Résumé du cours

1. Généralités

Une **fonction** est un **bloc d'instructions éventuellement paramétré** par un ou plusieurs arguments et **pouvant fournir un résultat** nommé souvent « valeur de retour ». On **distingue** la **définition** d'une fonction de son utilisation, cette dernière nécessitant une **déclaration**.

La **définition** d'une fonction se présente comme dans cet exemple :

```
float fexple (float x, int b, int c) // en-tête de la fonction
{ // corps de la fonction
}
```

L'en-tête précise le nom de la fonction (`fexple`) ainsi que le type et le nom (muet) de ses différents arguments (`x, b et c`). Le corps est un bloc d'instructions qui définit le rôle de la fonction.

Au sein d'une autre fonction (y compris main), on utilise cette fonction de cette façon :

```
float fepxle (float, int, int) ; // déclaration de fexple ("prototype")
 .....
fexple (z, n, p) ; //appel fexple avec les arguments effectifs z, n et p
```

La déclaration d'une fonction peut être omise lorsqu'elle est connue du compilateur, c'est-à-dire que sa définition a déjà été rencontrée dans le même fichier source.

resume.cpp

```
#include<iostream>

using namespace std;

//déclaration des variables globales


//déclaration des fonctions
void affiche(double, double, double);
double somme(double, double);



//programme principal
int main()
{
    //déclaration et définition des variables locales à la fonction main()
    double parm1(12.5);
    int parm2(5);
    double resultSomme(0);

    resultSomme = somme(parm1, parm2);
    affiche(parm1, parm2, resultSomme);


    return 0;
}


//définition des fonctions
void affiche(double arg1, double arg2 , double arg3){
    cout << arg1 << " + " << arg2 << " = " << arg3 << endl;
}

double somme(double arg1, double arg2){
    return arg1 + arg2;
}
```

2. Mode de transmission des arguments

Par défaut, les arguments sont transmis par valeur. Dans ce cas, les arguments effectifs peuvent se présenter sous la forme d'une expression quelconque.

En faisant suivre du symbole **&** le type d'un argument dans l'en-tête d'une fonction (et dans sa déclaration), on réalise une **transmission par référence**. Cela signifie que les éventuelles **modifications** effectuées au **sein** de la **fonction porteront sur l'argument effectif** de l'appel et **non plus sur une copie**. On notera qu'alors l'argument effectif doit obligatoirement être une *lvalue* du même type que l'argument muet correspondant. Toutefois, si l'argument muet est, de surcroît, déclaré avec l'attribut const, la fonction reçoit quand même une copie de l'argument effectif correspondant, lequel peut alors être une constante ou une expression d'un type susceptible d'être converti dans le type attendu.

Ces possibilités de **transmission par référence s'appliquent également à une valeur de retour** (dans ce cas, la notion de constance n'a plus de signification).

La notion de référence est théoriquement indépendante de celle de transmission d'argument ; en pratique, elle est rarement utilisée en dehors de ce contexte.

resume.cpp

```
#include<iostream>

using namespace std;

//déclaration des variables globales


//déclaration des fonctions
void affiche(double, double, double);
double somme(double, double);
void cumulerSomme(double&, int);
```

```
void affiche(double);



//programme principal
int main()
{
    //déclaration et définition des variables locales à la fonction main()
    double parm1(12.5);
    int parm2(5);
    double resultSomme(0);
    double sommeCumuler(0);

    /*resultSomme = somme(parm1, parm2);
    affiche(parm1, parm2, resultSomme);*/

    cumulerSomme(sommeCumuler, 10);
    affiche(sommeCumuler);


    return 0;
}


//définition des fonctions
void affiche(double arg1, double arg2 , double arg3){
    cout << arg1 << " + " << arg2 << " = " << arg3 << endl;
}

double somme(double arg1, double arg2){
    return arg1 + arg2;
}

void cumulerSomme(double &arg1, int arg2){
    arg1 += arg2;
}

void affiche(double arg1){
    cout << "Somme cumulée : " << arg1 << endl;
}
```

3. L’instruction return

L’instruction `return` sert à la fois à **fournir** une **valeur de retour et à mettre fin à l’exécution de la fonction**. Elle peut mentionner une expression ; elle peut apparaître à plusieurs reprises dans une même fonction ; **si aucune** instruction **return** n’est mentionnée, le **retour** est mis en place à la **fin** de la **fonction**.

Lorsqu’une **fonction** ne **fournit aucun résultat**, son en-tête et sa déclaration comportent le mot `void` à la place du type de la **valeur de retour**, comme dans :

```
void fSansValRetour (...)
```

Lorsqu’une **fonction** ne reçoit **aucun argument**, l’en-tête et la déclaration comportent une **liste vide**, comme dans :

```
int fSansArguments ()
```

4. Les arguments par défaut

Dans la déclaration d’une fonction, il est possible de prévoir pour un ou plusieurs **arguments** (**obligatoirement les derniers de la liste**) des **valeurs par défaut** ; elles sont **indiquées** par le signe **=**, à la suite du type de l’argument, comme dans cet exemple :

```
float fct (char, int = 10, float = 0.0) ;
```

Ces **valeurs par défaut** seront alors **utilisées** lorsqu’on **appellera** ladite **fonction** avec un **nombre d’arguments inférieur à celui prévu**.
Par exemple, avec la précédente déclaration, l’appel **`fct ('a')`** sera **équivalent** à **`fct ('a', 10, 0.0)`** ; de même, l’appel **`fct ('x', 12)`** sera **équivalent** à **`fct ('x', 12, 0.0)`**.
En revanche, l’appel **`fct ()`** sera **illégal**.

5. Conversion des arguments

Lorsqu’un **argument est transmis par valeur**, il est **éventuellement converti** dans le type mentionné dans la déclaration (la conversion peut être dégradante). Ces possibilités de **conversion disparaissent** en cas de **transmission par référence** : l’argument effectif doit alors être une `lvalue` du type prévu ; cette dernière ne peut posséder l’attribut const si celui-ci n’est pas prévu dans l’argument muet ; en revanche, si l’argument muet mentionne l’attribut const, l’argument effectif pourra être non seulement une constante, mais également une expression d’un type quelconque dont la valeur sera alors convertie dans une variable temporaire dont l’adresse sera fournie à la fonction.

6. Variables globales et locales

Une **variable déclarée en dehors de toute fonction** (y compris du main) est dite **globale**. La **portée** d’une variable globale est **limitée** à la partie du **programme source suivant sa déclaration**. Les variable globales ont une **classe d’allocation statique**, ce qui signifie que leurs **emplacements en mémoire** restent **fixes** (sur le tas) pendant l’exécution du programme.

Une variable déclarée dans une fonction est dite locale. La **portée** d’une variable locale est **limitée à la fonction dans laquelle elle est définie**. Les variables locales ont une **classe d’allocation automatique**, ce qui signifie que leurs **emplacements** (sur la pile) sont **alloués** à **l’entrée** dans la **fonction**, et **libérés** à la **sortie**. Il est possible de déclarer des variables locales à un bloc.

On peut **demander** qu’une **variable locale** soit de **classe d’allocation statique**, en la déclarant à l’aide du **mot-clé static**, comme dans

```
static int i ;
```

Les **variables** de **classe statique** sont, par défaut, **initialisées à zéro**. On peut les initialiser explicitement à l’aide d’expressions constantes d’un type compatible par affectation avec celui de la variable. Celles de **classe automatique** ne sont **pas initialisées** par défaut. Elles doivent être

initialisées à l'aide d'une expression quelconque, d'un type compatible par affectation avec celui de la variable.

7. Surdéfinition de fonctions

En C++, il est **possible**, au sein d'un **même programme**, que **plusieurs fonctions possèdent le même nom**. Dans ce cas, lorsque le **compilateur** rencontre l'appel d'une telle fonction, il effectue le **choix** de la « bonne » **fonction** en tenant compte de la **nature des arguments effectifs**. D'une manière générale, si les règles utilisées par le compilateur pour sa recherche sont assez intuitives, leur énoncé précis est assez complexe, et nous ne le rappellerons pas ici. Signalons simplement que la recherche d'une fonction surdéfinie peut faire intervenir toutes les conversions usuelles (promotions numériques et conversions standards, ces dernières pouvant être dégradantes), ainsi que les conversions définies par l'utilisateur en cas d'argument de type classe, à condition qu'aucune ambiguïté n'apparaisse.

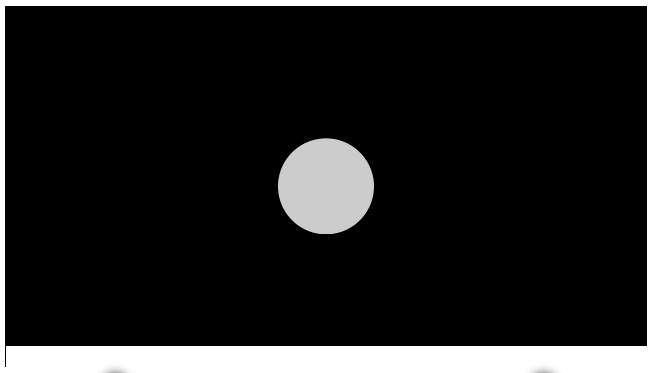
8. Les fonctions en ligne

Une fonction en ligne (on dit aussi « développée ») est une fonction dont les instructions sont incorporées par le compilateur (dans le module objet correspondant) à chaque appel. Cela **évite la perte de temps nécessaire à un appel usuel** (changement de contexte, copie des valeurs des arguments sur la « pile »…) ; **en revanche**, les **instructions** en question sont **générées plusieurs fois**.

Une fonction en ligne est nécessairement **définie en même temps qu'elle est déclarée** (elle **ne peut plus être compilée séparément**) et son en-tête est précédée du **mot-clé inline**, comme dans :

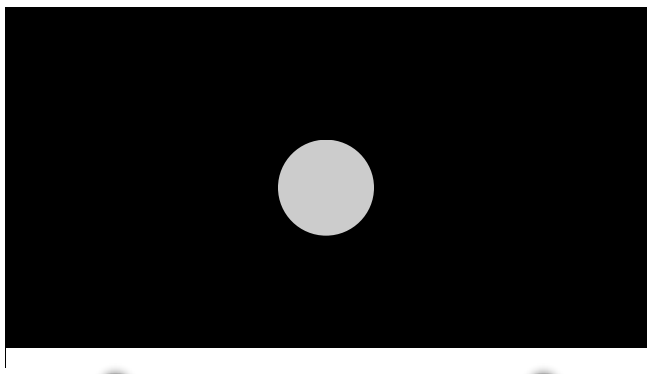
```
inline fct ( ...) { ..... }
```

Cours



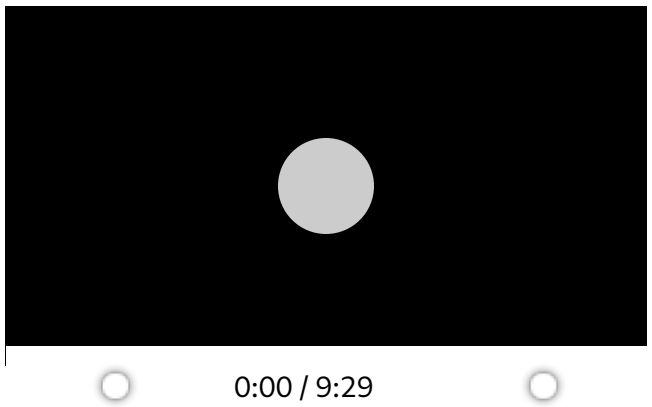
5 - 1 - Fonctions - introduction [16-07]

- Le cours en pdf



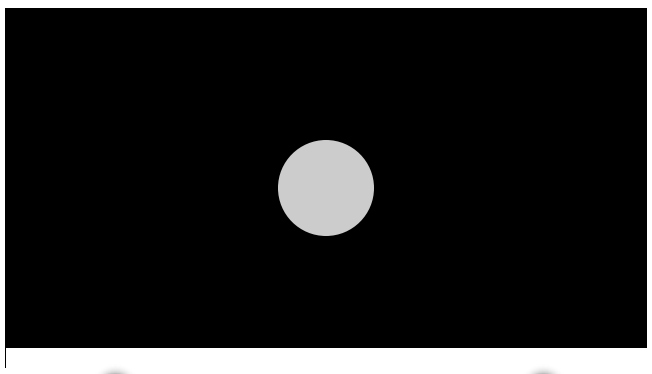
5 - 2 - Fonctions - appels [09-35]

- Le cours en pdf



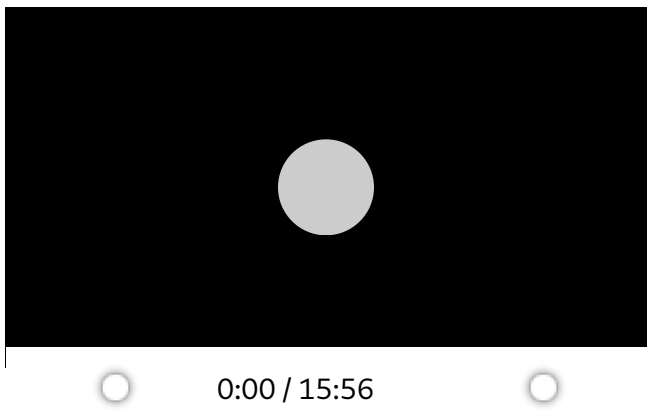
5 - 3 - Fonctions - passage des arguments [09-29]

- Le cours en pdf



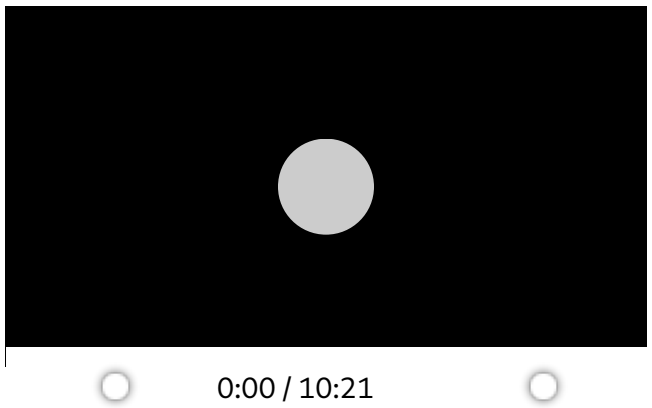
5 - 4 - Fonctions - prototypes [05-56]

- Le cours en pdf



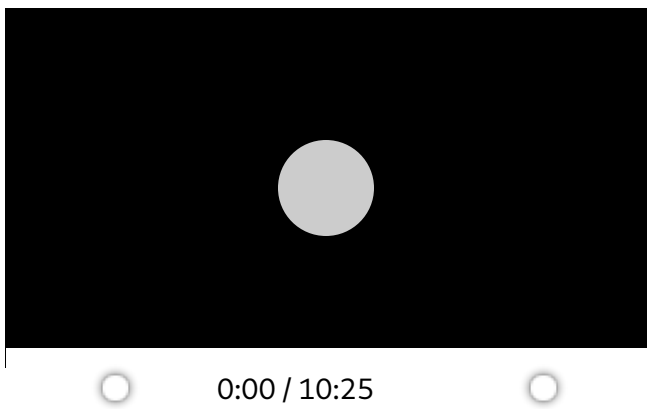
5 - 5 - Fonctions - définitions [15-55]

- Le cours en pdf



5 - 6 - Fonctions - méthodologie [10-21]

- Le cours en pdf



5 - 7 - Fonctions - arguments par défaut et surcharge [10-25]

- Le cours en pdf

tutoriels

- Énoncés

Exercices

Exercice 1

Énoncé
Quelle modification faut-il apporter au programme suivant pour qu'il devienne correct :

```
#include <iostream>
using namespace std ;
main()
{ int n, p=5 ;
 n = fct (p) ;
 cout << "p = " << p << " n = " << n ;
}
int fct (int r)
{ return 2*r ;
}
```

Exercice 2

Énoncé
Écrire :

- une fonction, nommée `f1`, se contentant d'afficher « bonjour » (elle ne possédera aucun argument, ni valeur de retour) ;
- une fonction, nommée `f2`, qui affiche « bonjour » un nombre de fois égal à la valeur reçue en argument (`int`) et qui ne renvoie aucune valeur ;
- une fonction, nommée `f3`, qui fait la même chose que `f2`, mais qui, de plus, renvoie la valeur (**int**) `0`.

Écrire un petit programme appelant successivement chacune de ces 3 fonctions, après les avoir convenablement déclarées (on ne fera aucune hypothèse sur les emplacements relatifs des différentes fonctions composant le fichier source).

Exercice 3

Énoncé
Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
int n=10, q=2 ;
main()
{
 int fct (int) ;
 void f (void) ;
 int n=0, p=5 ;
 n = fct(p) ;
 cout << "A : dans main, n = " << n << " p = " << p
 << " q = " << q << "\n" ;
 f() ;
}
int fct (int p)
{
 int q ;
 q = 2 * p + n ;
 cout << "B : dans fct, n = " << n << " p = " << p
 << " q = " << q << "\n" ;
 return q ;
}
void f (void)
{
 int p = q * n ;
 cout << "C : dans f, n = " << n << " p = " << p
 << " q = " << q << "\n" ;
}
```

Exercice 4

Énoncé
Écrire une fonction qui reçoit en arguments 2 nombres flottants et un caractère, et qui fournit un résultat correspondant à l'une des 4 opérations appliquées à ses deux premiers arguments, en fonction de la valeur du dernier, à savoir : addition pour le caractère `+`, soustraction pour `-`, multiplication pour `*` et division pour `/` (tout autre caractère que l'un des 4 cités sera interprété comme une addition). On ne tiendra pas compte des risques de division par zéro.
Écrire un petit programme (`main`) utilisant cette fonction pour effectuer les 4 opérations sur les 2 nombres fournis en donnée.

Exercice 5

Énoncé

Transformer le programme (fonction + main) écrit dans l'exercice précédent de manière que la fonction ne dispose plus que de 2 arguments, le caractère indiquant la nature de l'opération à effectuer étant précisé, cette fois, à l'aide d'une variable globale.

Exercice 6

Énoncé

Écrire une fonction, sans argument ni valeur de retour, qui se contente d'afficher, à chaque appel, le nombre total de fois où elle a été appelée sous la forme :
appel numéro 3

Exercice 7

Énoncé

Écrire 2 fonctions à un argument entier et une valeur de retour entière permettant de préciser si l'argument reçu est multiple de 2 (pour la première fonction) ou multiple de 3 (pour la seconde fonction).

Utiliser ces deux fonctions dans un petit programme qui lit un nombre entier et qui précise s'il est pair, multiple de 3 et/ou multiple de 6, comme dans cet exemple (il y a deux exécutions) :

```
donnez un entier : 9
il est multiple de 3
 ---------------
donnez un entier : 12
il est pair
il est multiple de 3
il est divisible par 6
```

Exercice 8

Énoncé

Écrire une fonction permettant d'ajouter une valeur fournie en argument à une variable fournie également en argument. Par exemple, l'appel (n et p étant entiers) :

```
ajouter (2*p+1, n) ;
```

ajoutera la valeur de l'expression 2*p+1 à la variable n.
Écrire un petit programme de test de la fonction.

Exercice 9

Énoncé

Soient les déclarations suivantes :

```
int fct (int) ; // fonction I
int fct (float) ; // fonction II
void fct (int, float) ; // fonction III
void fct (float, int) ; // fonction IV
int n, p ;
float x, y ;
char c ;
double z ;
```

Les appels suivants sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions éventuellement mises en place ?

a. `fct (n) ;`
b. `fct (x) ;`
c. `fct (n, x) ;`
d. `fct (x, n) ;`
e. `fct © ;`
f. `fct (n, p) ;`
g. `fct (n, c) ;`
h. `fct (n, z) ;`
i. `fct (z, z) ;`

Exercice 10

Énoncé

a. Transformer le programme suivant pour que la fonction fct devienne une fonction en ligne.

```
#include <iostream>
using namespace std ;
main()
{ int fct (char, int) ; // déclaration (prototype) de fct
 int n = 150, p ;
 char c = 's' ;
 p = fct ( c , n) ;
 cout << "fct (\'" << c << "\', " << n << ") vaut : " << p ;
}
int fct (char c, int n) // définition de fct
{ int res ;
 if (c == 'a') res = n + c ;
 else if (c == 's') res = n - c ;
 else res = n * c ;
 return res ;
}
```

b. Comment faudrait-il procéder si l'on souhaitait que la fonction fct soit compilée séparément ?

Devoirs

Pour les IR

- Énoncés

lireetdire.cc

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

```
int separer_chiffre_gauche(int& nombre)
{
  int dix(1);
  int temp(nombre);
  while (temp >= 10) {
    dix  *= 10;
    temp /= 10;
  }
  nombre %= dix;
  return temp;
}

/*****************************************************
 * Compléter le code à partir d'ici
 *****************************************************/

void ajouter_chiffre_droit(int& nombre, int chiffre)
{
}

void dire_chiffre(int& nombre, int repetitions_chiffre, int chiffre)
{
}

int lire_et_dire(int nombre)
{
}

/*****************************************
 * Ne rien modifier après cette ligne.
 *****************************************/

void repeter_lire_et_dire(int& nombre, int fois)
{
  while (fois-- > 0) {
    nombre = lire_et_dire(nombre);
  }
}

int main()
{
  int nombre(1);
  int fois(1);
  cin >> nombre >> fois;
  repeter_lire_et_dire(nombre, fois);
  cout << nombre << endl;
  return 0;
}
```

mastermind.cc

```
#include <iostream>
#include <ctime>

// pour les nombres aléatoires
#include <random>
#include <cstring> // strlen

using namespace std;

// ========================================================================
// Couleur au hasard
std::uniform_int_distribution<int> distribution;
std::default_random_engine generateur(time(NULL)); /* NOT using std::random_device since not
                                                    * all compilers seems to support it :-( */

char tirer_couleur()
{
  static const char* const couleurs = ".RGBCYM";
  static const int nb = strlen(couleurs) - 1;

  return couleurs[distribution(generateur,
                               std::uniform_int_distribution<int>::param_type {0, nb})];
}

// ========================================================================
char poser_question()
{
  char lu(' ');
  cout << "Entrez une couleur : ";
  cin >> lu;
  return lu;
}

// ---- prototype -----------------------------------------------------
bool couleur_valide(char c);

// ========================================================================
char lire_couleur()
{
  char lu(poser_question());
  while (not couleur_valide(lu)) {
    cout << "'" << lu << "' n'est pas une couleur valide." << endl;
    cout << "Les couleurs possibles sont : ., R, G, B, C, Y ou M." << endl;
    lu = poser_question();
  }
  return lu;
}

// ========================================================================
void afficher_couleurs(char c1, char c2, char c3, char c4)
{
  cout << ' ' << c1 << ' ' << c2 << ' ' << c3 << ' ' << c4;
}

// ========================================================================
void afficher(int nb, char c)
{
  while (nb-- > 0) {
    cout << c;
  }
}

// ---- prototype -----------------------------------------------------
void afficher_reponses(char c1, char c2, char c3, char c4,
                       char r1, char r2, char r3, char r4);
```

```
// ========================================================================
void afficher_coup(char c1, char c2, char c3, char c4,
                   char r1, char r2, char r3, char r4)
{
  afficher_couleurs(c1, c2, c3, c4);
  cout << " : ";
  afficher_reponses(c1, c2, c3, c4,
                    r1, r2, r3, r4);
  cout << endl;
}

// ========================================================================
void message_gagne(int nb_coups)
{
  cout << "Bravo ! Vous avez trouvé en " << nb_coups << " coups." << endl;
}

// ========================================================================
void message_perdu(char c1, char c2, char c3, char c4)
{
  cout << "Perdu :-(" << endl;
  cout << "La bonne combinaison était : ";
  afficher_couleurs(c1, c2, c3, c4);
  cout << endl;
}

/************************************************
 * Compléter le code à partir d'ici
 ************************************************/

// ========================================================================
bool couleur_valide(char c)
{
}

// ========================================================================
bool verifier(// A remplir
              )
{
}

// ========================================================================
void apparier(// A remplir
              )
{
}

// ========================================================================
void afficher_reponses(char c1, char c2, char c3, char c4,
                       char r1, char r2, char r3, char r4)
{
}

// ========================================================================
bool gagne(char c1, char c2, char c3, char c4,
           char r1, char r2, char r3, char r4)
{
}

// ========================================================================
void jouer(// A remplir
           )
{
}

/***************************************
 * Ne rien modifier après cette ligne.
 ***************************************/

int main()
{
  jouer();
  return 0;
}
```

Pour les EC

- La fonction cosinus

	Exercice 1	**Exercice 2**	**Exercice 4**	**Exercice 6**	**Exercice 7**	**Exercice 8**
ABAL HASSANI Habibi	OK					
BERENGUER Ianis	OK					
COLIN Tristan	OK					
FLEURY Anthony	OK					
FROSTIN Arnaud	OK					
LECOMTE Martin	OK	OK				