Chapitre 4 - 1 Programmation séparée et programmation objet en c++

Points abordés précedemment:

- variables, boucles, structures conditionnelles
- compilation de quelques programmes
- fonctions (itératives, récursives)
- première utilisation de struct et structures de la stl

Jusqu'à maintenant, nous avons créé un programme c++ à chaque question. Chaque question a donc nécessité la définition d'un point d'entrée (fonction `int main()`). Ceci ne pose pas de problème particulier de compilation (compilation séparée de chaque programme) mais un gros problème de lisibilité ! Cette partie du cours 3 est dédiée à la création d'un **projet** c++ regroupant plusieurs programmes rassemblés en un seul projet.

Programmation sur plusieurs fichiers (projet)

Principe

Le code est réparti par thèmes entre plusieurs fichiers cpp. **Un** fichier principal contient le point d'entrée du projet (fonction `int main()`). Un fichier peut utiliser un type ou une fonction définie dans un autre fichier. Pour cela, le compilateur (g++, clang, mingw, visual studio) doit connaitre le type ou le prototype de la fonction. Cette information est contenue dans les fichiers .h qui seront inclus partout ou cette information est utilisée.

Attension : On n'inclut pas des fichiers .cpp (définition des fonctions). C'est le compilateur qui trouvera dans quel fichier est définie la fonction appelée.

Avec le programme du TD

Prenons un exemple du programme (Fraction et decomposition) vu dans le TD précédent. Essayons de mettre un peu d'ordre dans ce programme en le séparant en plusieurs thèmes. Typiquement, est-ce que le calcul du pgcd rentre dans le même thème que les calculs associés aux fractions? Autrement dit, nous allons essayer de

- segmenter le programme par thème pour le rendre plus lisible.,
- placer les prototypes dans les entetes (header) et la définition des fonctions dans les .cpp, cela pour chaque programme obtenu,
- compiler les programmes créés,
- executer le fichier objet obtenu.

Arborescence standard

Etant donné les fichiers .c++ et .h du projet, il est standard de placer les fichiers d'entête dans un répertoire include et les fichiers .cpp dans un répertoire src. N'oubliez pas d'**inclure les fichiers d'entête** dans les fichiers .cpp. Ceci se fait avec l'instruction du préprocesseur

In [ ]:
```
#include "Point.h" // pour inclure le fichier d'entete Point.h dans le code P
```

Le contenu du fichier `Point.h` peut être le suivant.

In [1]:
```
%%writefile Point.h
#include <iostream>

struct Point{
    int abs;
    int ord;
};

void saisiePoint(Point* p);
```

```
Writing Point.h
```

Le contenu du fichier `Droite.h` peut être le suivant et devra inclure les prototypes de `Point.h` .

In [2]:
```
%%writefile Droite.h

#include "Point.h"

struct Droite{
    double a;
    double b;

    void construitDroite(Point point1, Point point2);

    void affiche();

};
int intersectionDroite(Droite alpha, Droite beta, Point* p, double epsilon);
```

```
Writing Droite.h
```

Les fichiers `.cpp` associés contiendront les implantations des fonctions et des `struct` dont les prototypes sont dans les `.h` . On ajoute donc les fichiers `Point.cpp` et `Droite.cpp` .

In [3]:
```
%%writefile Point.cpp

#include "Point.h"

void saisiePoint(Point* p){
    std::cout << "saisir les coordonnées du point ..."<<std::endl;
    std::cin >> (*p).abs >> (*p).ord;
}
```

```
Writing Point.cpp
```

In [4]:
```
%%writefile Droite.cpp

#include "Droite.h"

void Droite::construitDroite(Point point1, Point point2){
        if((point2.abs-point1.abs)==0)
            std::cout << "attention meme abscisse"<<std::endl;
        else{
        a = (point2.ord - point1.ord)/(point2.abs - point1.abs);
```

```
        b = point1.ord - a*point1.abs;
        }
}

void Droite::affiche(){
        std::cout << "Droite ("<<a<<","<<b<<")"<<std::endl;
}

int intersectionDroite(Droite alpha, Droite beta, Point* p, double epsilon){
    if(abs(alpha.a - beta.a)<epsilon){
        std::cout << "0 ou une infinité d'intersection" <<std::endl;
        return 0;
    }

    (*p).abs = (beta.b - alpha.b)/(alpha.a - beta.a); // y = a1*x + b1
    (*p).ord = (alpha.a*((*p).abs) + alpha.b);  // y = a2*x + b2

    return 1;
}
```

```
Writing Droite.cpp
```

Compilation

A ce point du cours, le programme est séparé en plusieurs `.cpp` et `.h` . Nous cherchons maintenant à executer le programme. Tout d'abord, pensez bien à définir le point d'entrée de votre programme dans un fichier `.cpp` . Il peut y avoir plusieurs `.cpp` mais **une** seule fonction `int main()` dans un seul fichier `.cpp` . S'il n'y a pas de fonction `main()` définie alors la compilation du programme peut retourner ce type d'erreur

```
Undefined symbols for architecture arm64:
  "_main", referenced from:
     implicit entry/start for main executable
ld: symbol(s) not found for architecture arm64
```

In [6]:

```
%%writefile main.cpp

#include "Droite.h"
#include "Point.h"


int main(){

    Point pt1,pt2,pt3;
    pt1.abs = 0;
    pt1.ord = 1;
    pt2.abs = 1;
    pt2.ord = 2;

    pt3.abs = 0;
    pt3.abs = 0;

    Droite d1,d2;
    d1.construitDroite(pt1,pt2);
    d1.affiche();

    d2.construitDroite(pt3,pt2);


    Point ptintersection;
    double epsilon = 1e-7;
```

```
    intersectionDroite(d1,d2,&ptintersection,epsilon);

    std::cout << "Point intersection("<< ptintersection.abs << ","<<ptinterse

    return 0;
}
```

```
Overwriting main.cpp
```

In [7]:

```
%%shell
g++ -Wall -o test main.cpp Droite.cpp Point.cpp
./test
```

```
In file included from main.cpp:3:
Point.h:3:8: error: redefinition of ‘struct Point’
    3 | struct Point{
      |        ^~~~~
In file included from Droite.h:2,
                 from main.cpp:2:
Point.h:3:8: note: previous definition of ‘struct Point’
    3 | struct Point{
      |        ^~~~~
/bin/bash: line 2: ./test: No such file or directory
---------------------------------------------------------------------------
CalledProcessError                        Traceback (most recent call last)
<ipython-input-7-9daacba0ef87> in <cell line: 1>()
----> 1 get_ipython().run_cell_magic('shell', '', 'g++ -Wall -o test main.cpp
Droite.cpp Point.cpp\n./test\n')

/usr/local/lib/python3.10/dist-packages/google/colab/_shell.py in run_cell_ma
gic(self, magic_name, line, cell)
    332     if line and not cell:
    333       cell = ' '
--> 334     return super().run_cell_magic(magic_name, line, cell)
    335
    336

/usr/local/lib/python3.10/dist-packages/IPython/core/interactiveshell.py in r
un_cell_magic(self, magic_name, line, cell)
   2471             with self.builtin_trap:
   2472                 args = (magic_arg_s, cell)
-> 2473                 result = fn(*args, **kwargs)
   2474             return result
   2475

/usr/local/lib/python3.10/dist-packages/google/colab/_system_commands.py in _
shell_cell_magic(args, cmd)
    110   result = _run_command(cmd, clear_streamed_output=False)
    111   if not parsed_args.ignore_errors:
--> 112     result.check_returncode()
    113   return result
    114

/usr/local/lib/python3.10/dist-packages/google/colab/_system_commands.py in c
heck_returncode(self)
    135   def check_returncode(self):
    136     if self.returncode:
--> 137       raise subprocess.CalledProcessError(
    138           returncode=self.returncode, cmd=self.args, output=self.outp
ut
    139       )
```

```
CalledProcessError: Command 'g++ -Wall -o test main.cpp Droite.cpp Point.cpp
./test
' returned non-zero exit status 127.
```

Résultat de la compilation

```
error: redefinition of struct Point ...
```

Ce n'est pas encore fini ! En effet, nous incluons plusieurs fois le même fichier d'entête dans le même projet. Par conséquent, les fonctions sont recopiées, ce qui induit une erreur de compilation puisqu'il est **interdit** de définir 2 fois la même fonction !

Utilisation de garde fou (include guard)

Pour résoudre ce problème, on ajoute l'ensemble d'instructions

```
#ifndef NOMFICHIERDENTETE
#define NOMFICHIERDENTETE

//fonctions
.
.
.


#endif
```

à **chaque** fichier d'entête. Ces instructions permettent de ne compiler et recopier ce programme **qu'une seule fois**. La macro `NOMFICHIERDENTE` sera définie lors de la compilation du deuxième programme qui inclut la même entête.

In [ ]:
```
%%writefile Droite.h

#ifndef DROITEH
#define DROITEH


#include "Point.h"

struct Droite{
    double a;
    double b;

    void construitDroite(Point point1, Point point2);

    void affiche();

};
int intersectionDroite(Droite alpha, Droite beta, Point* p, double epsilon);
#endif
```

```
Overwriting Droite.h
```

In [ ]:
```
%%writefile Point.h

#ifndef POINTH
#define POINTH
```

```
#include <iostream>

struct Point{
    int abs;
    int ord;
};

void saisiePoint(Point* p);
#endif
```

```
Overwriting Point.h
```

La compilation devrait maintenant passer et nous pouvons voir le résultat.

In [ ]:

```
%%shell
g++ -Wall -o test main.cpp Droite.cpp Point.cpp
./test
```

```
Droite (1,1)
Point intersection(1,2)
```

Out[ ]:

Compilation séparée multiplateforme

L'écriture de la commande `g++` peut poser plusieurs problèmes :

- Lisibilité et réutilisabilité, lorsque le projet se compose de plusieurs programmes ou que vous voulez partager votre programme avec quelqu'un,
- dépendance à la plateforme, lorsque le compilateur c++ utilisé n'est pas g++, visual studio par exemple...

Nous commencons pour cela à étudier un métalangage pour effectuer la compilation multiplateforme avec `cmake` . Pour ce faire, nous créons un fichier CMakeLists.txt dans la racine de notre projet.

```
# commentaire
cmake_minimum_required (VERSION 3.12)

# le nom de votre projet, aussi pour la création d une solution
sous visual studio
project(fractionAndDecomposition)


message(STATUS "directory : ${PROJECT_SOURCE_DIR}")

# ajouter un ensemble fichiers cpp à la variable SRCS
file(GLOB SRCS CONFIGURE_DEPENDS src/*.cpp )

# lister les programmes cpp trouves
message(STATUS "source cpp ${SRCS} ")

# créer la cible
add_executable(fractionAndDecomposition ${SRCS} )

# ajouter le répertoire contenant les fichiers d'entete dans la
```

```
cible
target_include_directories(fractionAndDecomposition PUBLIC
include)
```

Comment lancer la compilation et l'éxecution?

```
cmake ..
make
./fractionAndDecomposition
```

Et sous Visual Studio Code?

- Installer cmake,
- Installer l'extension cmake sous visual studio code (onglet extension),
- Lancer la configuration (configure) et la construction de l'executable (build).

Notions de programmation objet en c++

Commençons par un bref rappel de ce qu'on sait faire avec les structures puisque cette structure va être notre base pour faire de la programmation objet en c++:

In [ ]:

```
struct Personnage{
    // attributs ou champs
    int vie;
    int attaque;
    int force;



    // methodes membres
    void monterNiveau();
    int attaquer( Personnage autre);
    void parler(std::string discours);

}; // attention au ;

class Personnage{
    // attributs ou champs
    int vie;
    int attaque;
    int force;

    void faireQuelquechose();

    // methodes membres
public:
    void monterNiveau();
    int attaquer( Personnage autre);
    void parler(std::string discours);

}; // attention au ;


int main(){
    Personnage yoda, vador, palpatine;
    yoda.attaquer(palpatine);
    palpatine.parler("c'est un moment que j'attendais depuis bien longtemps,
```

```
    return 0;
}
```

Pour rappel, la programmation objet (POO) est un modèle de programmation construit autour de la notion d'objet. En bref, il s'agit d'une description abstraite d'un objet. Il définit une spécification d'un ensemble de données (attributs ou champs) avec l'ensemble des opérations qu'on peut effectuer sur les données. Trois principaux concepts sont liés à la programmation objet, à savoir

- l'encapsulation,
- l'héritage,
- le polymorphisme.

Encapsulation

Le principe est de faire en sorte que l'utilisateur puisse utiliser une structure (struct) sans connaître les attributs (propriétés) de la structure. On laisse le soin à la structure de modifier ses propriétés. Son application en c++ passe par l'écriture de fonctions pour l'utilisateur permettant de modifier les attributs de manière conforme à la définition de l'objet (getter/setter). Un exemple à partir de notre structure `Fraction` se trouve dans l'écriture d'une fonction :

In [ ]:
```
void make_fraction(int a, int b);
```

Héritage

L'héritage en programmation consiste à écrire des structures qui héritent des fonctions et des membres d'autres structures. Dès lors que vous dites qu'une structure S1 **est** une structure particulère S2 alors vous devez utiliser l'héritage. Par exemple, un chat est un animal. Si vous devez écrire la structure Chat après la structure Animal alors le Chat devra hériter de la structure Animal. L'héritage en c++ se fait de la manière suivante

In [ ]:
```
struct Animal { };
struct Chat : Animal { };
struct Chien : Animal { };
```

Comme dit plus haut, la structure fille qui hérite de structure mère hérite des méthodes et des attributs de la structure mère. L'exemple suivant est valide.

In [ ]:
```
%%writefile Personnage.cpp
#include <iostream>
#include <string>

struct Personne {
    int age;
    std::string nom;

    void modifierNom(std::string leNom){
        nom = leNom;
    }

    void modifierAge(int lage){
        age = lage;
    }
```

```
};

struct Etudiant : Personne {
    int cursus;

};
int main(){
  Etudiant e1;
  e1.modifierNom("Leo");
  e1.modifierAge(23);

  std::cout << "nom de l etudiant "<<(e1.nom)<<std::endl;
}
```

```
Writing Personnage.cpp
```

In [ ]:

```
%%shell
g++ -Wall -o personnage Personnage.cpp
./personnage
```

```
nom de l etudiant Leo
```

Out[ ]:

Par contre, une variable (instance) du type de la structure mère ne peut pas utiliser les attributs de la structure fille.

In [ ]:

```
%%writefile Personnage.cpp

#include <iostream>
#include <string>



struct Personne {
    int age;
    std::string nom;

    void modifierNom(std::string leNom){
        nom = leNom;
    }

    void modifierAge(int lage){
        age = lage;
    }

    void afficheAge(){
        std::cout << "age="<< age<<std::endl;
    }


};

struct Etudiant : Personne {
    int cursus;

};



int main(){
  Etudiant e1;
  e1.modifierNom("Leo");
```

```
  e1.modifierAge(23);

  Personne p1;
  p1.cursus = 4;

  e1.afficheAge();

  std::cout << "nom de l etudiant ="<<(e1.nom)<<std::endl;

  std::cout << "cursus de la personne ="<<(p1.cursus)<<std::endl;
}
```

```
Overwriting Personnage.cpp
```

In [ ]:

```
%%shell
g++ -Wall -o personnage Personnage.cpp
```

```
Personnage.cpp: In function ‘int main()’:
Personnage.cpp:31:6: error: ‘struct Personne’ has no member named ‘cursus’
   31 |   p1.cursus = 4;
      |      ^~~~~~
Personnage.cpp:35:47: error: ‘struct Personne’ has no member named ‘cursus’
   35 |   std::cout << "cursus de la personne ="<<(p1.cursus)<<std::endl;
      |                                               ^~~~~~
---------------------------------------------------------------------------
CalledProcessError                        Traceback (most recent call last)
<ipython-input-30-297110588c14> in <module>
----> 1 get_ipython().run_cell_magic('shell', '', 'g++ -Wall -o personnage Pe
rsonnage.cpp\n')

/usr/local/lib/python3.8/dist-packages/IPython/core/interactiveshell.py in ru
n_cell_magic(self, magic_name, line, cell)
   2357             with self.builtin_trap:
   2358                 args = (magic_arg_s, cell)
-> 2359                 result = fn(*args, **kwargs)
   2360             return result
   2361

/usr/local/lib/python3.8/dist-packages/google/colab/_system_commands.py in _s
hell_cell_magic(args, cmd)
    107   result = _run_command(cmd, clear_streamed_output=False)
    108   if not parsed_args.ignore_errors:
--> 109     result.check_returncode()
    110   return result
    111

/usr/local/lib/python3.8/dist-packages/google/colab/_system_commands.py in ch
eck_returncode(self)
    132   def check_returncode(self):
    133     if self.returncode:
--> 134       raise subprocess.CalledProcessError(
    135           returncode=self.returncode, cmd=self.args, output=self.outp
ut)
    136

CalledProcessError: Command 'g++ -Wall -o personnage Personnage.cpp
' returned non-zero exit status 1.
```

Polymorphisme

En c++, il y a deux types de polymorphismes:

1/ polymorphisme à la **compilation** du programme

2/ polymorphisme à l'**exécution** du programme

1/ polymorphisme à la compilation du programme

Il s'agit simplement de surcharger des fonctions : même nom de fonction mais avec des paramètres de type différents, exemple

In [ ]:
```
int multiplienValeurs(int a, int b){
    return a*b;
}

int multiplienValeurs(int a, int b, int c){
    return a*b*c;
}
int multiplienValeurs(int a, int b, int c, int d){
    return a*b*c*d;
}
```

Rappel s'il y a changement du type de retour, il doit y avoir changement des types de paramètres.

2/ polymorphisme à l'exécution du programme.

Ce polymorphisme s'applique lorsqu'il y a héritage de structures. Il a lieu lorsqu'une structure fille redéfinit une méthode d'une autre façon que la structure mère avec les mêmes nom, paramètres et type de retour.

In [ ]:
```
struct Personne {
    int age;
    std::string nom;

    void modifierNom(std::string leNom){
        nom = leNom;
    }

    void modifierAge(int lage){
        age = lage;
    }


};

struct Etudiant : Personne {
    int cursus;

    void modifierAge(int lage){
        age = lage-10;
    }

};
```

Remarque Le polymorphisme à l'exécution se trouve aussi au moment de surcharger des opérateurs en c++.

Surcharge d'opérateurs

Que donne la compilation de ce programme?

In [ ]:
```
%%writefile Vector3.cpp

#include <iostream>
struct Vector3{
  float a;
  float b;
  float c;

  void init(float _a, float _b, float _c){
      a=_a;
      b=_b;
      c=_c;
  }
};
int main(){
  Vector3 v1;
  Vector3 v2;
  Vector3 v3;

  v3 = v1 * v2;

  std::cout << v3 <<std::endl;
  return 0;
}
```

```
Writing Vector3.cpp
```

In [ ]:
```
%%shell
g++ -std=c++14 -Wall -o vecTrois Vector3.cpp
./vecTrois
```

Pourquoi?

Et pourtant les instructions `int a =10; int b=2; int c = a*b;` peuvent être facilement compilées. Par ailleurs, l'instruction `float a =10.2; float b=2.5; float c = a*b;` doit avoir un autre comportement pour les flottants. L'opérateur de calcul du produit entre deux flottants est donc **surchargé**. La solution est de reproduire cette surcharge d'opérateur pour les nouveaux types que nous définissons. Pour l'opérateur `*` entre deux Vecteur3, cette surcharge peut ressembler à

In [ ]:
```
%%writefile Vector3.cpp
#include <iostream>
struct Vector3{
  float a;
  float b;
  float c;

  void init(float _a, float _b, float _c){
      a=_a;
      b=_b;
      c=_c;
  }

  Vector3 operator*(const Vector3 & v2) const{
      Vector3 resultat;
      resultat.a = a * v2.a;
      resultat.b = b * v2.b;
```

```
        resultat.c = c * v2.c;
        return resultat;
    }
};
int main(){
    Vector3 v1{3,4,5};
    Vector3 v2{4,2,-5};
    Vector3 v3;

    v3 = v1 * v2;
    std::cout <<"resultat = ("<< (v3.a) << ", "<< (v3.b) << ","<< (v3.c) << ")"
    return 0;
}
```

```
Overwriting Vector3.cpp
```

In [ ]:

```
%%shell
g++ -std=c++14 -Wall -o vecTrois Vector3.cpp
./vecTrois
```

```
resultat = (12, 8,-25)
```

Out[ ]:

Remarque : On doit pouvoir aussi exécuter l'opérateur de la façon suivante `v3 = v1.operator*(v2);`

In [ ]:

```
%%writefile Vector3.cpp
#include <iostream>

struct Vector3{
    float a;
    float b;
    float c;

    void init(float _a, float _b, float _c){
        a=_a;
        b=_b;
        c=_c;
    }

    Vector3 operator*(const Vector3 & v2) const{
        Vector3 resultat;
        resultat.a = a * v2.a;
        resultat.b = b * v2.b;
        resultat.c = c * v2.c;
        return resultat;
    }
};

int main(){
    Vector3 v1{3,4,5};
    Vector3 v2{4,2,-5};
    Vector3 v3;

    v3 = v1.operator*(v2);//v3=v1*v2
    // Vector v4;
    // v4 = v1*v2*v3;
    std::cout <<"resultat = ("<< (v3.a) << ", "<< (v3.b) << ","<< (v3.c) << ")"
    return 0;
}
```

```
Overwriting Vector3.cpp
```

In [ ]:
```
%%shell
g++ -std=c++14 -Wall -o vecTrois Vector3.cpp
./vecTrois
```

```
resultat = (12, 8,-25)
```

Out[ ]:

Le mot clé `const` indique au compilateur que la fonction membre ne modifie pas les propriétés de la variable v2. l'opérateur `&` dans les paramètres est une **référence** vers le paramètre courant. L'association des deux autorise l'accès à la variable sans faire de copie de la variable **tout** en garantissant que les propriétés du paramètre ne seront pas modifiées. Exemple possible avec pythonTutor.

D'autres opérateurs?

Pour afficher dans la sortie standard le contenu d'une variable `struct`, on peut ajouter une fonction membre `affiche`. Cette fonction accède aux membres de la structure et fait appel à `std::cout`. Mais tout comme `std::cout << a <<std::endl;` est valide lorsque `a` est entier ou flottant. On peut aussi surcharger l'opérateur << pour une variable de tout type (struct). Reprenons le programme

In [ ]:
```
%%writefile Vector3.cpp
#include <iostream>
struct Vector3{
  float a;
  float b;
  float c;

  void init(float _a, float _b, float _c){
      a=_a;
      b=_b;
      c=_c;
  }

  Vector3 operator*(const Vector3 & v2) const{
      Vector3 resultat;
      resultat.a = a * v2.a;
      resultat.b = b * v2.b;
      resultat.c = c * v2.c;
      return resultat;
  }
};


std::ostream& operator<<(std::ostream& fluxSortie, )



int main(){
  Vector3 v1{3,4,5};
  Vector3 v2{4,2,-5};
  Vector3 v3;
  v3 = v1 * v2;

  std::cout <<"resultat = ("<< v3 << ")"<<std::endl;
}
```

```
Overwriting Vector3.cpp
```

In [ ]:

```
%%shell
g++ -std=c++14 -Wall -o vecTrois Vector3.cpp
./vecTrois
```

et surchargeons l'opérateur <<. Voici le résultat pour cette structure.

In [ ]:

```
%%writefile Vector3.cpp

#include <iostream>
struct Vector3{
  float a;
  float b;
  float c;

  void init(float _a, float _b, float _c){
      a=_a;
      b=_b;
      c=_c;
  }

  Vector3 operator*(const Vector3 & v2) const{
      Vector3 resultat;
      resultat.a = a * v2.a;
      resultat.b = b * v2.b;
      resultat.c = c * v2.c;
      return resultat;
  }

};


std::ostream& operator<<(std::ostream& fluxSortie, const Vector3& v){
        fluxSortie << v.a << ',' << v.b<< ',' << v.c ;
        return fluxSortie;
}

int main(){
  Vector3 v1{3,4,5};
  Vector3 v2{4,2,-5};
  Vector3 v3;
  v3 = v1 * v2;

  std::cout << "v1 = (" << v1 << ")"<<std::endl;
  std::cout << "v2 = (" << v2 << ")"<<std::endl;

  std::cout << "resultat = (" << v3 << ")"<<std::endl;
  return 0;
}
```

```
Overwriting Vector3.cpp
```

In [ ]:

```
%%shell
g++ -std=c++14 -Wall -o vecTrois Vector3.cpp
./vecTrois
```

```
v1 = (3,4,5)
v2 = (4,2,-5)
resultat = (12,8,-25)
```

Out[ ]:

La liste des opérateurs pouvant être surchargés est disponible sur cette page wikipedia .