
catch (...)
{
       // type d'erreur lancé inconu.
}
```

Ce mécanisme facilite grandement la gestion d'erreur, et, on pourrais facilement imaginer une hiérarchie d'exception pour traiter précisément les erreurs de n'importe quelle partie de notre programme, ayant comme méthode de résolution un traitement commun.

Il conviens de prévoir un traitement (try-catch) pour chaque possibilité de lancement d'exception, en effet, une exception lancée et non récupérée conduit a une terminaison du programme.

QUESTION 15

Qu'entend-on par encapsulation ?
Donner deux exemples illustrant l'intérêt de cette notion pour une classe simple et une classe container.
Où se situe l'encapsulation au niveau des itérateurs et des classes de flux ?

L'encapsulation représente la séparation entre la définition d'une classe (son interface), et l'implémentation de celle ci.
Elle permet donc de modifier la maniere dont est effectué un traitement sans toutefois modifier le programme utilisant ce traitement.

- Un(e) pile/file/vecteur implémenté via un tableau statique, peut tres bien etre transformé en liste/pile/vecteur dynamique sans changer d'une ligne les fonctions définissant son interface, en effet, que ce container soit dynamique ou statique, son interface reste inchangée (ajout/suppression/consultation), seul le traitement effectué dans cette interface (implémentation) change.

- Si un objet censé effectué un calcul sofistiqué et rébarbatif ayant comme entrée un nombre, et comme sortie ce nombre traité; Si un jour, un programmeur de génie trouve un algorithme bien plus éfficace que celui actuellement mis en place, il lui sera tres facile de modifier le traitement du nombre d'entrée sans toutefois changer la déclaration. (les éléments d'entrée/sorties restent inchangés, seule la manière d'effectuer le traitement change)

Les flux sont basé sur un certain nombre de classe *stream, elles memes utilisent un streambuffer, c'est ce dernier qui réalise l'encapsulation.
Il fourni une interface inchangée selon le buffer (lecture/ecriture d'un carac), mais la source et la destination dépend du type de streambuffer utilisé.
L'implémentation est donc dissimulée et il est facile de changer la destination de ce flux sans changer le code utilisateur.

QUESTION 16

Décrire et expliquer les mécanismes d'upcasting et downcasting.
Pourquoi le downcasting est-il souvent utilisé dans le cas
de l'utilisation d'une librairie C++ ?
Comment réaliser un downcasting sûr en parcourant un vecteurs d'objets hétérogènes ?
Expliquer au moyen d'un exemple.

Soit deux classes:

```
class A
{
};

class B : public A
{
        void f();
};
```

UpCasting: Consiste a assigner un ptr de classe dérivée dans un ptr de classe de base.

```
        A *ptr = new B; // UpCast implicite
        A *ptr = (A *) new B; // UpCast explicite
```

DownCasting: Consiste a caster un ptr de classe de base en un ptr de classe dérivée.

```
        ptr->f(); pose probleme !!
        ((B *)ptr)->f(); fonctionne, mais il faut etre sur du type de ptr..
```

Downcasting sûr:

```
        B *ptrb;
        if ((ptrb = dynamic_cast<B *> (ptr)) == 0)
                cout << "Casting error" << endl;
        else
                ptrb->f();
```

Le downcasting est souvent utilisé lors d'une librairie C++ car, le C++
étant fondé sur le principe des hiérarchies de classe, il est souvent nécessaire
d'atteindre une fonction d'une portée différente, si cette fonction appartient
a une classe fille, le downcasting intervient donc !

QUESTION 17

Quelle est l'utilité de l'opérateur de résolution de portée.
Illustrer par deux exemples.

L'operateur :: permet d'indiquer le contexte d'une variable

ou fonction si celle ci n'est pas dans le contexte courant ou implicite.
 Il permet également lors de l'appel d'une méthode statique.

Exemple:

 - int i = Compte::Instance();
 - class A
   {
        int i;
        public:
                void SetXX(int);
   };

   void A::SetXX(int var) { i=var; }

   SetXX étant dans le contexte de A,
  et donc, absent du contexte courant;

   Si l'on avais omis l'operateur de résolution de portée dans
  cet exemple, on aurais tout simplement redéfini une fonction
  SetXX, provoquant également une erreur (i n'existe que dans
  le contexte de A::).

+---+
| QUESTION 18 |
+---+

 Qu'entend-on par classe générique ?
 Comment sont-elles implémentées dans le C++ moderne et quels sont les éléments
du langage qui ont permis cette généricité ?
 Quelle est la principale difficulté rencontrée pour l'instanciation
et de quelles manières peut-elle être résolue ?

 Une classe générique est en fait générée au niveau du préprocesseur,
une grande macro est définie, rédigeant le code au cas par cas,
suivant le type de donnée a remplacer.
 Elle définit la classe en y impliquant ce parametre.

 La principale difficulté est qu'on ne peut définir plusieures classes génériques
dans le meme programme, étant donné que la macro défini a chaque fois le meme nom
de classe... une solution pour contourner ce probleme consiste a utiliser la macro
name2, définie dans generic.h.

 Il est a noté que dans le C++ moderne, la méthode de classe générique a été
abandonnée au profit des templates.

Exemple :

```
/* macro de définition du nom de la classe */
#define Vecteur(type) name2(type, Vecteur)

/* macro de definition d'une instance */
#define DVecteur(var, type) Vecteur(type) var

#define Vecteurdeclare(type)                                            \
```

```
class Vecteur                                                                   \
{                                                                               \
        private:                                                                \
                type *data;                                                     \
                int  sz;                                                        \
        public:                                                                 \
                Vecteur(type) (int n)                                           \
                {                                                               \
                        data = new type [n];                                    \
                        sz = n;                                                 \
                };                                                              \
                ~ Vecteur(type) (void) { delete data; };                        \
};
```

pour declarer un vecteur de type int:

```
declare(Vecteur, int)
```

pour instancier un vecteur de 10 int nommé 'v':

```
DVecteur(v(10), int);
```

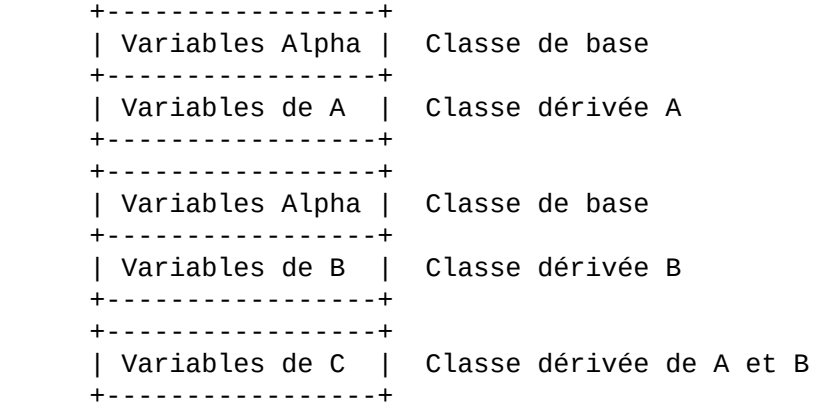
QUESTION 19

Qu'est ce qu'un héritage virtuel ?
Quand doit-on procéder à un tel héritage et pourquoi ?
Lorsqu'un héritage a été réalisé de manière virtuel, qu'est ce que cela engendre pour le reste de la hiérarchie ?
Illustrer par un exemple.

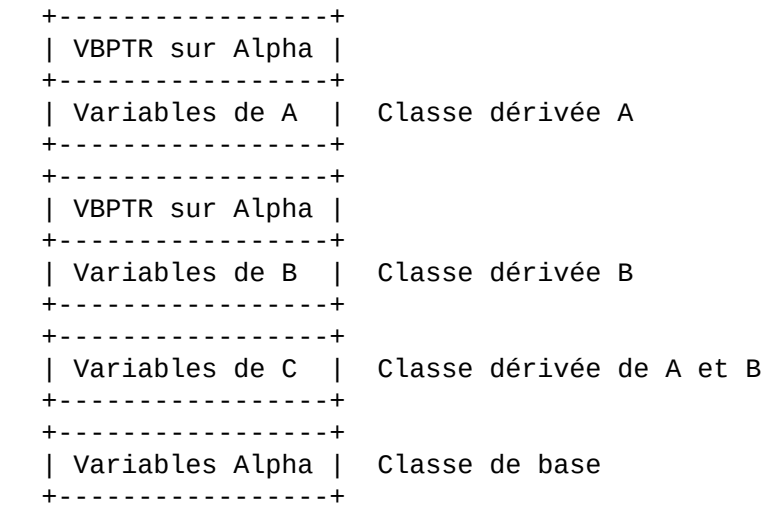
Soit, une classe A dérivant d'une classe Apha
une classe B dérivant également de cette classe Alpha

une classe C dérivant de A et de B.

La réprésentation d'une instance de cette classe C, suivrais la figure suivante:



Cela n'étant pas tres économique, on propose donc un héritage virtuel, permettant une fois appliqué a l'exemple ci dessu, d'obtenir cette représentation:



La transcription de ceci en code C++ donnerais :

```
class Alpha
{
};

class A : public virtual Alpha
{
};

class B : public virtual Alpha
{
};

class C : public virtual A, public virtual B
{
};
```

QUESTION 20

Pourquoi une méthode virtuelle inline n'a pas de sens ?
Quel est l'avantage des fonctions inline ?
Quand est-il intéressant de déclarer une méthode en inline ?
Un constructeur peut-il être inline ?
Et virtuel ?
Justifier vos réponses.

Une fonction inline est une méthode de remplacement des macro du C, consistant a recopier le code "inliné" a chaque endroit ou celui ci est nécessaire (appellé), le compilateur n'effectue donc pas d'appel dans le cas d'une fonction inline.

Une fonction virtuelle a quand a elle une adresse stockée dans une VTABLE, une fonction virtuelle inline n'a donc aucun sens, puisque cette fonction inline ne possede pas d'adresse.

Face a ce dilemne le compilateur copiera le code de la fonction inline a un seul endroit dans l'objet, la rendant donc non-inline.

L'avantage des fonctions inline, tout comme les macro du C, est de rendre l'éxécution du code de la fonction plus rapide en n'effectuant aucun appel. toutefois, un autre avantage par rapport a leurs homoloques du C, est qu'une fonction inline peut utiliser des variables locales.

Cette méthode n'est avantageuse que pour de petite fonction, lors de grande fonction, la perte de mémoire due au recopiage du code dépasse grandement le gain du a l'appel lors d'une fonction classique.

Un constructeur peut tres bien etre inline, meme si cette méthode est déconseillée si l'initialisation des variables membres va effectuer beaucoup d'operation (voir raison ci dessus).

Un constructeur ne peut etre virtuel d'un point de vue technique, en effet, une fonction virtuelle nécéssite l'éxistance au préalable d'une VTABLE, toutefois, un des roles du constructeur est de créer et d'initialiser le VPTR qui référence cette meme VTABLE dans l'objet.

QUESTION 21

Décrire et expliquer toutes les surcharges de l'opérateur << dans la librairie standard du C++.
En particulier, que sont les manipulateurs ?
En existe-t-il dans la librairie C++ standard ?

- Surcharges simples de types prédéfinis, char, short, int, long, char*, float, double, long double, unsigned ... et le pointeur void*. Ces operations reproduisent les fonctionnalités du printf en C.

- L'insertion du contenu d'un streambuf*

- Les manipulateurs: il est possible d'insérer dans un ostream, l'adresse d'une fonction prenant comme parametre une référence vers un ostream et retourne ce meme ostream. (ostream &fonction(ostream&)). Les manipulateurs standards C++ sont par exemple endl et flush. Il en éxiste également dans ios, pour gérer le formatage des données (dec, hex, oct).

La gestion des types précités est effectuée par le biais de prototypes de fonction membre d'ostream, de la maniere suivante:

```
ostream& operator<< (type);
```

Ou type représentant le type de donnée en question.

QUESTION 22

Une application contient des objets de types suivant :

- Mécanicien
- Client
- Voiture
- Prix
- Carnet entretien

Créer les classes correspondantes en illustrant leurs "liens" en sachant que :

Un mécanicien et un Client ont des caractéristiques communes
Une voiture est associé à un client et un client est associé à une voiture.
L'un pouvant exister sans l'autre.
Le carnet d'entretien est toujours associé à une voiture.
Une voiture possède une méthode permettant de calculer le prix d'un entretien à partir d'informations contenue dans le carnet d'entretien.

- le mécanicien et le client ayant des caractéristiques communes, on modélisera celles ci grace a une classe personne

- La voiture étant associée a un client et inversément, toutefois l'un peut exister sans l'autre, on préferera donc un lien par pointeur de l'un vers l'autre. On prendra également garde lors de la déstruction d'un des deux (client/voiture) de le déréférencer dans l'autre.

- Le carnet étant unique par voiture, et l'acompagnant tout on long de sa durée de vie, une contenance de celui ci par valeur est a préférer.

```
class Personne
{
                string _nom;
        public:
                Personne(const string &n="") : _nom(n) {};

                // interface de set/get pour le nom.

};


class Mecanicien : public Personne
{
                // membres propres a mécanicien
        public:
                Mecanicien(const string &n="") : Personne(n) {};

};

class Voiture;  // la déclaration de voiture va suivre
                // mais est utilisée dans Client.

class Client : public Personne
{
        private:
                Voiture *_voiture;
```

```
        public:
                Client(Voiture *v) : _voiture(v);
                ~Client() { if (_voiture) { _voiture->setClient(NULL); };

                bool setVoiture(Voiture *v) { _voiture = v; };
                Voiture *getVoiture()const { return _voiture; };

};

class Carnet
{
        public:
                Carnet();
};



class Voiture
{
        private:
                Client *_client;
                Carnet _carnet;

        public:
                Voiture(Client *c) : _client(c) {};
                ~Voiture() { if (_client) _client->setVoiture(NULL); };

                Prix getPrixEntretien();

                Client *getClient()const { return _client; };
                bool    setClient(Client *c) { _client = c; };

};


class Prix
{
        public:
                Prix();
};
```

```
+-------------------------------------------------------------------------+
|                              QUESTION 23                                |
+-------------------------------------------------------------------------+
```

Expliquer toutes les potentialités des objets cin et cout, en expliquant l'origine des méthodes et opérateurs évoquées.

- cin et cout sont des instances respectives de istream_withassign et ostream_withassign.
  Comme leur nom l'indique, ces deux classes permettent de reassigner un stream a un autre.

- Ces deux classes *stream_withassigne dérivant de *stream, possedent un grand nombre d'operateur << et >> permettant respectivement d'insérer ou d'extraire des données de types prédéfinis, et également l'utilisation de manipulateurs. Elles permettent également via certaines méthodes la modification du streambuf associé. Elles héritent toutes deux de la classe ios.

- La classe ios implémente les fonctionnalités communes a tout les flux.

+--+
| QUESTION 24 |
+--+

Que se passe-t-il lorsqu'on retourne dans une fonction une variable locale et que la fonction a pour type de retour une référence ?
Que faudrait-il faire pour solutionner ce problème ?
Donner un exemple.

Mauvais code:

```
A &fonction(void)
{
        A test;
        return &test;
}
```

La variable test étant locale a la fonction(), des que celle ci aura franchis le return, test sera considéré comme détruit (libéré).
Si l'on utilise cette variable retournée dans le programme, l'effet serais plus qu'imprévu..

Bon code:

```
A &fonction(void)
{
        A *test = new A;
        return *test;
}
```

Pour solutionner ce probleme, il "suffit" d'allouer une variable et de retourner celle ci par référence.

Cet espace alloué n'étant pas supprimé explicitement avant le return, il ne le sera donc pas et pourra etre utilisé au retour de la fonction.
Attention toutefois a veiller a supprimer cet espace dans le programme appellant la fonction()...

Une autre solution consiste a effectuer un retour par valeur, on est ainsi certain d'obtenir notre résultat au retour de la fonction sans devoir se soucier de sa libération.

Bon code (bis):

```
A fonction(void)
{
        A test;
        return test;
}
```

+--+
| QUESTION 25 |
+--+

On dispose de deux classes
Soldat, comportant notamment une méthode affiche();
Officier, dérivée de Soldat, comportant notamment une variable membre supplémentaire FonctionDeCommandement, une méthode affiche() et une méthode getFonctionDeCommandement().
Expliquer comment l'on peut utiliser un tableau hétérogène de Soldat et Officier de manière à afficher correctement chaque élément et à pouvoir appeler la méthode getFonctionDeCommandement() lorsque c'est possible.

Ecrire la fonction main() qui réalise le test.

```
class Soldat
{
        public:
                Soldat()
                {
                };
                virtual ~Soldat() {};

                virtual void affiche(void)
                {
                        cout << "Affichage des données SOLDAT" << endl;
                };

};


class Officier : public Soldat
{
        string FonctionDeCommandement;

        public:

                Officier(const string &f="Total !") : FonctionDeCommandement(f)
                {
                };
                ~Officier() {};

                string getFonctionDeCommandement(void)
                {
                        return FonctionDeCommandement;
                };

                void affiche(void)
                {
                        cout << "Données génériques Officier" << endl;
                        cout << "\tFonction : " << getFonctionDeCommandement() << endl;
                };
};
```

Un tableau étais mono-type, on déclare un tableau de pointeur utilisant la classe de base (polymorphisme d'objet), lors du traitement de ce tableau si l'on désire afficher la fonction de commandement d'un officier si l'objet courant est un officier, on utilise le principe de downcasting fourni par le RTTI (dynamic_cast).
Il est également a noté que le destructeur de soldat sera rendu virtuel, pour permetre lors de la suppression d'un officier, d'appeller le destructeur

de celui ci avant celui de la classe de base.

```
int main (void)
{
        Soldat s1,s2,s3;
        Officier o1;

        Soldat *vec[] = { &o1, &s1, &s2, &s3 };

        for (int i=0; i < 4; i++)
        {
                vec[i]->affiche();
                if (Officier *o = dynamic_cast<Officier*>(tab[i]))
                        cout << o->getFonctionDeCommandement() << endl; // on peut appeller ce
        }

        return 42;
}
```

QUESTION 26

Commenter cette affirmation : "Dans une classe, le gros intérêt de l'encapsulation est d'interdire l'accès depuis l'extérieur aux variables membres privées et donc de protéger celles-ci (notion de confidentialité)".

La véritable protection se situe au niveau de l'implémentation vue de l'extérieur, de cette facon, on peut modifier cette implémentation sans devoir modifier le code utilisateur. le seul code a modifier etant celui des fonctions constituant l'interface de ces membres privés.

La confidentialité des données est un bien grand mot, en effet, si ces données sont présente, c'est qu'elle doivent etre utilisées par l'utilisateur de cette classe (c'est leur raison d'etre !).

La vérification de l'accès a ces variable n'est effectué qu'a la compilation, on pourrais tres bien lire la portion de mémoire contenant la donnée "privée" pendant l'éxécution.

QUESTION 27

Lorsqu'on effectue la surcharge d'un opérateur en C++, combien de choix s'offrent au programmeur ?
Quelle est la différence entre ces deux choix ?
Dans quel(s) cas doit-on opter pour l'un ou pour l'autre ?
Dans quel(s) cas n'a t'on pas le choix ?
Comment le compilateur C++ traduit-il une instruction du style suivant :

```
        NombreComplexe c1(1,3),c2(4,-1),c3;

        c3 = c1 + c2;
```

- Deux choix s'offrent au programmeur d'une classe lors de la redéfinition d'un operateur, premierement, celui ci peut etre déclaré dans la classe (méthode membre), et prend dans ce cas comme argument l'objet sur lequel il est invoqué.
Deuxiemement, il est possible de redéfinir cet operateur a l'extérieur de cette classe, celui ci prendra dans ce cas, deux arguments, les membres de l'opération.

Voici un exemple de ces deux déclaration:

```
class NombreComplexe
{
        float _reel, _abstrai;
        public:
                NombreComplexe operator+(Nombrecomplexe b);
};
Nombrecomplexe operator+ (NombreComplexe a, NombreComplexe b);
```

Le c++ impose la premiere forme (méthode membre) lors de la redéfinition des operateurs =, () [], ->, new et delete.
Lors de la redéfinition d'un operateur utilisant une classe existante qui ne le prévoyais pas (ostream, istream), etant donné que l'on ne peut modifier les classes de la librairie standar a chaque déclaration d'une nouvelle classe utilisateur, nous sommes donc bel et bien forcé d'utiliser la deuxieme méthode (extérieur de la classe).

L'instruction c3 = c1 + c2, sera traduite par le compilateur c3 = c1.operator+ (c2);
ou bien c3 = operator+(c1, c2); suivant la déclaration utilisée dans l'exemple ci dessus.

QUESTION 28

Expliquer deux situations dans lesquelles il est nécessaire
de déclarer des classes sans les définir immédiatement à la suite de cette déclaration.

Lorsque deux classe sont référencées l'une envers l'autre, il est nécessaire d'en
déclarer une avant l'autre, pour pouvoir effectuer cette declaration,
il es nécessaire dire au compilateur que la classe utilisée sera déclarée ultérieurement.

```
class A;

class B
{
        A &a;
};

class A
{
        B &b;
};
```

Lors de projets imposant, il devient souvent nécessaire de séparer la déclaration des classes dans plusieurs fichier d'en-tetes. Une méthode consiste a forwarder la déclaration des classes utilisées uniquement comme pointeurs dans un fichier.

i.e.:

```
--- A.h ---

class B;

class A
{
        B *b;
        B &bb;
};

--- EOF(A.h) ---
```

Ici, on évite ainsi d'inclure une en-tete B.h dans laquelle se trouve la déclaration complete de la classe B.

```
+-------------------------------------------------------------------------+
|                              QUESTION 29                                |
+-------------------------------------------------------------------------+
```

Qu'apportent de plus les classes ostream_with_assign et istream_with_assign?
Quel en est l'usage classique ?

Ces classes sont des variantes des classes (i|o)stream, leur but est de rediriger un flux vers un autre.

Leur principale utilisation est "cin" et "cout" qui sont tout deux des instance de ces deux types de classes.

Exemple:

```
  ofstream f("file.dat", ios::out);
  cout = f;

  cout << "42" << endl;
```

```
+-------------------------------------------------------------------------+
|                              QUESTION 30                                |
+-------------------------------------------------------------------------+
```

Citer et expliquer les trois types d'accès à une variable/fonction membre qui existe en C++.
Faites de même pour les deux manières qu'une classe a d'hériter d'une autre classe.

```
private:        l'accès aux membres privés d'une classe est réservés exclusivement a cette cla
protected:      l'accès aux membres protégés d'une classe est réservé a cette classe ainsi qu'
public:         l'accès aux membres public d'une classe est accessible a tous.


class A
{
                int prive;
        public:
                int pub;
};
```

```
class B1 : public A
{

};

class B2 : private A
{

};
```

B1, hérite des membres de A, les membres de celle ci conserveront leur accès original.
B2, hérite des membres de A tout comme B1, a la différence pres que la portée de ces membres sera changée en private.

```
+-------------------------------------------------------------------------+
|                              QUESTION 31                                |
+-------------------------------------------------------------------------+
```

Je désire créer une classe abstraite "CFormeGeometrique" comportant :

- un nom

- une série de méthode faisant office d'interface de la classe et de TOUTES les classes hérit
- une méthode d'encodage de ses caractéristiques
- une méthode d'accès en lecture et en écriture de la variable "nom"
- une méthode d'affichage de ses caractéristiques
- une méthode de calcul de la superficie de la forme
- une méthode de calcul du périmètre de la forme

Certaines de ces méthodes doivent être redéfinies obligatoirement dans chaque type de forme g
Comment faire pour les obliger à le faire ?

Une méthode vérifiant que le nom est bien encodé suivant un format défini.
Ce format est le même pour toutes les formes géométriques.
Cette fonction sera utilisée uniquement dans la méthode d'encodage d'une forme.

Un compteur permettant de dénombrer le nombre d'objets géométrique définis dans l'applicatior

Donner et expliquer la définition de cette classe en associant à chaque élément le bon droit
Donner et expliquer la définition d'une classe "CCarre" hérité de la classe "CFormeGéométriqu et offrant en plus la possibilité de dénombrer le nombre de carrés créer dans l'application.

Créer une classe "CCarreSecurise" représentant un objet Carré à partir duquel l'utilisateur de la classe ne peut qu'afficher ses caractéristiques.

- Le mécanisme de comptage des instances devant pouvoir se retrouver non seulement dans la classe de base, mais également dans ses dérivées, et pour éviter un "bête" recopiage de code, nous résoudrons ce "problème" par un mécanisme maintenant bien connu, les templates.
  Pourquoi une template ? si nous avions déclaré et utilisé une classe simple (sans templates l'opération cpt++; aurait entrainé un warning a la compilation du type:

```
warning: direct base 'Compte' inaccessible in 'Carre' due to ambiguity
```

Ce warning prouve encore une fois la nécéssité de l'utilisation des templates.

- La classe CCarreSecurise "représente" un carré et donc CONTIENT un carré comme membre
 pour respecté la formule de l'énoncé, on définira une méthode affiche ayant pour but
 unique d'appeller la méthode carre.affiche();.

```
template <class T>
class Compte
{
        static unsigned int _cpt;

        public:
                Compte()                { _cpt++; };
                Compte(const Compte &)  { _cpt++; };
                ~Compte()               { _cpt--; };

                static unsigned int getNbr() { return _cpt; };
};

template <class T>
unsigned int Compte<T>::_cpt = 0;


class CFormeGeometrique : Compte<CFormeGeometrique>
{
        private:

                string Nom;                     // private car il existe une interface get/set
                                                //pour modifier cette variable

        protected:

                double     _peri;
                double     _aire;               // protected, les fonctions définisant ces
                                                //membres sont dans des classes filles

                bool    checkNom(const string &n);      // protected car les classes dérivées
                                                        //ont besoin dans la fonction de saisi

        public:
                CFormeGeometrique();
                CFormeGeometrique(const string &);
                CFormeGeometrique(const CFormeGeometrique &);
                virtual ~CFormeGeometrique();

                virtual void    Affiche (void) const;   //  virtual, la définition dans la cla
                                                        // de base affiche uniquement les élém
                virtual void    Saisie (void);          // leur redéfinition dans la classe dé
                                                        // n'est pas forcément nécessaire.
                                                        // public: ils doivent etre utilisable

                        string  getNom(void) const;     // interface set/get... const car ne m
                        void    setNom(string);

                        double  getAire(void)const;
                        double  getPeri(void)const;

                static  int     getCpt(void) { return Compte<CFormeGeometrique>::getNbr(); };

                virtual void    CalculeAire(void) = 0;
```

```
                virtual void    CalculePeri(void) = 0;


};

int CFormeGeometrique::cpt = 0;

class Carre : public CFormeGeometrique, public Compte<Carre>
{
        private:
                double _cote;


        public:

                void Saisie(void);
                void CalculeAire(void);
                void CalculePeri(void);

                CCarreSecurise();               // Ctor
                CCarreSecurise(double);         // Ctor
                CCarreSecurise(const CCareSecurise &);
                ~CCarreSecurise();

                void Affiche (void);            // Affiche...

                static  int     getCpt(void) { return Compte<Carre>::getNbr(); };
};

class CCarreSecurise
{
        private:
                Carre carre;
        public:
                CCarreSecurise(const Carre &c) : carre(c) {};
                void affiche() const { carre.affiche(); };      // seul affiche accessible
};

int main (void)
{
        Carre c1, c2, c3;
        CCarreSecurise cs1(c1);

        cout << "Nombre d'instance de Carre: " << Carre::getCpt() << endl;
        cout << "Nombre d'instance de FG:    " << CFormeGeometrique::getCpt() << endl;

        return 42;
}

+-------------------------------------------------------------------------+
|                               EOF                                       |
+-------------------------------------------------------------------------+
```