/**** Aide-mémoire sur C++ pour les examens d'INF3105 ****/

1 Types, déclarations, initialisations et opérateurs

Principaux types de base : `bool`, `char`, `int`, `unsigned int`, `short int`, `unsigned short int`, `long`, `unsigned long`, `float`, `double`, ...
Opérateurs arithmétiques : `*`, `/`, `+`, `-`.
Opérateurs logiques : `&&`, `||` (ou).
Opérateurs d'affectation sans/avec opération : `=`, `+=`, `++`, `-=`, `-`, `*=`, `/=`, `,|=`, `,&=`, etc.

```
int a; // a est non initialise
int b(); // b est initialise a 0
int c=2; // c est initialise a 2
int d(3); // d est initialise a 3
a=b++; // post-incrementation  (a=0,b=1)
a=++c; // pre-incrementation   (a=3,c=3)
```

2 Entrées et sorties

```
#include <iostream>
using namespace std; /* std:: est le namespace standard */
int main(){
    int a, b;
    cin >> a >> b; // std::cin >> a >> b;
    std::cout << "La somme est : " << (a+b) << std::endl;
}
```

3 Pointeurs, déférencement, arithmétique des pointeurs et références

- Un pointeur est un objet (une variable) qui contient une adresse mémoire (un entier) de X bits (selon plateforme).
- On peut manipuler un pointeur de façon similaire à un entier (incrémenter, additionner, etc.).
- Le déférencement d'un pointeur doit être explicite à l'aide de l'opérateur de déférencement `*`.
- Une référence est un objet (une variable) qui contient une adresse mémoire, mais se manipule comme un objet.
- Le déférencement d'une référence est implicite. On ne peut manipuler l'adresse mémoire contenu dans une référence.

```
int x; // declare un entier x
int* p; // declare un pointeur d'entier p
int* q = &x; // q prend pour valeur l'adresse memoire de x
p = q; // copie l'adresse memoire contenu dans q vers p
int y = *q; // accede au contenu pointe par q, c.-a-d. x.
int tab[4]; // tab est un pointeur sur le premier element d'un bloc de 4 entiers
p = tab + 2; // p = (l'adresse tab) + (2*sizeof(int)). Idem p=&(tab[2]).
p++; // p pointe sur l'entier suivant, c'est-a-dire tab[3]
cout << tab[2]; // equivaut : cout << *(tab+2).
int& r = x; // cree une refence r sur x
r = 3; // affecte 3 a l'entier refere par r, c-a-d x=3
```

4 Instructions de contrôle d'exécution et blocs d'énoncés

- `if(condition) enonce; [else enonce;]`
- `for(expression_init;expression_condition;expression_increment) enonce;`
- `while(condition) enonce;`
- `do enonce; while(condition);`
- `condition ? expression_sivrai : expression_sifaux;`
- `break` permet d'arrêter et de sortir de la bouche en cours.
- `continue` permet d'arrêter l'itération en cours et d'aller directement à la prochaine itération.
- Les accolades permettent de définir des blocs : `{ enonce1; enonce2; ...; enonceN;}`.

5 Classes, constructeurs, destructeurs, fonctions et appels de fonctions

```
// Declaration (point.h)
#include <iostream>
using namespace std;
class Point{
 public:
  Point();
  Point(double x_,
        double y_);
  ~Point();

  double dist(const
    Point& p) const;

 private:
  double x, y;

friend istream& operator
    >> (istream& is,
    Point& p);
friend ostream& operator
    << (ostream& os, const
    Point& p);
};
```

```
// Definition (point.cpp)
#include <math.h>
#include "point.h"
Point::Point(){}
Point::Point(double x_, double y_)
 : x(x_), y(y_)
{ /* Alternative: x=x_; y=y_; */ }
Point::~Point(){}
double Point::dist(const Point& p) const
{
  double dx=x-p.x, dy=y-p.y;
  return sqrt(dx*dx+dy*dy);
}
istream& operator>>(istream& is,Point& p){
  char parouvr, vir, parferm;
  is>>parouvr>>p.x>>vir>>p.y>>parferm;
  return is;
}
ostream& operator<<(ostream& os, const
    Point& p){
  os << "(" << p.x << "," << p.y << ")";
  return os;
}
```

```
// Utilisation (main.cpp)
#include "point.h"#include
    <math.h>
void f1(
Point a/*par valeur (copie)*/,
Point* b  /*par pointeur*/,
Point& c  /* par reference*/,
const Point& d/*ref. const*/
)
{
  double d1 = b->dist(a);
  double d2 = d.dist(c);
  *b=c=d;//p2=p3=p4
}
int main(){
  Point p1(0,0),p2(3,4),
        p3;
  double d=p1.dist(p2);
  Point* p4 = new Point();
  f1(p1, &p2, p3, *p4);
  delete p4;
  return 0;
}
```

Un constructeur (dans l'ordre) :

1. appelle le constructeur de la ou des classes héritées ;
2. appelle le constructeur de chaque variable d'instance ;
3. exécute le code dans le corps du constructeur.

Un destructeur (dans l'ordre) :

1. exécute le code dans le corps du destructeur.
2. appelle le destructeur de chaque variable d'instance ;
3. appelle le destructeur de la ou des classes héritées ;

6 Allocation de la mémoire automatique et dynamique

```
int main(){
  Point p1(0,0), p2(10,10), p3; // alloue automatiquement sur la pile d'execution
  double d=p1.dist(p2);
  Point* p = new Point();    Point* points = new Point[10];
  delete p;                  delete[] points;
}
```

7 Classes génériques

```
template <class T> class Tableau {
  T*  elements;
  int capacite, nbElements;
public:
  Tableau(int capacite_initiale=5);
  ~Tableau();
  int taille() const {return nbElements;}
  void ajouter(const T& item);
  T& operator[] (int index);
  const T& operator[] (int index) const;
};
```

```
template<class T> Tableau<T>::Tableau(int ci){
  capacite=ci; nbElements=0;
  elements=new T[capacite];
}
template<class T> Tableau<T>::~Tableau(){
  delete[] elements; //elements=NULL;
}
template <class T>
T& Tableau<T>::operator[] (int index){
 assert(index<nbElements);return elements[index];
}
```