EFREI

L2 TP1 C++

Exercice 1 : Manipulation de références sur des types primitifs

Rappel : une référence en C++ est un pointeur "caché". En utilisant une référence, on crée un alias de la variable référencée, c'est à dire un autre nom qui désigne la même adresse.
Pour les paramètres de fonction ou méthodes, une référence sera donc non pas une copie de la variable transmise en argument, mais la même variable.

Pour bien comprendre comment fonctionnent les références, implémenter les exemples suivants et commenter les résultats obtenus. Coder ces fonctions directement dans le fichier **main.cpp.**

Attention : il faut créer un projet pour chaque exemple, car les deux fonctions proposées ne peuvent pas cohabiter dans un seul et même projet.

Ces premiers exemples utilisent des **fonctions externes comme en langage C.**
Le code de main() est commun aux deux exemples.

Exemple 1 : sans utiliser de référence :

```
// j est un paramètre "normal", il reçoit une copie de la valeur passée en argument
void fonc(long j)
{
	cout << "adresse de j :" << &j;
	cout << "dans fonc(), valeur de j avant incrément :" << j << endl;
	j++;
	cout << dans fonc(), valeur de j apres incrément :" << j << endl;    return;
}
```

Exemple 2 : en utilisant une référence

```
 // j est une référence : il s'agit donc de la variable  passée en argument
// Regarder les adresses affichées
void fonc(long &j)
{
        cout << "adresse de j :" << &j;
        cout << "dans fonc(), valeur de j avant incrément :" << j << endl;
        j++;
        cout << dans fonc(), valeur de j apres incrément :" << j << endl;
        return;
}
```

Fonction main :

```
int main(int argc, char *argv[])
{
        long i;
        i = 4;
        cout << " adresse de i : " << &i << endl;
        cout << " dans main(), valeur de i avant fonc() " << i << endl;
        fonc(i);
        cout << " dans main(), valeur de i apres fonc() " << i << endl;
        return 0;
}
```

Exercice 2 : Manipulation de références sur des objets

Exemple 1 : sans utiliser de référence :

Implémenter la classe suivante dans un **fichier ouch.h.**

```
class ouch
{
private :
        long a;
public :
        ouch() {a = 0;}
        ~ouch() {}
        void incr(long n) {a  =  a + n;}
        void print() { cout << "a = " << a << endl; }
};
```

Puis le code suivant dans une **fichier main.cpp** :

```
void fonc(ouch objet)
{
        cout << "debut fonc()" << endl;
        objet.print();
        objet.incr(2);
        objet.print();
        cout << "fin fonc()" << endl;
}
```

```
int main(int argc, char *argv)
{
        ouch o;
        o.print();
        o.incr(4);
        o.print();
        fonc(o);
        o.print();
}
```

Que se passe-t-il à l'exécution ?

Exemple 2 : en utilisant de référence

Modifier simplement l'entête de la fonction ***fonc()*** de la manière suivante :**void fonc(ouch& objet)**

Que se passe-t-il à l'exécution ?

Que se passe-t-il après la modification suivante de l'***entête de fonc() :***

void fonc(const ouch& objet)

En fait le compilateur se plaint car le paramètre objet doit être **const**, et les méthodes appelées (print() et incr()) ne le sont pas.

Pour remédier à ceci, modifier les méthodes de la **classe ouch** pour qu'elles soient **const** de la manière suivante :

```
void incr(long n) const {a = a+n;}
void print() const {cout << "a = " << a << endl;}
```

Que se passe-t-il à la compilation ?

Exercice 3 : Surcharge de fonction

La surcharge de fonction permet d'avoir plusieurs fonctions ayant le même nom mais avec un nombre de paramètres différents et des paramètres de types différents.

Implémenter une fonction nommée ***func*** qui affiche quelle version est appelée, puis appeller cette fonction avec des arguments de différents types afin de matérialiser la surcharge.

Il faut 4 versions distinctes de cette fonction.

Exemple :

```
void func(int a, int b)
{
        cout << "toto(int, int)" << endl;
        return;
}
```

```
void func(double x)
{
        cout << "toto(double)" << endl;
        return;
}
Etc…
```

Exercice 4 : Surcharge de fonctions, références

Ecrire une fonction qui échange la valeur de ses deux paramètres de ***type char***, et qui fait en sorte que cet échange soit effectif pour les arguments utilisés lors de l'appel de la fonction en utilisant des références.

Surchargez cette fonction pour qu'elle échange ses deux paramètres, que ceux-ci soient tous les deux de ***type char, long ou double***.

Exercice 5 : Retour de référence et lvalue

Ecrire une **fonction valAt** qui retourne une référence sur une valeur entière qui est la valeur située dans un **tableau d'entiers tab** à l'indice i. tab et i sont fournis en paramètres de la fonction valAt.

Ecrire un programme qui affiche le contenu d'un tableau d'entiers en utilisant la fonction valAt.

Ecrire un programme qui initialise le contenu d'un tableau d'entiers en utilisant la fonction valAt.

Exercice 6 : Arguments avec des valeurs par défaut

Ecrire une **fonction somme** qui retourne la somme de ses paramètres, tous de ***type double***.

On doit pouvoir appeler cette fonction en lui fournissant entre 0 et 4 arguments.

Exercice 7 : Classe complex

On rappelle que pour l'écriture de programmes C++ avec des classes, la **programmation modulaire** est plus que souhaitable :

- Un **fichier main.cpp** avec le programme principal
- Un **module** ou plusieurs modules comportant :
    - un fichier **.h** avec la définition de la classe utilisée
    - un fichier **.cpp** avec les définitions des méthodes de la classe utilisée.

Classe complex

Ecrire la classe **complex**, avec ses constructeurs par défaut, avec arguments (par défaut, créer le complexe 0+0i) et son destructeur.
Toutes ces méthodes doivent afficher un message indiquant qu'elles sont en cours d'exécution.

On vous fournit le squelette de **complex.h,** vous devez écrire le fichier **complex.cpp**

```
class complex
{
private :
        double re, im;
public :
        complex();
        complex(double, double);
        ~complex();
};
```

Modifier le constructeur avec arguments afin qu'il ait deux arguments par défaut (valant 0).
Le constructeur par défaut (celui sans argument) est-il toujours nécessaire ?

Tester la classe Complex dans le **fichier main.cpp.**

Exercice 8 : Classe complex suite

Constructeur par recopie

Ajouter à la classe complex un constructeur par recopie affichant un message, puis ajouter au programme principal une instruction permettant l'appel de ce constructeur par recopie.
Le prototype du constructeur par recopie est : **complex(const complex &)**;

Exercice 9 :Recopie et tableaux statiques

- Créez une classe X dont un des membres est un **tableau alloué statiquement.**
- Créez deux objets a et b de classe X, en créant b à partir de a.

```
X a;
// création de b à partir de a : constructeur par recopie
X b(a);
```

- Quel est le constructeur appelé pour créer b ?
- Modifier un élément du tableau stocké dans a et affichez b.
- Qu'en déduire sur le constructeur utilisé pour créer b ?
- Ecrivez votre propre version de ce constructeur et testez-la.

Squelette de la classe X :

```
class X
{
        private :
                long vals[10];
                long util;
        public :
                X();
                X(const X &);
                ~X();
};
```

Exercice 10 : Recopie et tableaux dynamiques

new, new[], delete, delete[]

- Créer une classe Y dont un des membres est un tableau dynamique (donc un pointeur).
- Créer deux objets e et f de classe Y, en créant f à partir de e.
- Quel est le constructeur appelé pour créer f ?
- Modifier un élément du tableau stocké dans e et afficher f.
- Qu'en déduire sur le constructeur utilisé pour créer f ?
- Ecrire sa propre version de ce constructeur et la tester.
- Ecrire le destructeur sans oublier les [].

.

Atttention : Ne pas oublier d'utiliser les opérateurs d'allocation et de libération dans les constructeurs et destructeur!

Squelette de la classe Y :

```
class Y
{
private :
        double * vals;
        long tmax,  tutil;
public :
        Y(long = 10);
        Y(const Y &);
        ~Y();
};
```

Code du constructeur :

```
Y::Y(long size)
{
        if (size > 0)
        {
                tmax = size;
                vals = new double[tmax];
        }
        else
        {
                tmax = 0;
                vals = NULL;
        }

        tutil = 0;
}
```

Exercice 11 :Un destructeur loquace

- Créer une classe D dont les constructeurs et le destructeur s'annoncent en écrivant un message.
- Ecrire un programme créant des objets de classe D ainsi qu'un ou des pointeurs sur des objets de classe D.
- Quand les destructions d'objets ont-elles lieu ?
- Trouvez un moyen de pouvoir visualiser tous les messages affichés par le destructeur s'il en manque.
- Y a-t-il toujours autant de destructions d'objets que de créations d'objets ?

Exercice 12 : Tableau 2D

- Créer une **classe T** stockant **un tableau dynamique 2D** comportant **p lignes** et **q colonnes**.
- Les éléments de ce tableau doivent être des **objets d'une autre classe O.**
- Les constructeurs et le destructeur de T doivent faire des affichages pour indiquer qu'ils sont en cours d'exécution.
- Quand les constructeurs et les destructeurs de O sont-ils appelés ?
- Créer deux objets a et b de type T dans un programme principal, b est une copie de a.
  Observer les créations et destructions d'objets.
- Créer deux pointeurs c et d de type T * dans un programme principal.
  Initialiser c en utilisant l'opérateur new.
  Affecter c à d
  Libérer c grâce à l'opérateur delete. Faut-il libérer d?
  Observer les créations et destructions d'objets.

Les squelettes des classes O et T et le code du destructeur de T sont donnés ci-après.

Squelettes des classes O et T :

```
using namespace std:

class O
{
private :
        char _dummy;
public :
        O() {cout << "O::O()" << endl;}
        ~O() {cout << "O::~O()" << endl;}
        O(const O &) {cout << "O(const O&)" << endl;}
};
```

```
class T
{
private:
        O ** tab2d;
         // p lignes, q colonnes
        long p, q;
public :
        T(long = 5, long = 5);
        ~T();
        T(const T &);
};
```

Code du destructeur de T (s'en inspirer pour le constructeur)

```
T::~T()
{
        if(tab2d == NULL)
                return:

        for (long  i= 0; i < p;  i++)
        {
                if (tab2d[i] != NULL)
                {
                        delete [] tab2d[i];
                }
        }
        delete [] tab2d;
}
```

Exercice 13 : Relation de composition

Conception d'une ***classe CPersonne*** qui contient le ***nom*** ainsi que la ***date de naissance*** d'une personne, et qui puisse être utilisée avec le programme suivant :

```
// fonction externe
void appelParValeur( CPersonne  p )
{
…
}
void main()
{
        // Philippe Durant est né le 10/01/1979
        CPersonne phil("Philippe Durant", 10, 1, 1979 ) ;
         // récupère le nom de M. Durant
        const char * nom = phil.getName() ;
        cout << "Nom =" << nom << endl ;
         // récupère la date de naissance
        const CDate & date = phil.getDate() ;
        cout << "date=" << date.getDay() << "/" << date.getMonth() << "/"
                                                    << date.getYear() << endl ;
        appelParValeur( phil ) ;
}
```

1. Gestion du nom

Ecrire la déclaration de la ***classe CPersonne*** avec un tableau de caractères de taille maximum constante (fixe) et alloué statiquement pour représenter le nom.

Ajouter les fonctions membres de cette classe pour qu'elle fonctionne avec le programme suivant :

```
void main() {
        CPersonne phil( "Philippe Durant" ) ;
        const char * nom = phil.getName() ;
        cout << "nom=" << nom << endl ;
}
```

2. Gestion de la date

- Compléter la déclaration de la ***classe CPersonne*** avec un ***objet de type CDate*** pour représenter la date de naissance. Les attributs de la classe ***CDate*** sont tous de type entier.
- Ajouter des fonctions membres à cette classe pour qu'elle fonctionne avec le programme donné en début d'exercice.
- Ecrire la ***classe CDate*** dont le constructeur par défaut qui initialise avec la date 01/01/1970..

3 Transmission d'un objet de type CPersonne par valeur à une fonction

Que se passe-t'il quand on cherche à transmettre un objet de ***type CPersonne*** à une fonction ?

Par exemple :

```
void AppelParValeur( CPersonne p )
{
        …
}
void main()
{
CPersonne phil( "Durant", 10, 1, 1977 ) ;
AppelParValeur( phil) ;
}
```

Quelle solution permettrait d'optimiser le passage de paramètre d'un objet de type Cpersonne?

Exercice 14 : Relation d'association avec pointeur (agrégation)

On décide de changer l'implémentation de la ***classe CPersonne*** de l'exercice précédant en représentantvl'association de la date ainsi que son nom par des pointeurs.

Cela donne la déclaration suivante :

```
class CPersonne
{
        char * name ;
        int age ;
        CDate * dateNaissance ;
        public :
…
} ;
```

Toutefois, l'interface de la classe ne doit pas changer, et le programme de l'exercice précédent doit continuer à fonctionner de la même façon avec cette nouvelle implémentation.

1. Gestion du nom

Modifier si nécessaire le constructeur de la ***classe CPersonne*** ainsi que la ***méthode getName***.

2. Gestion de la date

Modifier si nécessaire le constructeur de la ***classe CPersonne*** ainsi que la ***méthode getDate.***

3 Transmission d'un objet de type *CPersonne* par valeur à une fonction

Que se passe-t'il quand on cherche à transmettre par valeur un objet de ***type CPersonne*** à une fonction ?

Par exemple :

```
void AppelParValeur( CPersonne p )
{
…
}
```

```
void main()
{
        CPersonne phil( "Philippe Durant", 10, 2, 1979 ) ;
        AppelParValeur( phil) ;
}
```

Pour résoudre ce problème, ajouter un constructeur de copie et un destructeur à la classe *Cpersonne*.

4 Affectation d'un objet de type CPersonne a un autre objet du même type

Que se passe-t'il quand on cherche à affecter un objet de ***type CPersonne*** à un autre objet de même type ?

Par exemple :

```
void main()
{
        CPersonne phil( "Philippe Durant", 10, 2, 1977 ) ;
        CPersonne albert( "Albert Gontrand", 24, 8, 1907 ) ;
        albert = philippe ; // réincarnation
}
```

- Que peut-on faire pour résoudre ce problème ?
- Faites les modifications nécessaires au programme.