Programmation orientee objet C++

Version 1.0

enseignants dpt info iuto

avr. 03, 2020

Les sujets de Cours

Contents :

CHAPITRE 1

COURS1

1.1 PROGRAMMATION ORIENTEE OBJET : C++

1.1.1 Introduction :

Le langage C++ est un langage de programmation orienté objets. **Bjarne Stroustrup** a développé ce langage dans les années 1980. Il s'agit d'une amélioration du langage C, dans lequel il a introduit les concepts d'"*objets* et de *généricité*.

- Pour être exécuté, un programme C++ doit d'abord être *compilé* en langage de bas niveau : **g++ -c Premier.cpp**. Cette étape construit un fichier **Premier.o**.
- Le programme compilé est ensuite *exécutable* si on lui fait subir une édition des liens : **g++ -o Premier Premier.o**.
- L'exécution se fait par appel à **Premier** : **./Premier**.
- Une version simplifiée de l'exécution de Premier.cpp peut se faire par : **g++ -o Premier Premier.cpp**

En C++, un programme doit au minimum débuter par les lignes suivantes et contenir une fonction **main** :

```
#include<iostream>
using namespace std;

int main() {
   // programme principal
   int x = 5 , y = -6; // déclarations de variables
   int aux = x;
   x = y;
   y = aux;
   cout<<" x = "<<x<<" y = "<<y;
   // affichage
   return 0;
}
```

L'inclusion (*#include*) permet de gérer les entrées - sorties.

- En C++, les variables ont un type et doivent être déclarées (pas nécessairement définies).
- Chaque instruction se terminent par un **;** et les blocs d'instructions sont définis à l'aide de **{** et **}**.
- On retrouve en C++ les instructions d'affectation, les conditionnelles et les boucles.

1.1.2 Premiers programmes en C++

Le programme, défini dans **PremierProgramme.cpp** ne contient que des affichages **cout<<** et des saisies **cin>>**.

```
#include<iostream>
using namespace std;

int main() {
 cout << " Premier Programme " << endl ; // affichage
 cout << " Entrer deux entiers " ;
 int x, y ; // déclaration de variables sans initialisation
 cin >> x >> y ; // lecture des deux valeurs des variables
 cout << " Produit " <<x * y << " Somme " <<x + y << endl ;
 // affichages des résultats
 return 0;
}
```

1. g++ -c PremierProgramme.cpp (compilation -> PremierProgramme.o)
2. g++ -o PremierProgramme PremierProgramme.o (édition des liens -> PremierProgramme).

ou simplement

g++ -o PremierProgramme PremierProgramme.cpp

ou encore

g++ PremierProgramme.cpp (a.out est l'exécutable).

Pour exécuter le programme : **./nomDeLExecutable** (./PremierProgramme ou ./a.out)

Utilisation d'une fonction simple :

```
#include<iostream>
using namespace std;

int somme(int n) {
 // somme des n premiers entiers : 1 + 2 + ... + n
 int som = 0;
 for(int i = 1; i < n + 1 ; ++i)
   som += i;
 return som;
}

int main() {
 cout<<"somme des 12 premiers entiers : "<<somme(12)<<endl;
 // somme des 12 premiers entiers : 78
}
```

— Vous noterez qu'en C++, à la différence de java (intrégralement objet), il n'est pas nécessaire de définir une classe.
— Les types C++ sont très proches de ceux de java : int, float, double, bool, char, string, ...
— Les fonctions (simples) sont définies avec une valeur de retour typée et 0, un ou plusieurs paramètre(s) typé(s).
— Utilisation de la boucle **for**.

1.1.3 Programme un peu plus complexe :

```
#include<iostream>
using namespace std;

int nbChiffre(int nombre) {
  // nombre de chiffres d'un nombre
  int nbChif;
  if (nombre == 0)
    nbChif = 1;
  else {
    nbChif = 0;
    while(nombre != 0) {
      nombre = nombre / 10;
      nbChif += 1; // ou encore ++nbChif;
    }
  }
  return nbChif;
}

int main() {
  int val;
  cout<<" saisir une valeur entière ";
  cin>> val;  // 543298
  cout<<" nombre de chiffres = "<< nbChiffre(val) <<endl; // 6
  return 0;
}
```

Utilisation de la boucle **while**, de l'écriture plus rapide **+=** et de la division **/**. En C++, la division entre deux entiers retourne un entier et la division de deux réels (float ou double) retourne un réel.

1.1.4 Les tableaux en C++ :

Les tableaux en C++ s'apparentent aux listes python, mais ils *ne peuvent pas s'agrandir ni diminuer en taille*. Il existe plusieurs façons de déclarer un tableau, on vous en présente deux :

1. int tab[10] ; déclare un tableaux de dix entiers (au maximum) non initialisé.
2. float tabF[5] = {3, 8.5, 5.5, -4, 12.3} ;

Les tableaux de taille fixe sont appelés **tableaux statiques** :

```
#include<iostream>
using namespace std;

#define MAX 10

int main() {
  int tab[MAX] = { ..... };
  // il est possible de mettre moins de 10 éléments
  // ceux qui ne sont pas initialisés valent 0.
  ...
  return 0;
}
```

— Si une fonction utilise un tableau en paramètre, on ne donne pas sa taille dans la déclaration, en revanche, il faut ajouter un paramètre qui fournit la taille du tableau. Il n'existe pas de fonction qui permette d'observer la taille d'un tableau statique (comme len(l) en python).

— Une fonction ne peut pas retourner un tableau.
— On désire calculer la somme des entiers positifs d'un tableau de réels :

```
#include<iostream>
using namespace std;
# define MAX 10

float sommePositifs(float t[], int taille) {
  // paramètres t (tableau) taille (sa taille)
  float som = 0.0;
  for(int i = 0; i < taille; ++i) // assez proche du range python
    if (t[i] > 0)
      som += t[i];
  return som;
}

int main() {
  float tab[MAX] = {3, -5.8, 5.5, 9.3, -6.3};
  cout<<" somme des positifs du tableau "<<sommePositifs(tab, MAX)<<endl;
  for(int i = 0; i< MAX; ++i)
    cout<<tab[i]<<" ";
  cout<<endl;
}
```

Les affichages sont :

```
somme des positifs du tableau 17.8
3 -5.8 5.5 9.3 -6.3 0 0 0 0 0
```

Vous noterez que les 5 éléments non définis dans tab valent 0.

Comment « remplir » un tableau C++ ?

Comme il n'est pas posible de retourner un tableau comme résultat de fonction, celle que nous allons définir retournera le type **void**. De même la méthode d'affichage d'un tableau (qui ne s'affiche pas directement avec un *cout*<<) retournera également un type void.

```
#include<iostream>
using namespace std;
# define MAX 10

void lecture(float tab[], int taille) {
  // lecture des éléments d'un tableau
  cout<<" saisie des éléments d'un tableau de taille "<<taille <<endl;
  for(int i = 0; i < taille; ++i) {
    cout<<"tab["<<i+1<<"] ";
    cin>>tab[i];
  }
}
```

```
void affiche(float tab[], int taille) {
  // affichage des éléments d'un tableau
  for(int i = 0; i < taille; ++i)
    cout<<tab[i]<<" ";
  cout<<endl;
}

int main() {
  lecture(tab, 5);
  affiche(tab, 5);
}
```

Montrer l'exécution du code

CHAPITRE 2

COURS2

2.1 PROGRAMMATION ORIENTEE OBJET : C++

2.1.1 1. Les classes en C++ :

Comme en java, une classe permet de définir un type (et de représenter une entité p.e.).

A la différence de java, l'encapsulation se fait par « blocs » : un bloc **private :** pouvant être suivi d'un « bloc » **public :** et de nouveau (si nécessaire) un « bloc » privé. L'**encapsulation** consiste à rendre les « parties » d'une classe soit privées, soit publiques (il existe un autre statut que nous verrons plus tard)

Nous allons examiner une classe **Personne** en java et la même en C++ :

```
class Personne {
   private String nom;
   private int age;
   private char sexe;

   public Personne()
   //un constructeur sans paramètre
   {
       this.nom = "";
       this.age = 0;
       this.sexe = 'F';
   }

   public Personne(String nom, int age, char sexe)
   // un constructeur avec trois paramètres
   {
       this.nom = nom;
       this.age = age;
       this.sexe = sexe;
   }
```

(suite sur la page suivante)

(suite de la page précédente)

```
   public String toString()
   // une méthode d'affichage
   {
        String res = "";
        if(this.sexe == 'F')
            res += "Mme ";
        else
            res += "Mr ";
        res += this.nom+ " âgé de "+ this.age + " an(s)";
        return res;
   }
}
```

```
class Executable  {
   public static void main(String [] args) {
        Personne p1 = new Personne();
        Personne p2 = new Personne("Appolinaire", 34, 'M');
        System.out.println(p1); // p1.toString()
        System.out.println(p2);
   }
}
```

```
#include<iostream>
using namespace std;

class Personne {
 private :   // bloc privé pour les attributs
   std::string nom;
   int age;
   char sexe;

 public : // bloc public pour les méthodes publiques
   Personne()
   {   // constructeur sans paramètre
     this->nom = "?";
     this->age = 0;
     this->sexe = 'F';
   }

   Personne(std::string nom, int age, char sexe)
   {  // constructeur à trois paramètres
     this-> nom = nom;
     this-> age = age;
     this-> sexe = sexe;
   }

   std::string toString() const
   { // méthode d'affichage (noter le décorateur const)
     std::string res = "";
     if(this->sexe == 'F')
       res += "Mme ";
     else
       res += "Mr ";
     res += this->nom+ " agé de " +std::to_string(this->age) +" an(s) \n";
     return res;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
};
```

```
int main() {
    Personne personne1("Appolinaire", 34, 'M');
    Personne personne2;
    cout<<personne1.toString()<<endl;
    cout<<personne2.toString()<<endl;
}
```

1. Vous notez tout d'abord le bloc **private :** dans lequel sont définis les attributs. Dans le bloc **public :** on trouve les deux **constructeurs** possibles (comme en java) ainsi qu'une méthode **affiche()**. On explicitera un peu plus tard le « décorateur » **const**.
2. La fin de la classe est matérialisée par un ; (suivant l'accolade fermante).
3. Comme en java, il est recommandé (si c'est nécessaire) de définir des getters et setters.
4. Les blocs *private :* et *public :* peuvent être positionnés n'importe où dans le code de la classe (non recommandé) et il peut y en avoir alternativement plusieurs.
5. Comme en java, une classe C++ peut avoir plusieurs *constructeurs*.
6. L'instance de classe en python est **self**, celle de java est **this**. et celle de c++ est `*this`. Attention là, en C++, il s'agit bien d'une **référence** ou pointeur. L'écriture `*this.` se réécrit **this->**.
7. Le type chaîne de caractères en C++ est un type objet, on peut importer la bibliothèque le contenant ou simplement écrire std : :string. (#include<string>).
8. La méthode

```
std::to_string(this->age)
```

permet de transformer une valeur en chaîne de caractères.

La déclaration d'une instance de Personne en java :

```
Personne pers = new Personne("Martin", 33, 'F');
```

correspond à la déclaration en C++ :

```
Personne pers("Martin", 33, 'F');
```

Attention en java une instance de **Personne** est un pointeur (correspond à une adresse en mémoire dynamique ou tas), en C++ une instance de **Personne** n'est pas un pointeur et elle est définie dans une mémoire appelée **mémoire statique**.

Note : Les pointeurs existent en C++ mais nous les verrons beaucoup plus tard.

2.1.2 2. La surcharge / redéfinition d'opérateurs :

On a vu en java que l'opérateur **==** ne gérait pas l'égalité des valeurs mais celles des références. En C++, il est possible (ce n'est pas le cas en java) de surcharger ou de redéfinir n'importe quel opérateur (sauf certains très spécifiques comme new . . .).

Ainsi, on peut très simplement tester l'égalité **==** de deux instances de Personne en redéfinissant l'opérateur **==**.

Dans la classe Personne on rajoute une méthode **==** :

```
class Personne {
 // ....
 bool operator == (Personne personne)
 { // surcharge de ==
    return this->nom == personne.nom &&
           this->age == personne.age &&
           this->sexe == personne.sexe;
 }
};
```

1. La surcharge ou redéfinition se fait en utilisant le mot clef **operator**.
2. Comme **==** est une méthode de classe, l'instance `*this` existe, ce qui implique que l'opérateur compare les divers attributs de l'instance `*this` et d'une seconde instance de la classe Personne (passée en paramètre).
3. L'utilisation de l'égalité se fait de façon stantard (écriture standard), même si elle peut s'utiliser comme une méthode de classe :

On suppose l'existence de deux instances de Personne *personne1* et *personne2*.

```
if(personne1 == personne2) ...  // écriture standard

// ou encore

if(personne1.operator == (personne2)) ... // méthode de classe
```

Comment faire des affichages à la *toString()* ?

On aimerait pouvoir écrire une instruction telle que : **cout<<personne1<<endl ;**. Il faut pour cela surcharger l'opérateur **<<**.

Attention la surcharge de l'opérateur **<<** dans le cadre d'un affichage *ne peut pas être une méthode de classe*. Il faut le surcharger à l'extérieur de la classe.

```
ostream & operator <<(ostream & sortie, const Personne & personne) {
  sortie<<personne.toString();
  return sortie;
}
```

On utilise ici la méthode **toString()** de la classe Personne.

NB : le paramètre de l'opérateur est **const Personne &** la méthode toString() de la classe doit également être **const**.

On reviendra sur ce décorateur plus tard.

2.1.3 3. Des méthodes spécifiques à C++ :

Java gère lui-même ses objets en utilisant un garbagge collector : toute instance java est une référence. Aucun travail de récupération mémoire n'est nécessaire.

C++ ne gère pas lui-même sa mémoire dynamique (pour les pointeurs non encore vus). C'est pourquoi il doit y avoir en C++ une méthode qui **détruit** les instances de classes (lorsque celles-ci *meurent*).

1. Si ces instances sont statiques, C++ s'en charge seul. Le **destructeur** existe par défaut. (Nos deux instances personne1 et personne2 seront détruites).
2. Si ces instances sont dynamiques (cf cours pointeur), il **faudra** définir, utiliser et appeler le destructeur de la classe.

En C++ le destructeur d'une classe se nomme toujours **~NomDeLaClasse()**.

Reprenons la classe **Personne** et ajoutons lui le code suivant :

```
~Personne() {cout<<"destructeur de Personne"<<endl;}
```

```
int main() {
   Personne personne1("Appolinaire", 34, 'M');
   Personne personne2;
   cout<<personne1<<endl;
   cout<<personne2<<endl;
}
```

Les affichages obtenus sont :

```
Mr Appolinaire agé de 34 an(s)
Mme ? agé de 0 an(s)
destructeur de Personne
destructeur de Personne
```

Vous noterez les deux affichages du **destructeur** indiquant que celui-ci est sytématiquement appelé quand les instances *meurent* (ici fin de portée du main).

CHAPITRE 3

COURS3

3.1 PROGRAMMATION ORIENTEE OBJET : C++

3.1.1 Introduction aux listes de la STL list :

Une classe **Personne** et sa surcharge de l'opérateur **<<**.

```
class Personne {
  string nom;
  int age;
public :
  Personne(string nom = "Dupond", int age = 10) {
    // constructeur avec initialisation par défaut
    this-> nom = nom;
    this-> age = age;
  }
  int getAge() {return this->age;}
  string getNom() {return this->nom;}
  string toString() {
    return this->nom +" âgé de "+ to_string(this->age) + " ans ";
  }
};
```

```
ostream & operator <<(ostream & s, Personne pers) {
  s<<pers.toString();
  return s;
}
```

La S(tandard) T(emplate) L(ibrary) comporte plusieurs classes spécifiques permettant de gérer des listes. La première qui sera présentée est la classe **list**. Il s'agit d'un **conteneur**. Tous les conteneurs possèdent des méthodes identiques telles que *size()*, *push_back() front() back()* ....

```
#include<iostream>
#include<string>
#include<list>
using namespace std;

class ListePersonnes {
  list<Personne> personnes;
public :
  ListePersonnes() {}
  void add(Personne personne) {
    this->personnes.push_back(personne);
  }
  string toString() { // à définir et utiliser
    string res = "[";
    for(Personne pers : this->personnes)
      res += pers.toString();
    res+="]";
    return res;
  }
  Personne doyen() { // le plus âgé
    int age = 0;
    Personne persM;
    // attention ici constructeur sans paramètre : il doit exister
    for(Personne pers : this->personnes) //parcours à la java
      if(pers.getAge() > age) {
        age = pers.getAge();
        persM = pers;
      }
    return persM;
  }
  bool estTriee() { // liste triée par âge croissant
    if(this-> personne.size()) return true;
    Personne prec = this->personnes.front();
    for(Personne pers : this->personnes)
      if(pers.getAge() < prec.getAge())
        return false;
    return true;
  }
};
```

```
ostream & operator <<(ostream & s, ListePersonnes liste) {
  s<<liste.toString();
  return s;
}
```

Vous noterez qu'il est indispensable (au contraire de java) que la classe Personne possède un constructeur sans parmètre. On peut soit écrire plusieurs constructeurs soit définir des paramètres par défaut (en C++).

Il n'existe pas comme en java, une méthode permettant d'afficher complètement les éléments des divers conteneurs. Il faut écrire une méthode toString() puis surcharger l'opérateur <<, si nécessaire.

La méthode d'ajout en fin de liste est appelée *push_back()*. Le parcours d'un conteneur peut se faire de plusieurs manières, en pariculier sous la forme :

```
for(Type elem : this->liste)
   .....
```

Certaines méthodes comme front() qui fournit le premier élement d'une liste et back() qui fournit le dernier, sont

communes à tous les conteneurs.

3.1.2 La classe vector de la STL :

Nous allons reprendre le même exercice mais cette fois-ci avec un vecteur de la STL.

```
class VecteurPersonnes {
   vector<Personne> personnes;
 public :
   VecteurPersonnes() {}
   void add(Personne personne) {
     this->personnes.push_back(personne);
   }
   string toString() { // à définir et utiliser
     string res = "[";
     for(Personne pers : this->personnes)
       res += pers.toString();
     res+="]";
     return res;
   }
   Personne laPlusJeune() {
     int age = this->personnes.front().getAge(), ag;
     Personne personne;
     // parcours par indice
     for(int i = 1; i < this->personnes.size(); ++i) {
       ag = this->personnes[i].getAge();
       if(ag < age) {
         age = ag;
         personne = this->personnes[i];
       }
     }
     return personne;
   }
   int minimum(int deb) {
     int ind = deb;
     for(int i = deb + 1; i < this->personnes.size(); ++i)
       if(this->personnes[i].getAge() < this->personnes[ind].getAge())
         ind = i;
     return ind;
   }
   void trier() {
     int ind;
     Personne pers;
     for(int i = 0; i < this->personnes.size(); ++i) {
       ind = this->minimum(i);
       pers = this->personnes[i];
       this->personnes[i] = this->personnes[ind];
       this->personnes[ind] = pers;
     }
   }
 };
```

```
ostream & operator <<(ostream & s, VecteurPersonnes personnes) {
  s<<personnes.toString()<<endl;
  return s;
}
```

On note que le conteneur **vector** permet un parcours à la fois par élément et par indice. On utilise ici les [et]. Attention, il faut que le vecteur ne soit pas vide (qu'il ait été rempli à l'aide de push_back()) afin de pouvoir utiliser les [et].

CHAPITRE 4

COURS4

4.1 PROGRAMMATION ORIENTEE OBJET : C++

4.1.1 Les pointeurs C++ Introduction

Un pointeur est en tout premier lieu une adresse en mémoire (statique ou dynamique).

Pour déclarer un pointeur on commence par donner le type de l'objet pointé suivi du symbole $\star$.

Soit par exemple : int * ptr ; ou char * ptrc ; float * ptrf ; Couple * ptrC ; voire même int **ptrPtr ;

ptr est un pointeur sur un entier ptrc est un pointeur sur un caractère ptrf est un pointeur sur un float ptrC est un pointeur sur un Couple et ptrPtr est un pointeur (ici probablement un tableau) dont les éléments sont des poiteurs sur entiers.

Lorsque ces divers pointeurs existeront (auront une valeur) :

l'objet pointé par ptr sera un entier l'objet pointé par ptrc sera un caractère l'objet pointé par ptrf sera un réel l'objet pointé par ptrC sera un Couple et l'objet pointé sera un tableau (probablement) de pointeurs sur entiers.

Voici un exemple simple :

```
int main() {
 int * pointeurSurEntier = nullptr; //  pointeur nul
 pointeurSurEntier = new int;
 cout<<pointeurSurEntier<<endl;
}
```

On note bien le type : **pointeur sur un entier** ; l'affichage donne une adresse mémoire : 0x2440c20. Cela signifie qu'à cette adresse (en mémoire dynamique ici à cause du **new**) il y a une place réservée pour stocker un *entier* (attention à cette place il n'y a aucune valeur d'entier).

Comment donner une valeur à un pointeur ? sachant que cette valeur ne peut être qu'une adresse.

En C++ il existe deux solutions pour donner une valeur à un pointeur :

1. Soit en donnant une adresse existante (par exemple d'un entier qui a déjà été déclaré). Tout se passe généralement en mémoire statique.

2. Soit par allocation dynamique en utilisant l'opérateur **new**. Ce qui a été vu dans le petit exemple (et que nous expliciterons bien sûr). Ceci concerne la mémoire dynamique (ou tas).

4.1.2 Les Pointeurs et adresses (sans allocation dynamique)

On voit ici comment on donne une valeur (adresse) à un pointeur en utilisant une adresse existante. Cela a l'inconvénient d'associer à un pointeur une variable qui peut fournir son adresse. On note que tout ceci se passe en mémoire **statique**.

```
int main() {
 int * ptrEnt = nullptr;
 int entier (5);
 cout<<&entier<<endl; // 0x7ffc6fa22acc
 ptrEnt = & entier;
}
```

Ici on a un pointeur sur l'entier *entier* c'est-à-dire que l'adresse de la variable *entier* dans la mémoire statique est la même que celle du pointeur. La valeur du pointeur est donc l'adresse de l'entier. Pour afficher la valeur de *entier* on a alors deux solutions, soit :

```
cout<<entier<<endl;
```

qui affichera 5

soit encore

```
cout<<*ptrEnt<<endl;
```

qui affichera également 5.

L'expression `*ptrEnt` est appelée *valeur (ou objet) pointée* ici c'est la valeur de *entier*. Cette expression est modifiable (par affectation).

Ainsi :

```
*ptrEnt = -2;
cout<<entier<<endl;
```

affichera -2.

Des exemples

```
#include<iostream>
using namespace std;
struct Couple { // attributs publics
  int x;
  char c;
} ;

struct AutreCouple{
  int * ptrX;
  int * ptrY;
};
```

```
int main() {
  int * p = nullptr;
  int x;
  p = &x; // p et x ont la même adresse
  cout<<p<<endl; // adresse où se trouve x (sans valeur actuellement) en mémoire  //␣
↪0x7fff1113507c (peut être ?)
  cout<<x<<endl;  // 22025 (peut être ?) x non défini !!!
  *p = 5;
  cout<<x<<endl; // 5
  // &x = 0x7fff1113507c; // (impossible erreur de compilation &x est statique)
  // error: lvalue required as left operand of assignment
  //        &x = 0x7fff1113507c;
  Couple couple; // une structure
  couple.x = *p;
  couple.c = (char)(97);
  // cout<<couple<<endl; (impossible) ou alors surcharger <<
  cout<<"couple "<<couple.x<<" "<<couple.c<<endl;
  //                  5              'a'
  Couple * ptrCouple = nullptr;
  ptrCouple = &couple;
  ptrCouple->x = *p - 3;
  ptrCouple->c = (*p) * 120;
  cout<<"couple "<<couple.x<<" "<<couple.c<<endl;
  // couple          5               X
  AutreCouple autreCouple;
  AutreCouple * ptrAutreCouple = nullptr;
  autreCouple.ptrX = &x;
  autreCouple.ptrY = p;

  // que se passe-t-il si on écrit ceci ?
  // *(*ptrAutreCouple).ptrX = -3;
  // erreur de segmentation : le pointeur ptrAutreCouple n'ayant pas de valeur
  // on ne peut accéder à l'objet pointé!!!
  ptrAutreCouple = &autreCouple;
  *(*ptrAutreCouple).ptrX = -3; // s'écrit également
  //                              * ptrAutreCouple->ptrX = -3;
  cout<<"x = "<<x<<"*p = "<<*p<<endl; // -3
}
```

Un peu plus complexe

```
class Couple {
  int premier;
  int second;
public :
  Couple() {this-> premier = this -> second = 0;}
  Couple(int premier, int second) {
    this-> premier = premier;
    this -> second = second;
  }
  int getPremier() const {return this->premier;}
  int getSecond() const {return this->second;}
  void setPremier(int premier) {this->premier = premier;}
  void setSecond(int second) {this->second = second;}
  std::string toString() const {
    std::string res = "";
```

(suite sur la page suivante)

(suite de la page précédente)

```
    res += "("+ std::to_string(this->premier) + ", "+ std::to_string(this->second) +
↪") ";
    return res;
  }
};

ostream & operator <<(ostream & s, const Couple & c) {
   s<<c.toString()<<" ";
   return s;
}
```

```
int main() {
  Couple couple(4, -2);
  cout<<couple.toString()<<endl;
  // (4, -2)
  Couple *ptrCouple = nullptr;
  ptrCouple = &couple ;
  (*ptrCouple).setPremier(5);
  cout<<couple<<endl;
  // (5, -2)
  (*ptrCouple).setSecond((*ptrCouple).getSecond() + 2);
  cout<<couple<<endl;
  //(5, 0)
}
```

NB : Faire ici les dessins mémoire (statique)

Les lignes :

```
(*ptrCouple).setPremier(5);
(*ptrCouple).setSecond((*ptrCouple).getSecond() + 2);
```

peuvent également s'écrire :

```
ptrCouple -> setPremier(5);
ptrCouple -> setSecond(ptrCouple -> getSecond() + 2);
```

4.1.3 Pointeurs et allocation dynamique

Il n'est pas avantageux d'utiliser des variables ou instances pour donner des valeurs aux pointeurs (un entier pour un int *, un Couple pour un Couple *, . . .)

Il est donc possible en C++ d'utiliser l'opérateur **new** pour allouer dynamiquement de la mémoire (à l'exécution) dans le **tas**. (Le tas a beaucoup plus de place que la mémoire statique)

```
int * prt = new int;
*ptr = 35;
```

L'opérateur **new** se comporte de la façon suivante :

1. il recherche dans le tas un place disponible pour stocker un objet du type de l'objet pointé (ici un int).
2. s'il trouve de la place, il fournit au pointeur l'adresse de début de la zone de stockage.
3. s'il ne trouve pas de place il fournit au pointeur la valeur *nullptr*

Pour le code précédent on peut également écrire (et c'est plus efficace) :

```
int * prt = new int(-5);
```

La démarche est la même et si une place est trouvée donc réservée, alors il y stocke la valeur (-5 ici).

Comme le *garbage collector* n'existe pas en C++, la place qui a été réservée dans le tas doit être explicitement rendue au tas (désallouée).

L'opérateur qui s'en charge est **delete** :

```
delete ptr;
ptr = nullptr;
```

On *détruit* le pointeur (rend au tas la place occupée pour une future utilisation) et il est recommandé de remettre le pointeur à **nullptr** pour un nouvelle allocation.

Attention l'opérateur **delete** ne peut être utilisé sur un pointeur que si l'opérateur **new** a lui même été utilisé sur celui-ci !

On peut reprendre un partie du code précédent (statique)

```
#include<iostream>
using namespace std;
struct Couple {
  int x;
  char c;
} ;

struct AutreCouple{
  int * ptrX;
  int * ptrY;
};
```

```
int main() {
   int * p = new int;
   *p = 5;
   cout<<"p = "<<p<<endl;
   //     p =    0x564f4bc41e70 (par exemple)
   Couple * ptrCouple = new Couple; // constructeur par défaut
   ptrCouple->x = *p - 3;
   ptrCouple->c = (*p) * 120;
   cout<<"ptrCouple->x "<<ptrCouple->x<<" ptrCouple->c "<<ptrCouple->c<<endl;
   // ptrCouple->x          2             ptrCouple->c        'X'

   AutreCouple * ptrAutreCouple;
   // que se passe-t-il si on écrit ceci ?
   // ptrAutreCouple -> ptrX = new int(-3);
   // erreur de segmentation : le pointeur ptrAutreCouple n'ayant pas de valeur
   // on ne peut accéder à l'objet pointé!!!
   ptrAutreCouple = new AutreCouple; // constructeur par défaut
   ptrAutreCouple ->ptrX = new int(-3);
   cout<<"*(ptrAutreCouple -> ptrX) "<<*(ptrAutreCouple -> ptrX)<<endl;
   // *(ptrAutreCouple -> ptrX)              -3
   // attention on ne peut accéder à *(ptrAutreCouple -> ptrY) !!
}
```

4.1.4 Passage par valeur (ou copie) passage par référence (ou adresse)

Il existe en C++ deux types de passage de paramètres : par valeur ou copie (celui de python et java) et celui par référence ou adresse (permet de retourner entre autre plusieurs valeurs de retour).

Un exemple simple :

```
int parValeur(int x, char c) {
  ++x;
  c += 1;
  cout<<"fonction parValeur : x "<<x<<" c "<<c<<endl;
  // fonction parValeur : x 7 c 'd'
  return x + (int) c; // 7 + 100 = 107
}
```

Dans la fonction, la valeur du paramètre x est incrémentée de 1 et celle du caractère également (i.e. caractère suivant).

```
void parReference(int &x, char &c) {
   ++x;
   c += 1;
   cout<<"fonction parReference : x "<<x<<" c "<<c<<endl;
   // fonction parReference : x 7 c 'd'
}
```

De nouveau les mêmes résultats.

```
int main() {
  int entier(6);
  char caractere('c');

  int z = parValeur(entier, caractere);
  cout<<"Programme principal (parValeur) : z "<<z<<" entier "<<entier<<" caractere "<
↪<caractere<<endl;
  // Programme principal (parValeur) : z 107 entier 6 caractere c

  parReference(entier, caractere);
  cout<<"Programme principal (parReference) : entier "<<entier<<" caractere "<
↪<caractere<<endl;
  // Programme principal (parReference) : entier 7 caractere 'd'
}
```

Après l'appel de la fonction avec passage de paramètres par valeur, entier et caractere sont inchangés, en revanche après l'appel de fonction avec passage de paramètres par référence, les deux variables entier et caractere sont modifiées (7 et d).

Le passage de paramètres par référence est souvent utilisé pour définir une fonction avec plusieurs valeurs de retour (ou résultats). Prenons l'exemple simple d'un tableau d'entiers dont on veut connaître la valeur du minimum et l'indice de celui-ci.

```
void minIndice(int t[], int taille, int & minimum, int & indice) {
  // deux passages par référence (résultats)
  indice = 0;
  for(int i(1); i < taille; ++i)
    if(t[i] < t[indice])
      indice = i;
  minimum = t[indice]; // mise à jour du minimum
}
```

Des fonctions annexes

```
void lire(int t[], int taille) {
  cout<<" saisie des "<<taille<<" éléments du tableau "<<endl;
  for(int i(0); i < taille; ++i) {
    cout<<i+1<<"ème ";
    cin>>t[i];
  }
}
```

```
void affiche(int t[], int taille) {
  for(int i(0); i < taille; ++i)
    cout<<t[i]<<" ";
  cout<<endl;
}
```

Programme principal

```
#define Max 5

int main() {
  int tab[Max];
  lire(tab, Max);
  affiche(tab, Max);
  int mini, indiceMin;
  minIndice(tab, Max, mini, indiceMin);
  cout<<" minimum "<<mini<<" indice "<<indiceMin<<endl;
```

}

CHAPITRE 5

COURS5

5.1 PROGRAMMATION ORIENTEE OBJET : C++

5.1.1 1. L'Héritage

L'héritage est un des atouts de la programmation orientée objet. Il permet de réutiliser du code déjà écrit. En C++ il existe 3 formes d'héritage :

1. héritage **public** (le seul permettant le **polymorphisme**)
2. héritage **privé** (utilisé souvent pour implémenter une classe par héritage sans pouvoir utiliser les méthodes de l'ancêtre).
3. héritage **protégé** (pas d'utilisation connue)

On écrit :

```
class ClasseDerivee : [public | private | protected ] ClasseAncetre {
   // ....
};
```

— par défaut l'héritage est **privé**
— l'héritage utilisé dans la polymorphisme est l'héritage **public** (on verra plus tard)
— l'héritage peut être **multiple** en C++ (au contraire de java)

Un exemple simple

— Une classe **Animal**
— Un héritier **Chien**
— Un héritier **Chat**

```
#include<iostream>
using namespace std;
class Animal {
```

(suite sur la page suivante)

(suite de la page précédente)

```
   std::string espece;
public :
  Animal(std::string espece = "?") : espece(espece) {}
  ostream & affiche(ostream & s) const {
    return s<<"Animal d'espèce "<<this->espece<<" ";
  }
  std::string getEspece() const {return this->espece;}
  ~Animal() {cout<<"destruction Animal "<<endl;}
};

ostream & operator <<(ostream &s, const Animal &a) {
  return a.affiche(s);
}
```

Les deux héritiers seront des héritiers publics (Chien et Chat)

```
class Chien : public Animal {
   std::string nomMaitre;
 public :
   Chien(std::string nom ="?") : Animal("mammifère"), nomMaitre(nom){}
   ostream & affiche(ostream & s) const {


   }
   std::string getMaitre() const {return nomMaitre;}
   ~Chien() {                                          }
};

ostream & operator <<(ostream &s, const Chien &c) {
   return c.affiche(s);
}
```

Classe **Chat**

```
class Chat : public Animal {
   std::string couleurPelage;
public :
   Chat(std::string pelage ="?");
   ostream & affiche(ostream & s) const {



   }
   std::string getCouleur() const {return couleurPelage;}
   ~Chat() {                                          }
};

ostream & operator <<(ostream &s, const Chat &c) {
   return c.affiche(s);
}
```

Puis le main

```
int main() {
   Animal animal("mammifère");
   Chien chien("Arthur");
   Chat chat("roux");
```

(suite sur la page suivante)

(suite de la page précédente)

```
   cout<<animal<<endl;
   cout<<chien;
   cout<<chat;
}
```

Les affichages

```
à vous
```

Plusieurs remarques :

- toutes les méthodes **publiques** de l'ancêtre sont accessibles dans les héritiers.
- dans chaque héritier, l'attribut (ou les) de l'ancêtre existe(nt) mais n'est(ne sont) disponible(s) que via la méthode **getEspece()** ou de façon générale, les getters. C'est normal les attributs dans l'ancêtre sont privés.
- Pour construire un héritier, on construit d'abord la partie ancêtre puis la partie héritière.

```
Chien(std::string nom ="?") : Animal("mammifère"), nomMaitre(nom {}
// constructeur de Chien    : constructeur de Animal
```

- On notera l'utilisation des **:** ainsi que l'appel au constructeur **Animal**.
- Pour utiliser une méthode de l'ancêtre dans l'héritier, si celle-ci porte le même nom dans l'ancêtre et l'héritier, il suffit d'ajouter l'opérateur de résolution de portée **Animal : :** devant le nom de la méthode (cf. méthode *affiche*)
- En ajoutant les destructeurs, on constate dans le main que le **destructeur** d'un héritier appelle automatiquement (et en premier) le **destructeur** de l'ancêtre.

A vous de jouer

1. Essayer de définir un tableau d'Animal contenant un Animal, un Chien, une Chat(Animal TabAnimal[3]). Faites des affichages. Que constatez vous ?
2. Essayer de définir un tableau de pointeurs sur Animal (Animal * TabPAnimal[3])et stockez y un pointeur sur Animal Chien et Chat. Faites des affichages. Que constatez vous ?
3. Ajoutez devant les méthodes **affiche** de chaque classe le mot clef **virtual**. Réessayez de tester les 2 programmes précédents ? Que constatez vous ?
4. Ne conservez que la surcharge de l'opérateur **<<** de la classe Animal. Testez tous vos programmes. Que constatez vous ?

CHAPITRE 6

TD TP C++

6.1 Semaine 1

6.2 TD1

6.2.1 1. Moyenne :

— Ecrivez un programme permettant de calculer la moyenne de N (saisie clavier) notes.
— Modifiez ce programme pour que N soit positif et que les notes saisies soient bien comprises entre 0 et 20.

6.2.2 2. Inverser :

— Définir une fonction de profil **int inverser(int)** permettant d'inverser les chiffres d'un nombre : par exemple *inverser(2345)* vaut *5432*.
— Définissez un programme principal testant cette fonction.

6.2.3 3. Somme en base 3 :

— Définir une fonction **int sommeBase3(int, int)** - puis un programme de test - qui étant donnés deux entiers écrits en base3 calcule leur somme en base 3. Par exemple : *2210 + 112 = 10022*

6.2.4 4. Nombres premiers et nombres parfaits :

— Écrire une fonction testant si un entier est premier ou non.
— Écrire une fonction qui détermine si un nombre est parfait ou non (la somme de ses diviseurs stricts est égale au nombre)
— Définissez un programme principal testant si un nombre est premier et fournissant tous les nombres parfaits inférieurs à 100.

6.2.5 5. Minimum des éléments d'un tableau et fréquence d'un entier :

— Définir la fonction **int minimum(int tab[], int taille)** qui retournez le minimum des éléments d'un tableau.
— Définir une fonction calculant le nombre d'occurrences d'un entier dans un tableau d'entiers.

6.2.6 6. Appartenance à un tableau :

— Écrire une fonction retournant un booléen indiquant si un entier appartient ou non à un tableau d'entiers.
— Modifiez cette fonction pour qu'elle retourne -1, si l'élément n'appartient pas et l'indice de celui-ci sinon.
— Supposez que le tableau passé en paramètre soit trié par ordre croissant, utilisez l'algorithme dichotomique pour retourner -1 ou l'indice.

6.3 Semaine 2

6.4 TD2

6.4.1 Classe Couple :

— Définir une classe Couple permettant d'afficher les messages du programme principal suivant :

```
int main() {
 Couple couple, couple1(4, -2), couple2(-1, 0);
 cout<<c<<" "<<couple1<<" "<<couple2<<endl;
 couple = couple1 + couple2;
 cout<<" couple1 + couple2 "<<couple<<endl;
 couple.setPrem(66);
 couple.setSec(-15);
 cout<<" couple "<<couple<<std::endl;
}
```

— Ajouter un destructeur afin de visualiser la destruction des 3 couples.
— Ajouter une surcharge de l'opérateur **==** testant l'égalité de deux couples.
— Modifier votre programme principal pour tester l'égalité de deux instances de Couple.

6.4.2 Classe Complexe :

— Même exercice pour le programme principal suivant utilisant une classe **Complexe**.

```
int main() {
 Complexe c1(1.0f, 2.5f), c2(-1.5f, 3.0f);
 cout<<c1<<" "<<c2<<" "<<"c1 + c2 "<<c1+c2<<" c1 - c2 "<<c1 - c2<<" c1 * c2 "<<c1 * c2
↪<<endl;
 cout<<c1(1)<<endl;   // 1.0f
 cout<<c1(2)<<endl;   // 2.5f
 return 0;
}
```

— Vous surchargerez l'opérateur () pour représenter les deux getters donnant la partie réelle et la partie imaginaire de deux complexes.
— N'oubliez pas le destructeur.

6.4.3 Classe Date :

Complétez la classe **Date** suivante :

```
#include<iostream>
using namespace std;

class Date {
private :
   int jour, mois, an;
public :
  Date(int j, int m, int a) ;
  bool bissextile() ;
```

(suite sur la page suivante)

(suite de la page précédente)

```
  int nbJourMois() ;
  bool valide() ;
  std::string  toString() const ;
};
```

— Ajoutez une surcharge de l'opérateur **<<**
— Ecrivez un programme principal testant votre classe et ses méthodes.

6.4.4 Week end entre amis (Bonus) :

Reprenez l'exercice vu en java sur le weekend entre amis et essayez de le *traduire* en C++. Créez les classes **Personne**, **Depense**

6.5 Semaine 4

6.6 C++ TD3

6.6.1 Fantôme

Définir une classe **Fantome** permettant l'exécutable suivant :

```
int main() {
  Fantome lemure("Lémure", "romaine", 5);
  Fantome willis("Willis", "slave", 2);
  Fantome mau("Mau", "égyptienne");
  cout<<"Des Fantomes : "<<lemure<< " "<<mau<<endl;
}
// Des Fantomes : Fantome Lémure d'origine romaine de nuisance 5
//                Fantome Mau d'origine égyptienne
```

Par défaut la nuisance vaut -1. Ajoutez les getters et setters ainsi qu'une surcharge de l'opérateur << utilisant une méthode toString(). Vous serez probablement amenés à modidifier cette classe en ajoutant certaines méthodes par la suite.

6.6.2 Armée :

Une armée de fantômes est une **liste** de Fantome(s). Définir la classe **Armee**.

- Ajoutez la méthode **enrole** qui permet d'ajouter un Fantome à l'armée.
- Ajoutez la méthode **enroleSpecial** qui permet d'enrôler un fantôme dans l'armée, uniquement si la nuisance de celui-ci n'est pas déjà une nuisance d'un fantôme déjà présent dans l'armée.
- Ajoutez une méthode toString() et une surcharge de l'opérateur <<.
- Ajoutez la méthode **leMoinsNuisible** qui retourne le fantôme de l'armée ayant la nuisance la plus faible (!= -1).
- Définir la méthode **nuisancePlusDe** qui retourne la liste des fantômes dont la nuisance est supérieure ou égale à une nuisance donnée.

Le programme principal suivant doit fournir les affichages proposés.

```
int main() {
   Fantome lemure("Lémure", "romain", 5);
   Fantome willis("Willis", "slave", 2);
   Fantome mau("Mau", "égyptienne");
   cout<<"des Fantomes : -> "<<lemure<< " "<<mau<<endl;
   Armee armeeFantomes;
   armeeFantomes.enrole(lemure);
   armeeFantomes.enrole(willis);
   armeeFantomes.enrole(mau);
   cout<<"Armee enrôlement : -> "<<armeeFantomes<<endl;
   // Armee enrôlement : -> Armée de Fantomes :
   //       Fantome Lémure d'origine romaine de nuisance 5
   //       Fantome Willis d'origine slave de nuisance 2
   //       Fantome Mau d'origine égyptienne

   armeeFantomes.enroleSpecial(Fantome("Inanna", "mésopotamienne", 7));
   armeeFantomes.enroleSpecial(Fantome("Charon", "grecque", 2));
   cout<<"Enrôlement spécial Armee : -> " <<armeeFantomes<<endl;
   // Enrôlement spécial Armee : -> Armée de Fantomes :
```

(suite sur la page suivante)

(suite de la page précédente)

```
  //      Fantome Lémure d'origine romaine de nuisance 5
  //      Fantome Willis d'origine slave de nuisance 2
  //      Fantome Mau d'origine égyptienne
  //      Fantome Inanna d'origine mésopotamienne de nuisance 7

  cout<<"Le moins nuisible : -> "<<armeeFantomes.leMoinsNuisible()<<endl;
  // Le moins nuisible : -> Fantome Willis d'origine slave de nuisance 2
}
```

— Que devez vous écrire dans votre main afin de pouvoir afficher tous les fantômes de nuisance >= à 3 ?

6.7 Semaine 5

6.8 TD4

6.8.1 Quelques algorythmes : Passage par valeur et par référence

Définir une fonction **nbreChiffresMult3** prenant en entrée un entier et fournissant son nombre de chiffres et s'il est ou non multiple de 3 remplacer les ... par les paramètres nécessaires et écrivez le code Pensez au cas nul.

```
void nbreChiffresMult3(...                                          ) {









}
```

Voici une fonction de tests

```
void main1() {
  cout<<" Entier : nombre de chiffres et multiple de 3 "<<endl;
  int entier, nbChif;
  bool mult3;
  cout<<" saisir un entier "<<endl;
  cin>> entier;
  nbreChiffresMult3(entier, nbChif, mult3) ;
  cout<<entier<<" a "<<nbChif<<" chiffres ";
  if(mult3)
    cout<<" et est multiple de 3"<<endl;
  else
    cout<<" et n'est pas multiple de 3"<<endl;
}
```

on vous fournit les deux fonctions lire et affiche suivantes :

```
void lire(int * tab, int taille) {
  cout<<" saisie des élements du tableau "<<endl;
  for(int i = 0; i < taille; ++i) {
    cout<<i<<"ème élément : ";
    cin>>tab[i];
  }
}
```

```
void affiche(int * tab, int taille) {
  cout<<"[";
  for(int i = 0; i < taille; ++i)
    cout<<tab[i]<< " ";
```

(suite sur la page suivante)

(suite de la page précédente)

```
  cout<<"] "<<endl;
}
```

Ecrire une fonction **sommeTabPosNegNulMult5** prenant en entrée un tableau d'entiers et sa taille et fournissant le nombre d'éléments positifs, négatifs ou nuls du tableau ainsi que le nombre d'éléments multiples de 5. On vous donne les deux paramètres d'entrée complétez ensuite les paramètres et écrivez le code

```
void sommeTabPosNegNulMult5(int * tab, int taille,
↪                ) {









}
```

Puis uen nouvelles fonction de tests :

```
void main2() {
  cout<<" Tableau : somme, nombre de positifs, néf=gatifs et nuls et nombre de␣
↪multiples de 5 "<<endl;
  int tab[NB];
  int somme, pos, neg, nul, nbMult5;
  lire(tab, 5);
  affiche(tab, 5);
  sommeTabPosNegNulMult5(tab, 5, somme, pos, neg, nul, nbMult5);
  cout<<" somme "<<somme<<" nb positifs "<<pos<<" nb négatifs "<<neg<<" nb nuls "<<nul
↪<<" nb multiples de 5 "<<nbMult5<<endl;
}
```

6.8.2 Travail sur les Pointeurs

On vous propose de définir une classe Entier dont l'attribut est un pointeur sur un entier. Complétez

```
class Entier {
  int * val;
public :
  Entier(int valeur = 0) {                                    }
  int getVal() {                                            }
  void setVal(int valeur) {                                 }
  string toString() {                                     }
};
```

```
ostream & operator <<(ostream & s, Entier entier) {
  s<<entier.toString()<<" ";
  return s;
}
```

```
void main3() {
  cout<<" classe Entier et pointeur "<<endl;
  Entier ent(5);
  cout<<"entier "<<ent<<" valeur "<<ent.getVal()<<endl;
  ent.setVal(-7);
  Entier autre; // (1)
  autre = ent;
  autre.setVal(-2);
  cout<<"entier "<<ent<<" autre entier "<<autre<<endl;
}
```

6.9 Semaine 6

6.10 TD5

6.10.1 Classe Entier simple

```
#include<iostream>
#include<string>
using namespace std;

class Entier {
  int val;
public :
  Entier(int val = 0) : val(val) {}
  // ici c'est la meilleure façon d'écrire le constructeur
  // idem (mais moins couteux) que : {this -> val = val;}

  Entier(const Entier &ent) : val(ent.val) {
    cout<<" Constructeur de recopie de "<<ent.val<<endl;
  }
  // constructeur de recopie : c'est son profil
  // équivalent à {this -> val = ent.val;}

  ~Entier() {cout<<"Destructeur d'Entier de valeur "<<this->val<<endl;}
  // Le destructeur ici ne fait rien car uniquement des données statiques

  int getVal() const {return this-> val;}
  // getter typé const : indique que cette méthode ne modifie pas
  // son instance (i.e. this-> val)

  string toString() const {
    return "<"+std::to_string(this->val)+"> ";
  }

  Entier operator +(Entier entier) {
    return Entier(this->val + entier.val);
  }
  // surcharge de l'opérateur +
  // remarquez qu'ici le paramètre est passé par valeur i.e. avec recopie

  Entier operator -(const Entier &entier) {
    return Entier(this->val - entier.val);
  }
  // ici paramètre par référence constante -> pas de recopie
```

};

```
ostream & operator <<(ostream &sortie, const Entier &entier) {
  sortie<<entier.toString();
  return sortie;
}
// redéfinition de << permettant d'afficher une instance de Entier.
// Cette instance est définie comme référence constante (pour éviter
// une recopie qui est couteuse). Voilà pourquoi toString() est déclaré const.
```

On vous demande de tester cette classe et de bien comprendre les affichages. Ajoutez dans votre fichier les affichages

et les explications.

```
int main() {
  Entier x, y(5), z(-3);
  cout<<"x "<<x<<" y "<<y<<" z "<<z<<endl;

  cout<<" Somme "<<endl;
  z = y + z;

  cout<<" somme de x et y "<<z<<endl;

  cout<<" Soustraction "<<endl;
  x = y - z;

  cout<<" soustraction de y - z "<<x<<endl;

}
```

6.10.2 Classe Entier avec pointeur

On définit la même classe avec le même programme principal. Complétez la classe **EntierP**

```
class EntierP {
  int * val;
public :
  EntierP(int val = 0);
  EntierP(const EntierP &ent);
  ~EntierP();
  int getVal() const ;
  string toString() const ;
  EntierP operator +(EntierP entier);
  EntierP operator -(const EntierP &entier);
};
```

Testez ce programme principal. Que constatez vous ?

```
int main() {
  EntierP x, y(5), z(-3);
  cout<<"x "<<x<<" y "<<y<<" z "<<z<<endl;

  cout<<" Somme "<<endl;
  z = y + z;

  cout<<" somme de x et y "<<z<<endl;

  cout<<" Soustraction "<<endl;
  x = y - z;

  cout<<" soustraction de y - z "<<x<<endl;

}
```

Que proposez vous ?

6.11 Semaine 6 Bis

6.12 TD6

6.12.1 Classe Personne et Héritiers

Définissez une classe **Personne** dont les attributs sont : le nom, le prénom et l'âge.

Définissez un **Employe** comme une personne ayant un métier et un salaire.

Définissez un programme principal pour tester ces deux classes.

6.12.2 Etudiant

Définissez une classe **Etudiant** comme une Personne ayant une formation et un tableau de notes.

- Vous devrez ajouter les getters correspondants.
- Le tableau de notes est initialement vide, pour simplifier les notes ne tiendront pas compte des matières et l'étudiant a au plus 50 notes.
- Vous devez pouvoir ajouter une note au tableau de notes puis effectuer la moyenne (il n'y a pas non plus de coefficient) de celles-ci.
- Vous modifierez votre main afin qu'il permmette de tester cette nouvelle classe.
- Vous pourrez ensuite améliorer le type **Note** et le tableau de notes afin qu'il puisse s'agrandir si nécessaire.

Vous prendrez la trame du TD6 sur celène.

CHAPITRE 7

Sujets de Contrôles C++

7.1 Classe Animal (7 points)

Un **Animal** a deux attributs : son nom, et son nombre d'enfants.

1. Définissez les constructeurs suivants : sans paramètre(par défaut nom = « » et 0 enfant), un paramètre (aucun enfant) et deux paramètres.
2. DÃéfinissez les getters, et la méthode toString() permettant l'affichage d'un Animal (son nom et son nombre d'enfants)
3. Surchargez l'opérateur << pour afficher une instance d'Animal.

Complétez les . . . :

```
class Animal {
     ...              nom;
0.5  ...              nbEnfants;

1  ...

// définition (s) de(s) constructeur (s) nécessaire (s)
 ...                                          {

1.5









  }

1  ...            getNbEnfants() {...                                    }

1 ...             getNom() {...                                          }
```

(suite sur la page suivante)

(suite de la page précédente)

```
  ...            toString() {
     ...

1

   }
};
```

```
...                operator <<(...                                    ) {

1

}
```

7.2 Classe ListeDAnimaux (13 points) :

Complétez également la classe **ListeAnimaux** :

```
class ListeAnimaux {
  list<Animal> animaux;
public :
  ListeAnimaux() {}

  void add(string nom, int nb) {
  // ajoute un animal à la liste si aucun animal de celle-ci n'a déjà ce nom.

 3









   }

   ...  moinsDEnfants() {...
   // le nom de l'Animal ayant le moins d'enfants


 3









   }
```

(suite sur la page suivante)

(suite de la page précédente)

```
   string toString() {
   // permet d'afficher la liste des animaux


 2

   }

   list<string> plusDe(int nb) {
   // la liste des animaux ayant plus de nb enfants


 2

   }

   bool estTriNb() {
   // détermine si la liste des animaux est triée par nombre d'enfants croissant


 1

   }
};

...                operator <<(                                          ) {


 1


 }
```

CHAPITRE 8

Indices and tables

— genindex
— modindex
— search