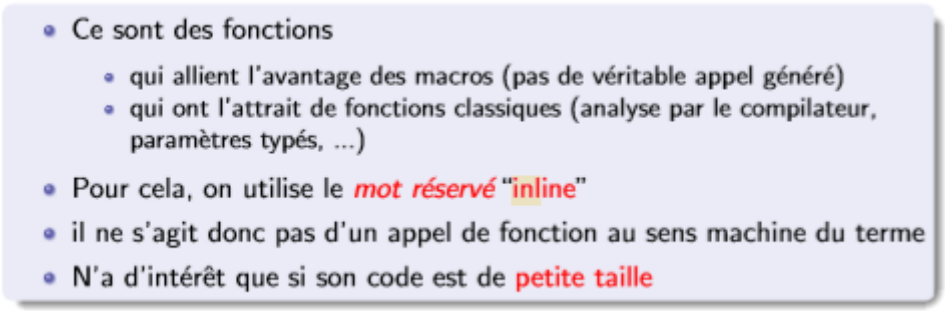
Theorie 1 :

Question 1) MACRO INLINe

en C les macro ne sont pas contrôlés par le compilateur, çela reste donc du texte que le pré-processeur va remplacer

en C++ les fonctions INLINe sont interpreté par le compilateur(donc trace des erreurs), de plus la fonction INLINe a tout les attrait d'une fonction classique C++ (encapsulation)



Question 2) MALLOC / NeW FRee / DeLeTe

Malloc et Free en C sont des FONCTIONS : Reserve de la place dans le "tas/heap" retourne un pointeur non typé void * Le HeAP ne peut pas allouer de l'espace sans init

New et Delete en C++ sont des OPeRATeURS : Reserve de la place en mémoire libre retourne un pointeur typé LA MeMOIRe LIBRe peut allouer de l'espace sans Init + APPeL DeSTRUCTeUR eT CONSTRUCTeUR

Sur une erreur Malloc renvoie NULL alors que new renvoie une exception (throw).

Comme new est un opérateur == SURCHARGe !

New appels les constructeurs !

La taille NeW alloué est calculer par le compilateur alors que MALLOC c'est à la main !

Question 3) expliquer le polymorphisme des fonctions et des opérateurs en C++.

Le POLYMORPHISMe est un ajout du C++ pour gerer des fonctions/methodes qui ont le même nom mais prennent des param differents. On appel ça la SIGNATURe de la méthode.

ATTeNTION : Le type retour ne fait en aucun cas parti de la signature (seulement les param).

C++ rend cela possible contrairement au C car il génère non seulement le nom des fonctions

utilisées, mais également des informations complémentaires sur les types des paramètres de

ces fonctions. On parle encore du "name mangling" des fonctions C+.

Question 4) expliquer en quoi une classe de la P.O.O. en C++ apporte de la modularité

Modularité : une classe se suffit à elle-même puisqu'un objet qui l'instancie contient ses données et ses traitements.

Abstraction : une classe devient un type de données analogue aux types

prédéfinis.

Les données sont encapsulées dans la classe et sont donc protégées contre les assauts d'autres fonctions non membres de la classe. elles sont invisibles pour le monde extérieur.

Les fonctions membres publiques sont utilisables pour tout le monde. Ceci permet la programmation front-end.

Question 5)

Chaque objet d'un même type possède partage le même code avec chaque objet de ce type.

Le compilateur générer un code qui retrouve l'objet auquel les méthodes évoquées s'appliquent. il passe un paramètre supplémentaire, et ce de manière implicite, à chaque fonction : c'est le pointeur this qui est un pointeur sur l'objet en question.

Theorie 2 :

Question 1) constructeur de copie

Une reference est un espece de pointeur qui à la difference d'un pointeur on a pas besoin de spécifié le & ce qui rend l'encapsulation possible

(car le programmeur avec pointeur ne sais pas forcement si ont veut l'adresse ou la valeur pointer par elle)

Le constructeur de copie fonctionne donc avec la reference d'un objet. Cette reference peut être modifier

Comme un pointeur une reference, reference donc une zone en mémoire,

Une référence doit être initialisée à sa déclaration et une référence ne peut jamais désigner une autre variable que celle de sa définition;

autrement dit, elle ne peut être modifiée ...

Remarques

1) Une référence ne peut être nulle.

2) Une référence sur un void (void &) n'est pas permise.

2) Les références prennent toute leur importance dans la surcharge des opérateurs

3) Un danger de l'utilisation de paramètres passés par référence est que l'on peut perdre de vue que ces paramètres peuvent être modifiés.

Lors de la création d'un objet à partir d'un autre du même type (appel

explicite)

Lors du passage par valeur d'un objet dans une fonction (appel implicite)

Lorsqu'un objet est renvoyé par valeur par une fonction (appel implicite)

Question 2) 2. expliquer ce qu'est un destructeur et comment il est appelé. Quel rapport avec l'opérateur delete [] (prendre un exemple) ?

le destructeur d'un objet est invoqué lorsque l'on quitte la portée de cet objet (objet

statique) ou lorsqu'il est explicitement détruit (objet dynamique);

il libère la mémoire occupée par l'objet.

Il peut aussi réaliser d'autres tâches, comme sauver les données dans un fichier puis

fermer celui-ci ou afficher des messages.

¬ Qui ne peut être surchargé : le destructeur d'une classe est unique

¬ Qui porte le même nom que la classe précédé du symbole « ~ »

¬ Qui n'a pas de type (même pas void)

¬ Qui ne peut avoir de paramètres

L'opérateur Delete [] fait appel au destructeur d'objet avant de désallouer la mémoire.

Question 3) expliquer et justifier les notions de variable membre statique et de méthode statique.

Pourquoi peut-on imaginer des variables membres statiques dans une classe qui sont des

instances de cette classe [à compléter cours 3]?

Il est cependant possible de définir une variable qui appartient à tous les objets instanciant la classe para. Il lui correspond alors un seul emplacement mémoire. Il s'agit d'une variable statique.

Déclaration d'une variable statique : dans le .H

Définition et initialisation de la variable : dans le .CPP

a variable statique est donc une sorte de variable globale

L'accès à la variable statique publique : on utilise l'opérateur de

résolution de portée « :: » avec le nom de la classe

- On peut utiliser cette variable statique avant même qu'un seul objet de

cette classe n'existe !

Qui dit variable membres statiques dit aussi méthodes membres statique qui vont avoir accès à cette variable statique.

en effet comment justifier qu'une variable membre statique existe avant toute instanciation d'objet si on peut pas y acceder avant d'instancier un objet ? NON SeNS ☺

Donc :

#Les méthodes statiques ne peuvent pas accéder aux variables membres non statiques !! Comme pour les variables statiques elles peuvent jouer un rôle même avant l'instanciation d'un objet !

Donc non sens si elles avaient accès aux variables membres classiques qui elles, sont instancié ! (Accéder a rien ?).

On peut utiliser une variable membre statique pour compter par exemple le nombre d'objet instancié de la classe qui la possède ! UN Bete Compteur QUOI ! PAR exemple.

Question 4°) expliquer et illustrer les différentes associations possibles entre deux classes [à compléteravec le cours 4]. en particulier, comment un objet d'une classe peut-il "envoyer un message" à un objet instance de l'autre classe ? [à compléter avec cours 3]

Dans une application, les classes ne sont pas totalement indépendantes les

unes des autres. Le plus souvent, elles coopèrent entre elles.

♥A°) AGReGATION PAR VALeUR :

On ajoute à la classe Compte une variable membre instanciant le Client : on

parle de relation de type « par valeur » (exemple Dimension dans ImageNG)

ImageNG instancie une variable membre d'un type d'objet venant d'une autre classe : type :Dimension variable :Dim

On peut donc appeler les methodes de la classe instancié : ImageNG.dim.Gethauteur();

Ou encore : Client client; (dans la classe Compte)

Cependant, l'objet encapsulé n'existe que pendant que l'objet contenant existe

également.

¬ Ceci pose problème dans le cas où un client n'a pas encore de compte.

Un objet client peut exister sans compte.

¬ Un seul et même client peut avoir plusieurs comptes

♥B°) AGReGATION PAR ReFeReNCe :

L'idée est qu'un objet Client peut être référencé par plusieurs objets Compte

différents. On ajoute à la classe Compte un pointeur désignant le client. On

parle alors de relation du type « par référence » ou encore de « relation

d'agrégation par référence ».

Client* pClient;

Cette fois l'objet pClient peut exister hors de la classe instanciant le pointeur.

♥C°) LeS UTILISATIONS :

```
Placement pl(capital,tauxA);
credit(capital);
credit(pl.interet(nbMois));
```

On peut remarquer que cette méthode créée une variable local pl, instance de Placement, dans le but d'utiliser la méthode interet(). On parle de « relation d'utilisation ». On entend par là que la classe Compte utilise un « service » de la classe Placement.

Theorie 3

Question 1°) 1. Qu'est-ce qu'un opérateur membre et un opérateur ami ? Dans quelles circonstances choisit-on l'un ou l'autre pour un opérateur donné ?

Un opérateur est vu en C++ comme une fonction particulière.

Pour cela, il faudrait pouvoir utiliser les opérateurs classiques (lorsque cela a un sens) comme l'addition « + », la soustraction « - », etc… sur nos nouveaux types de données.

Un operateur amie d'une classe possède les mêmes privilèges vis-à-vis de celle-ci que ses fonctions membres.

Comment déterminer si un opérateur doit être membre ou ami ?

¬ Un operateur membre assure que la première opérande est forcément un objet de la classe : aucune conversion n'est possible.

¬ Au contraire, un opérateur libre (ou ami) a pour première opérande un argument de fonction normal. Il peut donc être quelconque.

¬ Un opérateur libre est souvent utilisé lorsque les opérandes sont des objets de type différent, qui peuvent se présenter dans n'importe quel ordre.

DONC :

Un opérateur libre s'impose pour une classe qu'il n'est pas possible de modifier.

¬ Mais, certains opérateurs ne laissent cependant pas le choix : le langage

C++ impose qu'ils soient membre de la classe. Il s'agit de := , () , [] , -> ,

new et delete.

♥ A°)Donner et expliquer un exemple pour un

opérateur binaire de cas où

♦ la 1ère méthode est obligatoire MeMBRe

Lorsqu'on a en premier parametre : operator++

OU operator[]

♦ la 2ème méthode est obligatoire, AMIS

La méthode amis est obligatoire pour les operateurs << et >>

♦ on peut utiliser indifféremment l'une ou l'autre méthode.

operateur+

Question °2) Justifier et expliquer les formes classiques des surcharges des opérateurs +, >, << et ++.

OPeRATeUR + : membre ou friend

friend cdate operator+(cdate d1,int nj);

friend cdate operator+(int nj,cdate d1);

OU

cdate::operator+(int) (PReMIeR PARAM eST UN OBJeT De LA CLASSe)

OPeRATeUR > : membre

cdate::operator>(const cdate &) (PReMIeR PARAM eST UN OBJeT De LA CLASSe) --> const cdate & , const cdate &

OPeRATeUR << : friend car IL APPeL OSTReAM !!

friend ostream & operator<<(std::ostream & s, const cdate & ref)

OPeRATeUR ++ : MeMBRe CAR UNAIRe !

cdate cdate::operator++(int) POST INCRe ou cdate cdate::operator++() PRe INCRe

Question 3°) expliquer en utilisant un exemple pourquoi il peut arriver que l'on déclare une méthode en private.

où compD est une fonction membre (privée, car utilisée uniquement par elle même) de la classe cdate permettant d'effectuer la comparaison entre deux

dates (comparaison de l'année, du mois si nécessaire puis du jour si

nécessaire).

```
int cdate::compD(const cdate& d)
{
 if (annee < d.annee) return -1;
 if (annee > d.annee) return 1;
 // meme annee
 if (mois < d.mois) return -1;
 if (mois > d.mois) return 1;
 // meme mois
 if (jour < d.jour) return -1;
 if (jour > d.jour) return 1;
 // dates egales
 return 0;
}
```

Question °4) expliquer comment certains appels implicites d'un constructeur ou d'un casting peuvent conduire à une application

s'exécutant sans erreurs mais donnant des résultats erronés. Comment se prémunir de ce genre de problème ?

Actuellement, si on écrit

```
…
int main()
{
  cdate d3(25,12,95);
  d3 = 12;
  cout << "d3 = ";
  d3.affiche();

  return 0;
}
```

Le résultat affiché à l'écran (si on exécute le programme le 23/09/2014) ne sera pas le résultat espéré :

```
bash-3.00$ a.out
d3 = 12/9/2014
bash-3.00$
```

Le compilateur a fait appel au **constructeur de date ne réclamant qu'un seul paramètre pour créer un objet constant qu'il a affecté à d3** ☹ !

Pour se premunir de ce genre de probleme on implémente notre proprre opérateur= (opérateur d'affectation).

Theorie 4

L'héritage est un des piliers du C++ et de la Programmation Orientée Objets en

général.

Il vise la récupération du code développé pour un type d'objet donné en vue

d'en créer un nouveau type semblable au premier mais en le spécialisant

davantage ou l'étendant au niveau de ses données et de ses fonctionnalités

Question 1. expliquer et illustrer les différentes associations possibles entre deux classes au moyen

des classes Repas, Plat, Banquet, Ingredients, CalculPrix, RepasBanquet, …. Donner le code

de base de chaque classe avec l'implémentation concrète de l'assciation. Comment faire

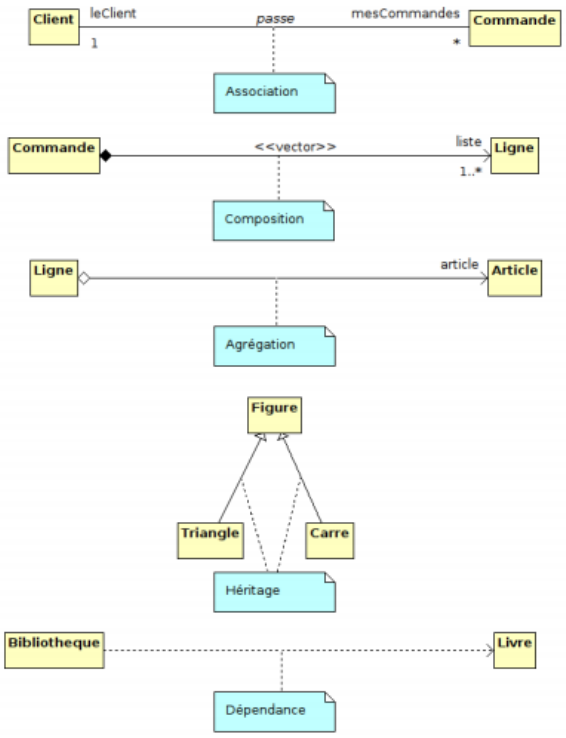
Héritage public : Une classe héritée peut être de type : A est un element de B où B est la classe Mère(base) et A la classe fille(dérivée)

Héritage privée : Une classe héritée peut être de type : A est construite à partir de B

Étant donné qu'en POO les objets logiciels interagissent entre eux, il y a donc des **relations** entre les classes.

On distingue cinq différents types de **relations** de base entre les classes :

- l'**association** (trait plein avec ou sans flèche)
- la **composition** (trait plein avec ou sans flèche et un losange plein)
- l'**agrégation** (trait plein avec ou sans flèche et un losange vide)
- la relation de **généralisation** ou d'**héritage** (flèche fermée vide)
- la **dépendance** (flèche pointillée)



1-Agrégation :

► soit une variable membre pointeur désignant le voyageur : on parle alors de relation de

type "has" par référence10 ou encore de relation d'agrégation par référence ou encore

simplement de relation d'agrégation. Concrètement, on trouvera donc dans la déclaration

de la classe chambre :

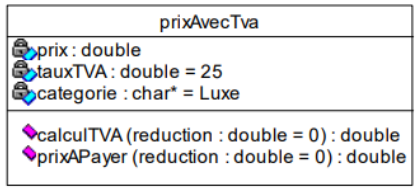
```
 class chambre
{
 private:
 int numXro;
 int coeffConfort;
 char occupe;
 voyageur * leVoyageur;
 public :
 ...
};
```

2- Composition : Agrégation par valeur voir Theorie (plus haut) .

►soit une variable membre instanciant le voyageur : on parle alors de relation de type "has" par valeur ou encore de relation d'agrégation par valeur ou encore de relation de composition. Concrètement, on trouvera donc dans la déclaration de la classe chambre :

```
class chambre
{
 private:
 int numXro;
 int coeffConfort;
 char occupe;
 voyageur leVoyageur;
 public :
 ...
};
```

3-Utilisation




```
class prixAvecTVA
{
        private :
                double prix, tauxTVA;
                char * categorie;
        public :
                prixAvecTVA(double p, double t=25, char * c = "Luxe");
                ~prixAvecTVA();
                double getPrix() const;
                double getTauxTVA() const;
                char * getCategorie() const;

                double calculTVA (double reduction = 0);
                double prixAPayer (double reduction = 0);
};
```

Ajoutons à notre classe chambre une nouvelle méthode prixFinal() dont le nom est assez explicite. Cette méthode pourrait s'écrire :

```
double chambre::prixFinal() const
{
        double reduc=0;

        if (occupe && strcmp(getVoyageur()->getNom(), "vil")==0) reduc=1250;

        prixAvecTVA p(1250+coeffConfort*250);

        return p.prixAPayer(reduc);
}
```

On peut remarquer que cette méthode crée une variable locale p, instance de prixAvecTVA, dans le but d'utiliser la méthode prixAPayer() : on parle alors de relation de type ***"uses"*** ou encore de ***relation d'utilisation***. On entend donc par là que la classe chambre utilise un "service", c'est-à-dire, par exemple, une méthode, une variable memebre statique, ... de la classe prixAvecTVA :

Héritage : Voir question dessous !

Question 2. Dans quelle situation peut-on être amené à créer un héritage ? Comment redéfinir une méthode ou un constructeur dans une classe dérivée en utilisant la méthode ou un constructeur de la classe mère existant ?

Il vise la récupération du code développé pour un type d'objet donné en vue

d'en créer un nouveau type semblable au premier mais en le spécialisant

davantage ou l'étendant au niveau de ses données et de ses fonctionnalités.

►L'héritage permet de ne pas devoir écrire deux fois la même chose en ce qui concerne la partie commune des deux classes. On réinscrit le même nom de méthode par exemple :

Cela revient à dire qu'il faut redéfinir la méthode affiche() au sein de la classe TWagonVoy. On parle d' ***overriding***.

Ce qui donnera :

```
class TWagonVoy : public TWagon
{
  private :
    unsigned int classe;
    int nbreVoy;

  public :
    TWagonVoy(char *num, unsigned int c, int nV=80) : TWagon(num)
    {
      classe = c; nbreVoy = nV;
    }

    void affiche (void); // redéfinition de la méthode affiche()
    void modifNbreVoy(void);
};
```

Question 3. Dans le cas d'une redéfinition de méthode au sein d'un processus d'héritage, décrire exhaustivement (avec code minimal) les deux cas où il faut obligatoirement les qualifier de virtual ? Comment faut -il appeler ces méthodes pour que le mécanisme fonctionne

1)°quand la classe de base (mère) est abstraite les fonctions sont forcement virtual, car la classe sert juste à faire de l'héritage elle sera jamais instancié dans le code donc les méthodes sont obligatoirement virtuelles.

Une méthode virtuelle est une méthode dont la résolution de l'appel s'effectue **à l'exécution** et **pas lors de la compilation** du programme.

A quoi cela sert-il ?

Dans les cas où le type des objets est « clairement » défini, comme dans le cas suivant

```
TWagon w1(…), wPos(…),

TWagonVoy wv1(…), wvSnob(…);

w1.affiche();
wPos.affiche();

wv1.affiche();
wvSnob.affiche();
```

cela ne sert à rien car on connaît, on est sûr de, la nature exacte des objets.

Par contre, quand on a un pointeur d'objets, il se peut qu'on ne puisse pas définir avec exactitude, lors de l'édition de liens, le type d'objet.

Exemple :

```
int main()
{
  TWagon w1("nr125891","Wagon diplomatique"),
         wPos("nr5251","Usage postal", 1987, 32, 8500,6500),
         w3(wPos);

  TWagonVoy wv1("vy526211",2),
            wv2("vy12652",2),
            wvSnob("vy6969",1,50);

  TWagon* train[] = { &wvSnob, &w1, &wv1, &wv2, &wPos, &w3 };
  for (int i=0; i< sizeof(train) / sizeof(train[0]) ; i++)
    train[i]->affiche();

  return 0;
}
```

Le vecteur est un vecteur de Twagon, or, un WagonVoy est aussi un objet de type Wagon donc dans ce cas-ci la fonction affiche va afficher tout les informations de base de wagon et wavonVoy et pas l'affichage spécifique a wagonVoy.

Question 4. expliquer par quels mécanismes il est possible, dans un

processus d'héritage

1) d'assurer que méthode originale ou la méthode redéfinie sera appelée au sein d'un ensemble mixte d'objet de base et d'objets dérivés

2) de pouvoir utiliser de manière sûre une méthode spécifique aux objets dérivés au sein d'un ensemble mixte d'objet de base et d'objets dérivés.

Ces mécanismes s'appelement le upcasting et le downcasting.

Caster vers le haut : Upcasting

Le programme suivant

```
int main()
{
  derivee *pDerivee = new derivee();
  base *pBase;
  pBase = (base*)pDerivee; // Upcasting

  pBase->f();

  delete pBase;
  return 0 ;
}
```

fournit l'exécution suivante

```
bash-3.00$ a.out
1.* constructeur de base
1.** constructeur de derivee
fonction derivee
bash-3.00$
```

On remarque que

```
pBase->f() ;
```

appelle la fonction de la classe dérivée car **la fonction f est virtuelle**.

On parle du « **Upcasting** » car un pointeur d'objet derivee est casté en pointeur d'objet de base. Ce qui ne pose aucun problème dans la mesure où un objet « derivee » est une « base » avec des éléments en plus.

Remarque :

L'instruction

```
delete pBase ;
```

appellera le destructeur de la classe derivee si celui-ci a été déclaré **virtuel** dans la classe base.

ATTeNTION Le DOWN CASTING PeUT POSeR De NOMBReUX PROBLeMeS.

Question 5. expliquer les manières d'utiliser les éléments d'une librairie orientée objet en prenant comme exemple une librairie O.O. d'interfaces graphiques expliquer sur base d'un exemple concret pourquoi le downcasting est parfois nécessaire et quel opérateur "safe" utiliser.

L'opérateur safe a utiliser est :

"dynamic_cast<classe *> (pointeur) "

Question 6. expliquer dans quelles circonstances il peut être intéressant de définir en virtual le destructeur. Même question pour une méthode inline. Décrire la structure et le fonctionnement du code particulier mis en place dans ce cas par le compilateurs.

2) Le destructeur peut être virtuel, comme les fonctions-opérateurs surchargées. Ceci peut être utile dans le cas de structures de données (type container) qui doit détruire ses éléments en fin de traitement en appelant le destructeur approprié.

3) Le mécanisme des fonctions virtuelles ne fonctionne pas dans un constructeur ou un destructeur. C'est compréhensible si l'on pense à l'ordre d'appel des constructeurs ...

Une methode virtuelle si elle possede une reference ou un pointeur sur une classe ne peut pas être inline.

Car l'appel est fait au moment de l'execution. Mais si elle est appelé dans une méthode sans reference ni pointeur, le compilateur connait la classe exacte de l'objet au moment de la compilation, elle peut donc être « inliné ».

Theorie 5

Question 1°) expliquer le fonctionnement de l'héritage multiple en C++; en particulier, expliquer le problème qui peut survenir et comment le résoudre. Même question pour l'héritage en diamant.

Les versions actuelles de C++ permettent de dériver une classe à partir de plusieurs autres : on parle dans ce cas d'"héritage multiple". Si ces classes mères dérivent elles-mêmes d'une même classe de base, on parle alors d'"héritage en diamant", à cause de la forme du schéma d'héritage. Un programmeur peut avoir besoin d'une telle possibilité lorsqu'il souhaite regrouper les caractéristiques de deux (ou même de plusieurs) classes qu'il souhaite conserver distinctes.

L'héritage multiple n'est cependant pas très fréquent : il peut en effet se poser des problèmes d'ambiguïtés. On le voit dans l'exemple ci-dessus avec l'appel de la méthode getChoix(). Si on écrit :

```
cout << endl << "on a choisi " << monMenuC.getChoix() << " !" << endl;
```

le compilateur hurle à l'appel ambigu ! en effet, les deux classes de base de menuAnsiAvecSm héritent toutes deux de menu : la classe menuAnsiAvecSm possède donc deux méthodes getChoix() ! Il faut donc préciser dans l'appel quelle est la méthode utilisée :

```
cout << endl << "on a choisi "   << monMenuC.menuAvecSm::getChoix() << " !" << endl;
```

etait-il indifférent d'appeler plutôt monMenuC.menuAnsi::getChoix() ?

Pour résoudre le probleme :

Il suffit donc de dériver virtuellement les deux classes intermédiaires menuAnsi et menuAvecSm à partir de la classe menu. De cette manière, la classe de base n'est pas recopiée telle quelle dans la classe dérivée : c'est, en quelque sorte, un pointeur sur celle-ci qui est incorporé à la classe dérivée. en cas d'héritage multiple, ces pointeurs désignent ainsi les mêmes variables membres.

Question 2°) expliquer le fonctionnement du lancement d'une exception et le comparer au mécanisme C du longjump. en particulier, expliquer comment une instance de classe d'exception peut être récupérée pour gérer l'erreur survenue.

Les exceptions sont le mécanisme de gestion d'erreurs propre à l'ensemble des langages de programmation orienté objet, et au C++ en particulier

Le mécanisme des exceptions, au contraire, permet de se brancher à un endroit particulier lorsque tel ou tel événement se produit.

Ce qui a pour intérêt :

- De séparer clairement la logique de programmation de la gestion des erreurs

◊ augmentation de la lisibilité du code.

- De réaliser une gestion d'erreurs plus fine et plus adéquate.

- De normaliser, de systématiser, la gestion des erreurs.

- De pouvoir gérer les erreurs se produisant dans les constructeurs de

classe.

Les branchements en C

en C, les sauts non locaux (voir cours de Systèmes d'exploitation) ou branchement se font à l'aide des fonctions :

- setjmp(jmp_buf) qui mémorise l'environnement d'appel. Cette fonction

retourne 0 lors d'un appel direct, c'est-à-dire lors de son premier appel.

- longjmp(jmp_buf,int) qui permet de restaurer cet environnement.

Concrètement, l'exécution retourne au point marqué par le « setjmp ».

Celui-ci renvoie alors cette fois comme valeur le second argument de la

fonction « longjmp ».

Ces deux fonctions permettent donc de se brancher en un point lors d'une

erreur. Mais il n'est pas possible de détruire les variables automatiques créées

entre l'appel de « setjmp » et de « longjmp » !

Le C++ définit un mécanisme d'exceptions qui possède 3 instructions :

1) try

Un segment de code dans lequel une exception peut se produire doit être placé

dans un bloc préfixé du mot réservé « try ».

---

try

{

<code pouvant donner lieu à une exception> ;

}

Ceci indique qu'il faudra tester l'existence d'exceptions éventuelles.

2) throw

Les exceptions du C++ sont en réalité des objets. en réalité, c'est le type

d'objet qui présente un intérêt.

Lorsqu'une situation d'exception se présente, le programme peut lancer un

objet à l'aide de l'instruction :

throw(objet) ;

en fait, Cette instruction crée un objet temporaire qui est initialisé au moyen du constructeur de copie de la classe. L'exécution est alors transférée (sans espoir de retour !) au point déterminé par l'instruction suivante (catch). Mais, Contrairement aux fonctions « setjmp » et « longjmp », il y appel au destructeur des objets créés depuis l'entrée dans le bloc try.

3) catch

Un bloc « try » est obligatoirement suivi d'un certain nombre de blocs

« catch ». Ces blocs sont appelés les « gestionnaires d'exceptions » ou

« handler » :

---

catch(<paramètre d’une classe donnée>)

{

<gestion de l’exception> ;

// code exécuté en cas d’exception

}

Lorsqu’un throw est exécuté, le programme se branche au handler le plus

proche correspondant au type d’objet lancé.

- Si il le trouve, la pile est vidée de ses variables automatiques et le

contrôle est transféré au handler.

- Si il ne le trouve pas, le programme se termine de la manière habituelle

en appelant la fonction terminate() dont nous reparlerons plus loin.

Question 3) Qu'est-ce qu'une classe d'exception ? Quels éléments comporte-t-elle au minimum et pourquoi ? Quelle est la structure classique d'un handler d'exception ?

STRUCTURe CLASSIQUe D'UN HANDLeR :

---

catch(<paramètre d’une classe donnée>)

{

<gestion de l’exception> ;

// code exécuté en cas d’exception

}

---

eLeMeNTS MINIMUM :

Constructeur par defaut ? Car par defaut

Destructeur (il y est pas défaut)

Constructeur de copie (car l'objet lancé est crée par le biais d'un constructeur de copie).

Dès que le nombre d'exceptions à traiter devient important, il est naturel de définir une classe pour chaque type d'exception.
Certaines sont indépendantes les une des autres, d'autres appartiennent à la même famille.
L'idée est de créer une hiérarchie de d'exceptions

Bien sûr, il convient de placer le handler des classes dérivées avant celui de la classe de base !

Les blocs « catch » doivent être placés de manière à avoir les classes les plus spécialisées dans le traitement d'une exception en premier et les plus générales vers la fin.

4. expliquer et justifier ce que l'on entend par "container" en C++ en vous basant sur l'exemple de la classe Vecteur. en particulier, en utilisant l'exemple de la classe Pile, expliquer comment l'encapsulation appliquée dans les classes containers assure le maximum d'abstraction.

Un container (conteneur en français) est une classe qui contient une collection de données (objets ou types de base),

Un premier exemple : un classe « Vecteur de réels »

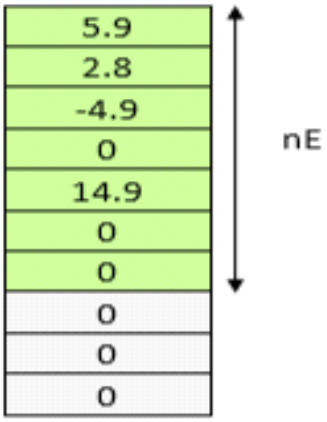
```
class CVecteur
{
  private :
    float v[10];
    int nE;
    char checkIndex (const int i);

  public :
    CVecteur (void);
    CVecteur (const int valInit);
    CVecteur (unsigned int nEl) { nE = nEl; }

    int getDim (void) { return nE; }
    void setDim (const int nEl);

    float getVal (int i);
    void setVal (int i, float val);

    void afficheVec (void);
    void entreVec (void);
};
```



Une telle classe est consistante et cohérente, car elle fournit à l'utilisateur toute l'interface qui permet d'incorporer des objets de cette classe dans un programme.

Pourquoi parle t-on d'abstraction ?

Tableau statique

```
class pile
{
  private:
    int nbreElem ;
    int tab[N_MAX];
    int pTop;

  public:
    pile(int n=N_MAX);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};
```

Une version un peu « moins statique »

```
class pile
{
  private:
    int tab[N_MAX];
    int pTop;


  public:
    pile(int n=N_MAX);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};
```

Tableau dynamique

```
class pile
{
  private:
    int *tab;
    int *pTop;

  public:
    pile(void);
    ~pile(void);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};
```

Liste chaînée dynamique

```
class pile
{
  private:
    noeud *pPile;

  public:
    pile(void);
    ~pile(void);
    bool push(int n);
    bool pop(void);
    int top(void);
    bool empty(void);
    bool full(void);
    void affContenu(void);
};

struct noeud {
  int valeur;
```

Cela est dû au fait que les 4 versions diffèrentes (pour un même emploi) seulement dans leur implémentation et pas dans leur interface (les méthodes ont conservé le même prototype) ◊ c'est l'intérêt majeur de l'encapsulation et donc de l'abstraction car l'utilisateur ne sait pas qu'elle classe il utilise et il s'en fiche !

Theorie 6

- **Définir ce qu'est une classe abstraite et expliquer les rôles en utilisant l'exemple des containers liste/pile/pile/listeTriee. expliquer et comparer la manière utilisée en C++ et en Java pour indiquer qu'une**

classe est effectivement abstraite; quel effet souhaite--t-on ainsi obtenir

Une classe abstraite est une classe qui n'est jamais instanciée. en clair elle sert de base aux autres classes (filles). elle doit être donc le plus général possible.

En entrant dans les détails :

Classe abstraite pure

```
class structure
{
  protected :
    int nbreElem;

  public :
    structure () { nbreElem = 0; }
    virtual int tete (void) = 0;
    virtual boolean empty (void);
    virtual boolean full (void);
    virtual boolean ajout (int n);
    virtual boolean retrait (void);
};
```

```
class liste : public structure
{
  protected :
    noeud *pListe;

  public :
    liste (void);
    ~liste (void);
    int tete (void);
    boolean empty (void);
    boolean full (void);
    void affContenu (void);
};

struct noeud {
  int valeur;
  noeud *suiv;
};
```

Elle permet de

- **définir l'ensemble des services** que tous les éléments de la hiérarchie devront rendre aux utilisateurs
- définir en **un seul exemplaire** une ou plusieurs partie(s) de code qui seraient **communes à l'ensemble des classes dérivées**.
- définir un **interface commun** à l'ensemble des classes de la hiérarchie.
- fournir un **service élémentaire** (comme le comptage du nombre d'éléments dans notre exemple).

Remarque :

etant donné que plusieurs fonctions devront sûrement être redéfinies dans les classes dérivées, il est plus prudent de les déclarer comme étant virtuelles.Il faut indiquer au compilateur de ne pas générer de code pour les fonctions dont on ignore dans la super-classe la manière comment elles vont être implémentées Mais si l'on déclare les méthodes en virtuelles, elle pourront quand même être instanciée. Or on veut que la super-classe ne le soit pas.

```
virtual int tete(void) = 0 ;
```

On déclare donc les méthodes virtuelles en virtuelles pures :

Donc en C++ il suffit qu'une methode de la classe soit virtuel pure pour que la classe le soit.

```
1  abstract class Animal{ }
```

en java on la déclare simplement :

- **<u>expliquer et justifier la notion d'itérateur : quel est l'objectif poursuivi ? Quel est le code C++ classique de parcours d'un container au moyen d'un itérateur ? Quelles sont typiquement les variables membres et les méthodes que l'on trouve dans ce genre de classe ? Illustrer avec le code de base de l'opérateur ++ pour un itérateur de vecteur et de liste chaînée.</u>**

Un **itérateur** est un objet qui permet de **balayer une structure de données**, et d'**accéder ainsi à tous les objets** qu'elle contient, **selon un ordre bien établi**.

A quoi cela sert-il ?

Il est parfois utile de pouvoir avoir accès à tous les éléments contenus au sein d'une structure de données pour, par exemple,

- les afficher un par un
- leur faire subir un traitement via l'appel d'une fonction
- etc...

Sans l'itérateur, il serait indispensable de connaître de manière approfondie l'implémentation du container (indice, pointeurs, ...) mais ceci est à l'opposé du principe d'encapsulation.

La notion d'itérateur permet donc de définir un **procédé standardisé d'accès** aux objets contenus au sein d'un container, et cela, sans en connaître l'implémentation.

Les caractéristiques d'un itérateur

1. Un itérateur doit être fortement lié à la classe « container » qu'il doit manipuler. Il doit en connaître les détails d'implémentation et accéder à ses données membres.

   Dès lors, La classe itérateur doit être **amie** de la classe container

2. L'itérateur doit être **initialisé** par son constructeur, sur le début de la structure de données qu'il permet de manipuler. Il doit pouvoir être ramené au début.

3. Il faut pouvoir **détecter la fin** de la structure de données et aussi pouvoir **réinitialiser** l'itérateur.

4. On passe d'un élément de la structure au suivant au moyen de **l'opérateur ++** : celui-ci devra donc être surchargé.

5. L'itérateur doit pouvoir **renvoyer l'élément** (objet) auquel il est parvenu.

```
class CIntVecIter
{
  private :
    CIntVecteur& p;
    int *pData;

  public :              // Pointeur initialisé au début
    CIntVecIter (CIntVecteur& s) : p(s), pData(s.tab) {};
    bool end() const  // Détecte la fin du vecteur
    {
      if (pData - p.tab >= p.taille) return true;
      else return false;
    }
    void reset() { pData = p.tab; } // Réinitialize l'itérateur
    bool operator++()  // Passe à l'élément suivant
    {
      if (!end())
      {
        pData++;
        return true;
      }
      else return false;
    }
    bool operator++ (int) { return operator++(); }
    operator int() const { return *pData; } // Renvoie l'élément
    void insert(int n) { *pData = n; }
};
```

Un itérateur pour un tableau (Retour sur la classe vecteur) :

- **expliquer ce qu'est une classe "template" (ou générique" ou "paramétrable"). Illustrer et justifier, avec le code d'un vecteur template basique, la syntaxe utilisée dans différentes situations.**

La notion de **template** permet de créer en C++ des **classes** (ou des **fonctions**) **génériques**, c'est-à-dire **indépendantes d'un type de données**.

➔ Une **fonction « template »** est une fonction dont au moins un des paramètres est « générique », c'est-à-dire qu'il peut être d'un type quelconque

➔ Une **classe « template »** est une classe dont au moins une de ses variables membres est « générique », c'est-à-dire d'un type quelconque.

Syntaxe :

```
template <class T> class TVecteur    // Définition de la classe
{                                    // template TVecteur
  private :
    T *data;
    int nbreElem;

  public :
    TVecteur (int n=10);
    ~TVecteur (void) { delete data;}
    T& operator[] (int i) {return *(data+i); }
};

template <class T> TVecteur<T>::TVecteur(int n)
{
  data = new T[n];    // Appel du constructeur par defaut de T
  nbreElem = n;
}
```

Theorie 7

1. expliquer le problème de l'instanciation des templates dans le cas où l'on travaille en modules séparés (pourquoi/comment il faut-il "forcer" cette instanciation ?). Illustrer par l'exemple d'un vecteur template utilisé pour manipuler trois vecteurs distincts (par exemple, un vecteur d'entiers, un vecteur de Cdate et un vecteur de char)

Le probleme des instanciation des templates en modules séparés est le fait que certains compilateurs plus anciens que d'autres n'instancient pas les templates automatiquements. Il va falloir donc forcer l'instanciation en donnant à l'avance les types que peux contenir le template :

Forcer avec un pragma :

```
#pragma define_template TVecteur<int>
#pragma define_template TVecteur<char>
#pragma define_template TVecteur<coord>
```

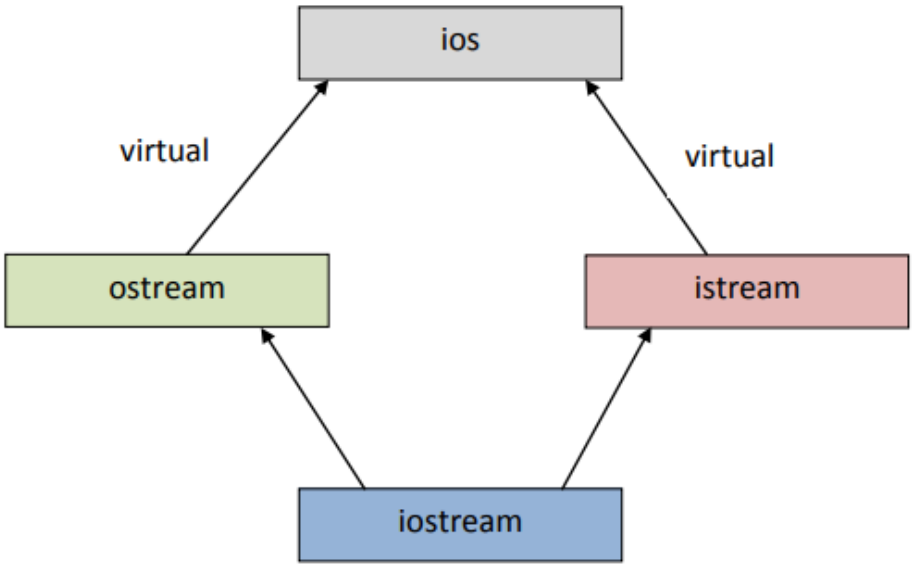
Les 3 vecteurs :

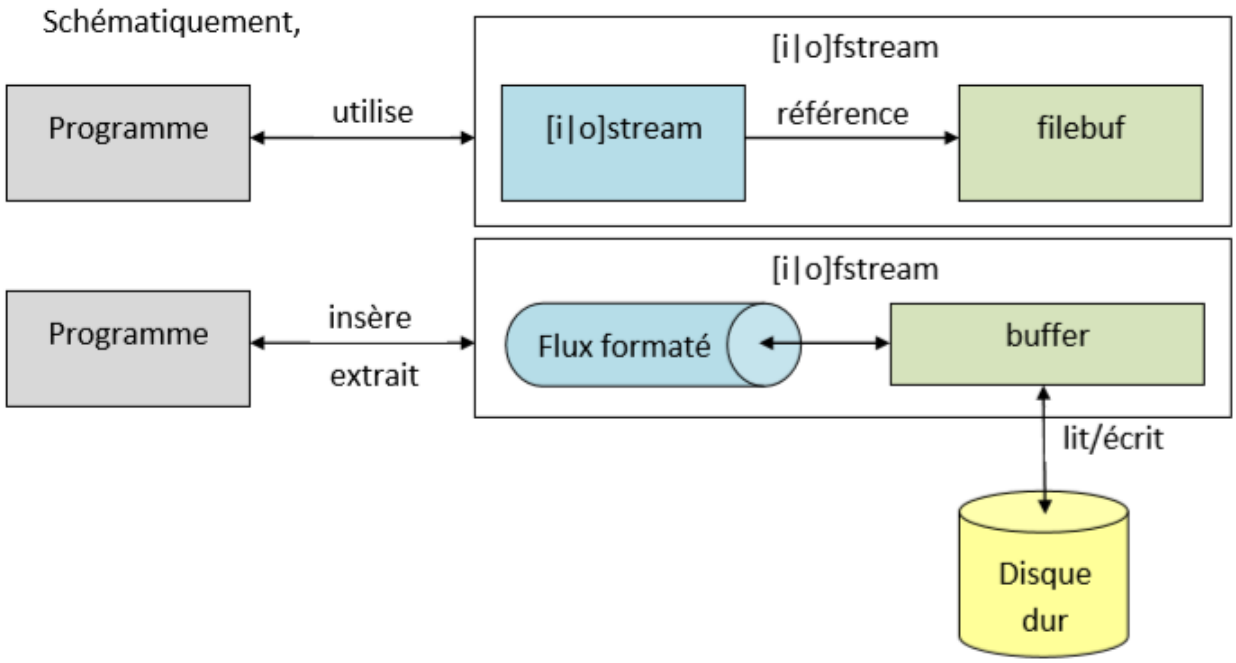
```
#include "CDate.h"
template class TVecteur<int>;
template class TVecteur<char>;
template class TVecteur <cdate>;
```

2. expliquer en la justifiant l'architecture des flux en 2 (4) couches du C++. Un flux de bas niveau est-il utilisable sans un flux de haut niveau associé ? et l'inverse ? Décrire (pas seulement citer !) les principales classes (et leurs associations d'héritage et d'agrégation) nécessaires pour réaliser les traitements standards sur des fichiers.

Un flux bas niveau et haut niveau sont toujours associer quand il y a une interface utilisateur. On pourrait imaginer par contre un programme qui envoie des données de serveur à serveur.

Les Principales classes des flux :





. Le bas niveau travail en byte et permet de faire des e/S sur le disque/réseau etc

1) Ios :

Il s'agit d'une classe abstraite, qui est la classe de base commune à toutes les classes de la hiérarchie. elle rassemble les caractéristiques communes à tous les flux formatés.

elle contient :

- La matérialisation d'une connexion d'e/S : il s'agit en fait d'un pointeur sur un objet de la classe streambuf. L'accès à ce pointeur s'effectue à partir de la valeur renvoyée par la méthode

streambuf* rdbuf(void) ;

- Le contrôle du flux : méthodes permettant de connaître l'état du flux à un instant donné. L'état du flux est mémorisé dans une variable membre privée « state ». Les méthodes d'accès sont

2°) Ostream :

Cette classe contient toutes les surcharges classiques de l'opérateur d'insertion "<<".

Ces surcharges réalisent toutes les conversions nécessaires pour les données numériques alors que les chaînes de caractères sont affichées telles quelles.
Si la donnée insérée est l'adresse d'un objet streambuf, c'est l'ensemble du flux qui est inséré sur le flux cible.

3) La classe istream

Cette classe contient toutes les surcharges classiques de l'opérateur d'extraction ">>", réalisant les conversions nécessaires :

4) La classe iostream

Cette classe hérite à la fois de istream et de ostream (et donc aussi de ios de manière virtuelle). elle contient donc l'ensemble des méthodes permettant d'accéder de manière standard à un périphérique d'entrée ou à un périphérique de sortie.
elle sera utilisée comme classe de base :

- de la classe fstream : classe spécialisée dans les opérations d'e/S sur fichier(s)
- de la classe strstream : classe spécialisée dans les opérations d'e/S en mémoire

3. expliquer ce qu'est un manipulateur dans le contexte des flux et expliquer la surcharge de l'opérateur << qui justifie la manière de l'utiliser. Décrire les manipulateurs endl, ends et quelques manipulateurs de formatage.

But : les manipulateurs sont des fonctions opérant sur un flux et dont le but est de simplifier, ou de rendre plus souple, le formatage des données.

Définition

Un manipulateur est une fonction opérant sur un flux d'e/S et qui est invoquée au sein même d'une insertion ou d'une extraction.

exemple:

Au lieu d'écrire

cout.precision(10) ;

cout << n << endl ;

On peut utiliser le manipulateur « setprecision » et écrire :

cout << setprecision(10) << n << endl ;

La souplesse acquise est évidente. Mais, endl est aussi un manipulateur ! ◊ il existe bon nombre de manipulateurs prédéfinis

Surchage de << :

ostream& operator<< (short);

ostream& operator<< (unsigned short);

ostream& operator<< (int);

ostream& operator<< (unsigned int);

ostream& operator<< (long);

Quelques formateur :

- **endl** : qui insère un retour de chariot sur le flux
- **ends** : qui insère un zéro de fin de chaîne sur le flux
- **flush :** qui vide le buffer de sortie

·**hex:** qui formate au format hexa etc..

4. expliquer comment on procède en pratique pour écrire des données dans un fichier binaire

(méthode avec deux classes, méthode avec une seule classe) ou dans un fichier texte. en quoi

les deux fichiers obtenus sont-ils différents ?

Pour traiter un fichier, il faut associer une instance de filebuf a un flux (de type output,

input ou les deux, selon le cas).

5. expliquer à partir d'un exemple pourquoi il est intéressant de travailler sur une zone

mémoire en utilisant un flux.

L'intérêt majeur des flux en mémoire est de sauvearder un affichage formaté

en mémoire, à la manière du sprintf en C.

Exemple :

```
char phrase2[60] ;
ostrstream maPhrase2(phrase2,60) ;
maPhrase2 << « Des tetes bien faites » << ends ;
cout << phrase2 << endl ;
```

6. expliquer ce qu'est l'objet cout et ses fonctionnalités en termes d'écriture formatée, de surcharges de l'opérateur << et de redirection.

cout : flux de sortie associé à l'écran (le stdout du C);

2.2 Le flux de sortie cout

Afficher une donnée revient donc à l'insérer sur le flux de sortie cout. Cette insertion s'indique au moyen de l'opérateur << . Oui, il s'agit de l'opérateur de shift gauche : il a été "*surchargé*". Ceci signifie qu'il a été redéfini pour le cas où il s'applique à un flux afin de représenter l'opération d'insertion ! La syntaxe de l'opération d'écriture devient donc :

cout << { *variable* | *constante* | *expression* } [<< { *variable* | *constante* | *expression* }] [...]

9. La vraie nature de cin et cout

Les classes *ostream_withassign* et *istream_withassign* sont des variantes des classes ostream et istream qui, comme leur nom l'indique, permettent l'assignation, l'affectation.

Le but de celle-ci est de permettre une **redirection**. Les flux prédéfinis ***cin***, ***cout*** et ***cerr*** sont précisément des instances de ces classes.

Le programme suivant redirige la sortie vers le fichier "redirec".

As an object of class ostream, characters can be written to it either as formatted data using the insertion operator (operator<<) or as unformatted data, using member functions such as write.