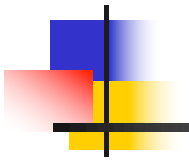
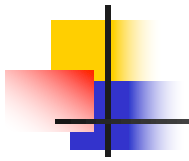


Programmation Orientée Objet: C++

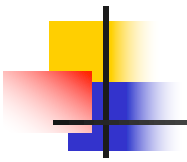
Quelques éléments de base
du langage C++



Objectifs du cours
Plan

- Objectifs du cours
  - Se rafraîchir la mémoire avec les fondamentaux du langage C …
  - … et assimiler certains concepts propres au langage C++, **dans le cadre d'une programmation structurée classique** (≠ programmation orientée objet)

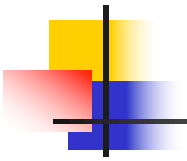
- Plan
  - Le langage C++
  - Les fonctions
  - Allocation dynamique de mémoire
  - Des fichiers source à l'exécutable
  - Exercice 1. Génération d'un exécutable
  - Exercice 2. Ajout de fichier et de fonction
  - Exercice 3. Passage par référence
  - Exercice 4. Surcharge de fonction
  - Exercice 5. Argument par défaut
  - Exercice 6. Allocation dynamique de mémoire



Le langage C++
Origine

- 1972: Langage C (AT&T Bell Laboratory)
  - Lexique assez réduit (opérateurs, type, mots clés)
  - Ensemble de fonctions standardisées regroupées en bibliothèques
  - Langage compilé
- 1982: Langage C++ (Bjarne Stroustrup / AT&T Bell Laboratory)
  - Ajoute au langage C ANSI les outils permettant de mettre en œuvre les principes fondamentaux de la Programmation Orientée Objet
  - La bibliothèque standard du C++ reprend et complète la bibliothèque standard du C
  - Langage hybride permettant de programmer en style C (appels de fonctions) et/ou en style orienté objet (manipulation d'objets)
  - Les normes successives introduisent de nouvelles fonctionnalités et font évoluer la bibliothèque standard

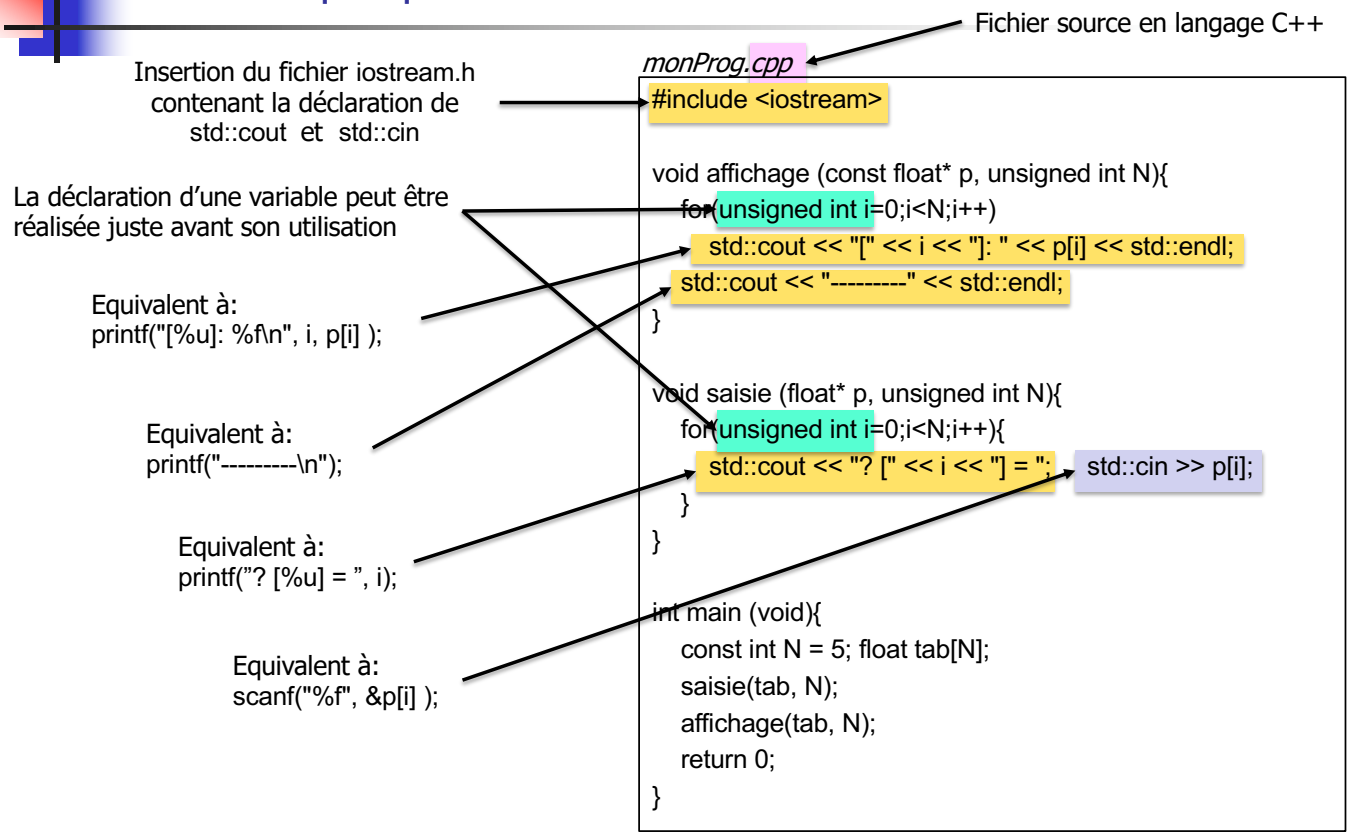
La description exhaustive de tous les mécanismes du langage n'étant pas envisageable, l'accent sera mis sur la compréhension de certains mécanismes du langage



Le langage C++
On retrouve la base du langage C …

- Un programme = succession d'appels de fonctions
  - Le point d'entrée: la fonction principale int main(void){ } et ses variantes
  - Le point de sortie: la dernière instruction de la fonction principale (ou return; ou exit();)
- Eléments de base des instructions d'une définition de fonction
  - Les variables et leurs types
    - int , float, char, …
    - int* , float*, char*, …
  - Les opérateurs
    - *, +, /, &, …
  - Les structures de contrôle
    - if, while, for, …
  - Les appels de fonctions
    - Les fonctions ne sont pas des éléments de base du langage et doivent donc être déclarées et définies
    - De nombreuses fonctions sont déjà définies dans des librairies

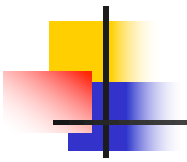
Le langage C++
… avec quelques subtilités



Les fonctions
Arguments

Principes de base valables quelque soit le mode de passage des arguments

- Paramètres d'entrée
  - Le contenu de chaque variable passée en argument d'entrée lors de l'appel de la fonction est copié dans le paramètre d'entrée correspondant
  - La fonction appelée n'a accès qu'à ses paramètres d'entrée ainsi qu'à ses variables locales
  - Les paramètres d'entrée de la fonction appelée, ainsi que ses variables locales, sont supprimés à la sortie du bloc d'instructions de la fonction appelée
- Paramètre de sortie
  - Le contenu du paramètre de sortie est copié dans la variable d'affectation de la fonction appelante (attention à sa validité !)

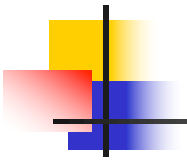


Les fonctions
Passage par valeur et par adresse

- Points clés du passage par valeur
  - La fonction appelée va exploiter la valeur numérique recopiée dans le paramètre d'entrée
  - Pertinent si la fonction appelée ne doit pas modifier l'argument d'entrée
  - Inadapté si l'argument d'entrée occupe beaucoup de place mémoire (ex: une structure ou un objet)

```
int foo (int data){
	data++;
	return data;
}

int main (void){
	int x = 5, y;
	y = foo(x);
	return 0;
}
```

Les fonctions
Passage par valeur et par adresse

- Points clés du passage par adresse
  - La fonction appelée va exploiter l'adresse mémoire recopiée dans le paramètre d'entrée
  - Le paramètre d'entrée de la fonction DOIT ETRE un pointeur pour pouvoir stocker l'adresse
  - Recours aux opérateurs d'indirection **&** et de déréférencement *
  - Pertinent si
    - l'argument d'entrée est une adresse (ex: début d'un tableau)
    - la fonction appelée doit modifier l'argument d'entrée
    - l'argument d'entrée occupe beaucoup de place mémoire (ex: une structure ou un objet)

```
void foo (int* data){
	(*data)++;
}

int main (void){
	int x = 5;
	foo(&x);
	return 0;
}
```

Les fonctions
Les références

- Les références sont des synonymes (ou alias) d'identificateurs de variables
  - La déclaration d'une référence ne crée pas une nouvelle variable mais permet de manipuler une variable sous un autre nom que celui sous lequel elle a été déclarée
  - **L'affectation d'une valeur à une référence revient à affecter cette valeur à la variable référencée**
  - Une référence **doit être initialisée** lors de sa déclaration car une référence ne peut pas être nulle
  - Une fois déclarée, une référence **ne peut plus être réaffectée**

- Déclaration d'une *référence* sur une *variable*
  - Syntaxe: *type* **&** *reference* **=** *variable* ;
  - Le *type* de la référence doit correspondre au type de la *variable* référencée

Les fonctions
Passage par référence

- Points clés du passage par référence
  - La fonction appelée va **accéder à la variable** passée en paramètre d'entrée
  - Le paramètre d'entrée de la fonction appelée DOIT ETRE une référence pour pouvoir stocker l'adresse de la variable
  - Pertinent si
    - la fonction appelée doit modifier l'argument d'entrée
    - l'argument d'entrée occupe beaucoup de place mémoire (ex: une structure ou un objet)

```
void foo (int& data){
    data++;
}

int main (void){
    int x = 5;
    foo(x);
    return 0;
}
```

Les fonctions
La surcharge

- Surcharge de fonction
  - Permet de définir plusieurs fonctions portant le même nom mais caractérisées par des paramètres d'entrée différents
  - La surcharge permet de créer plusieurs fonctions portant le même nom qui effectuent des tâches *similaires* mais sur des types de données différents
  - Lorsqu'une fonction surchargée est appelée, le compilateur sélectionne la fonction appropriée en se basant sur le nombre, l'ordre et le type des arguments figurant dans l'appel

```
void foo (int a, float b){
    // Des opérations …
}
void foo (int a){
    // D'autres opérations …
}
int main (void){
    int x = 5, float y = 3.14;
    foo(x);
    foo(x, y);
    return 0;
}
```

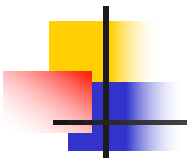
? En conservant le code existant, est-il possible de définir :

```
float foo (int& z){
    // Encore d'autres opérations …
}
```


Les fonctions
Les arguments par défaut

- Les arguments par défaut
  - Permettent d'appeler une fonction sans lui fournir tous les arguments d'entrée attendus
  - Lorsqu'un argument par défaut est omis dans l'appel de la fonction, la valeur par défaut spécifiée dans le code (constante, variable globale ou appel de fonction) est insérée automatiquement par le compilateur
  - Les arguments par défaut doivent être spécifiés avec la première occurrence du nom de la fonction (habituellement dans le prototype) avec pour chaque paramètre sa valeur par défaut précédée d'un signe d'affectation
  - Les arguments par défaut doivent être les plus à droite dans la liste des paramètres.

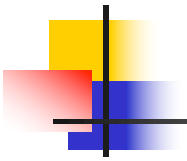
```
void foo (int a, float b =10);
void foo (int a, float b){
    // Des opérations …
}
int main (void){
    int x = 5, float y = 3.14;
    foo(x);
    foo(x, y);
    return 0;
}
```



Allocation dynamique de mémoire
La pile d'exécution et le tas

- Rappel du bilan de l'allocation mémoire dans le segment `stack` (ou pile d'exécution)
  - Les variables locales sont allouées lors de l'entrée dans une fonction et désallouées automatiquement lors de la sortie de la fonction
  - Il n'y a pas de fuites mémoires
  - La taille du segment est assez limitée
  - La taille de certaines variables peut n'être connue que lors de l'exécution du programme
  - Il est impossible d'allouer ou de libérer de la mémoire à des moments arbitraires de l'exécution

- Solution: allocation mémoire dans le segment `heap` (ou tas)
  - L'allocation et la libération de la mémoire sont de la responsabilité du programmeur et peuvent être réalisées à des moments arbitraires de l'exécution du processus
    - Contrôle de la durée de vie des variables (i.e. des zones mémoires allouées)
  - La taille du segment est "illimitée"
  - Du fait d'erreurs de programmation, des fuites mémoire peuvent survenir et provoquer
    - Une perte de performance et un ralentissement croissant de l'application (mémoire morcelée)
    - Le blocage d'autres applications car elle accapare toute la mémoire du système …



Allocation dynamique de mémoire
Gestion des variables dans le tas

- Allocation dynamique: **new** (remplace malloc)
  - Opérateur unaire: *pointeur* = **new** *type* ou *pointeur* = **new** *type*[*n*] dans le cas d'un tableau
  - Nécessite un *type* et retourne un *pointeur* du type sur l'emplacement mémoire alloué, ou sur le premier élément des *n* éléments du tableau
  - En cas d'échec, déclenche une exception de type **bad_alloc**

```
int main (void){
    float* x, int N = 5;

    x = new float[N];

    for(int i=0; i<N; i++)
        x[i] = i;

    delete[] x;
    return 0;
}
```

Allocation dynamique de mémoire
Gestion des variables dans le tas

- Libération dynamique: **delete** et **delete[]** (remplacent free)
  - Opérateur unaire: **delete** pointeur ou **delete[ ]** pointeur dans le cas d'un tableau
  - Libère un emplacement mémoire alloué par **new** (ne le faire qu'une fois !!!) et pointé par *pointeur*

```
int main (void){
      float* x, int N = 5;

      x = new float[N];

      for(int i=0; i<N; i++)
            x[i] = i;

      delete[] x;
      return 0;
}
```


Des fichiers source à l'exécutable
Répartition du code source

- La totalité des instructions peut se trouver dans un seul fichier
  - Pourquoi n'est ce pas judicieux ???

- Répartition classique des instructions dans plusieurs fichiers source
  - Eviter les variables globales …
  - Les déclarations de fonctions (ou prototypes) sont placées dans des fichiers d'extension *.h*
    - Les types des paramètres doivent figurer dans le prototype
    - Le type de la valeur de retour par défaut est **int**
  - Les définitions de fonctions sont placées dans des fichiers d'extension *.cpp*
    - Il faut concordance dans le nombre, l'ordre et les types des paramètres spécifiés dans la déclaration
    - La définition d'une fonction joue également le rôle de déclaration
    - Tout appel de fonction requière la déclaration préalable de la fonction (directive de précompilation **#include**)

Des fichiers source à l'exécutable
Répartition du code source

monProg.cpp

```
#include <iostream>

void affichage (const float* p, unsigned int N){
    for(unsigned int i=0;i<N;i++)
        std::cout << "[" << i << "]: " << p[i] << std::endl;
    std::cout << "---------" << std::endl;
}

void saisie (float* p, unsigned int N){
    for(unsigned int i=0;i<N;i++){
        std::cout << "? [" << i << "] = "; std::cin >> p[i];
    }
}

int main (void){
    const int N = 5; float tab[N];
    saisie(tab, N);
    affichage(tab, N);
    return 0;
}
```

mesFcts.cpp

```
#include "mesFcts.h"
#include <iostream>

void affichage(const float* p, unsigned int N){
    for(unsigned int i=0;i<N;i++)
        std::cout << "[" << i << "]: " << p[i] << std::endl;
    std::cout << "---------" << std::endl;
}

void saisie(float* p, unsigned int N){
    for(unsigned int i=0;i<N;i++){
        std::cout << "? [" << i << "] = ";
        std::cin >> p[i];
    }
}
```

main.cpp

```
#include "mesFcts.h"

int main(void){
    const int N = 5;
    float tab[N];

    saisie(tab, N);
    affichage(tab, N);

    return 0;
}
```

mesFcts.h

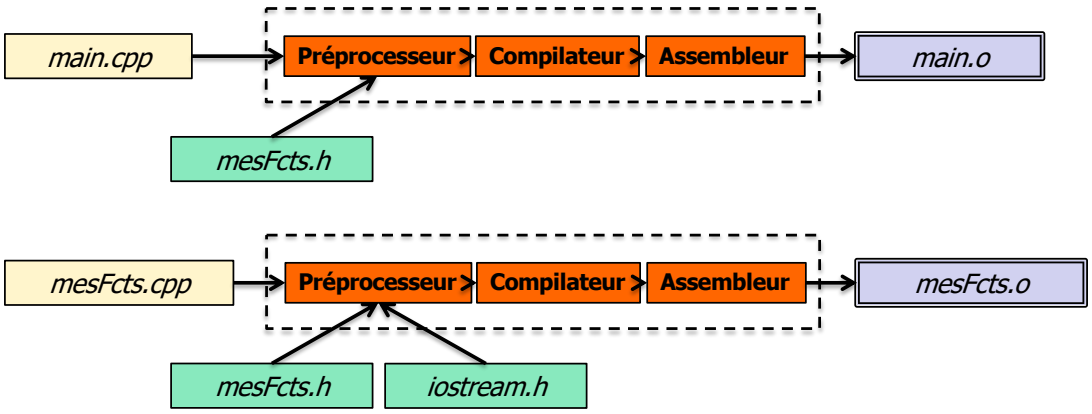
```
#ifndef MESFCTS_H
#define MESFCTS_H

void affichage(const float* p, unsigned int N);
void saisie(float* p, unsigned int N);

#endif
```


Des fichiers source à l'exécutable
Code source ➔ code machine

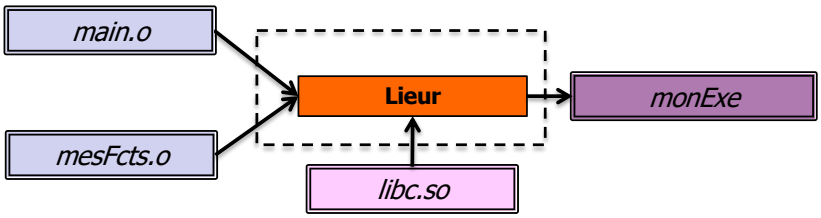
- Chaque fichier *.cpp* est compilé et assemblé indépendamment des autres fichiers *.cpp*



L'assemblage d'un fichier source nécessite la **déclaration** (*i.e.* le prototype) de toutes les fonctions appelées

Des fichiers source à l'exécutable
Codes machine ➔ code machine exécutable

- Le lieur exploite tous les fichiers assemblés ainsi que les bibliothèques nécessaires afin de générer un fichier exécutable



La génération d'un programme exécutable nécessite la **définition** de toutes les fonctions appelées (sous forme de fichier source, objet ou librairie)

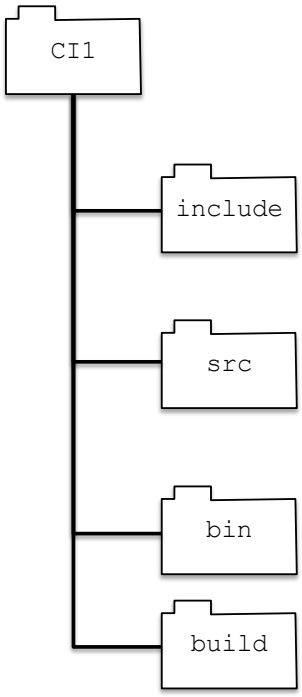
- Si un fichier de la chaîne est modifié, il est nécessaire de régénérer tous les fichiers qui en dépendent (mais seulement eux …)

Des fichiers source à l'exécutable
Une bonne organisation et des outils logiciels

- Utiliser une organisation rationnelle pour gérer les fichiers constituant un programme …
  - Un répertoire dédié aux fichiers *.cpp*
  - Un répertoire dédié aux fichiers *.h*
  - Un répertoire dédié au fichier exécutable
  - Un répertoire dédié aux fichiers "intermédiaires" générés automatiquement
  - Un fichier de configuration *CMakeLists.txt*
    - Répertoires à explorer
    - Options de compilation
- … et des outils performants pour générer le fichier exécutable
  - `cmake`
    - Exploite le fichier de configuration *CMakeLists.txt* afin de générer le fichier *makefile* correspondant
  - `make`
    - Exploite le fichier *makefile* afin de générer le fichier exécutable

Exercice 1. Génération d'un exécutable
Marche à suivre générale …

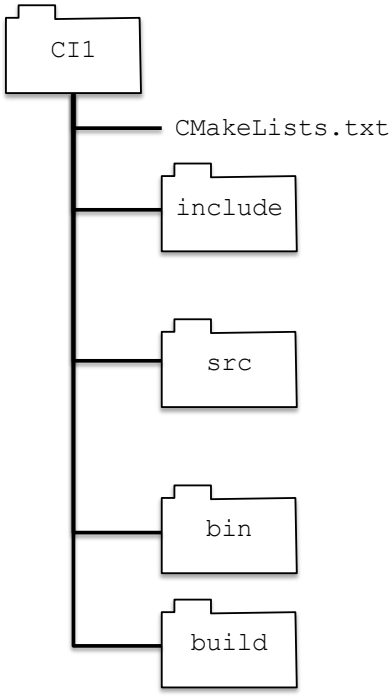
1) Depuis un répertoire dédié au programme, créer les répertoires
   - `src`, dédié aux fichiers *.cpp*
   - `include`, dédié aux fichiers *.h*
   - `bin`, dédié au fichier exécutable
   - `build`, dédié à certains fichiers intermédiaires générés automatiquement



Exercice 1. Génération d'un exécutable
Marche à suivre générale …

2) Créer (ou copier) un fichier *CMakeLists.txt* dans le répertoire dédié au programme

- Ajuster certains paramètres si nécessaires
  - Nom du fichier exécutable
  - Options de compilation



Exercice 1. Génération d'un exécutable
Marche à suivre générale …

CMakeLists.txt

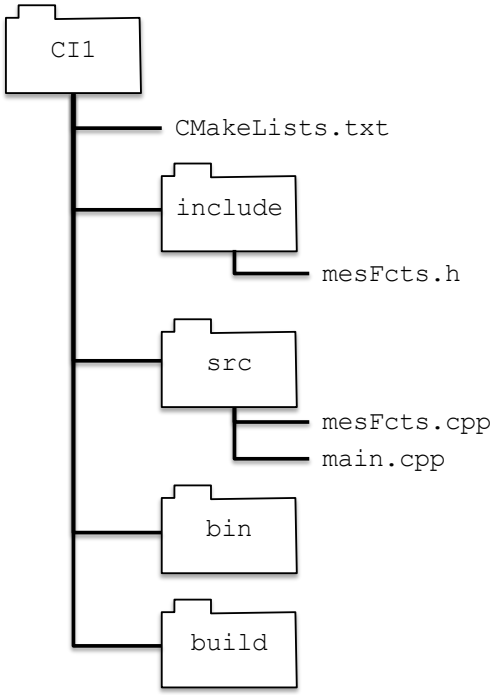
```
# S'assure que la version de make est suffisamment récente
cmake_minimum_required(VERSION 2.6)
# Indique le nom du projet
project(demo_cpp)
# Ajoute le répertoire include des fichiers .h au chemin de recherche des #include "xx.h"
include_directories(include)
# Impose le répertoire de création du fichier exécutable
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/../bin)
# Impose certaines options de compilation
set(CMAKE_C_FLAGS "-g -Wall -Wextra -ansi -pedantic-errors")
# Génère la liste des fichiers .cpp devant être compilés à partir des fichiers présents dans le répertoire src
file(
   GLOB_RECURSE
   srcFiles
   src/*.cpp
)
# Crée l'exécutable monExe à partir de la liste des fichiers .cpp
add_executable(
   monExe
   ${srcFiles}
)
```


Exercice 1. Génération d'un exécutable
Marche à suivre générale …

3) Copier/créer les fichiers sources de votre programme dans leurs répertoires respectifs

! Ne pas indiquer de chemin dans les directives `#include "xxx.h"`

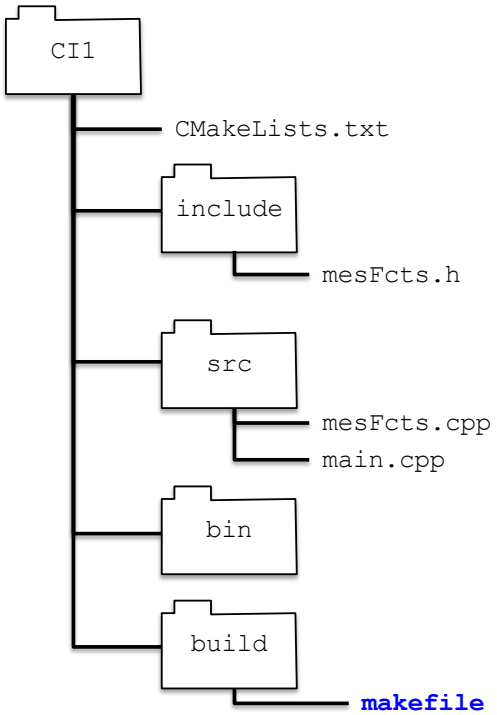
! Tous les fichiers *.cpp* du répertoire `src` seront compilés



Exercice 1. Génération d'un exécutable
Marche à suivre générale …

4) Générer le fichier *makefile* correspondant à votre programme à l'aide de `cmake` en spécifiant le répertoire contenant le fichier *CMakeLists.txt*

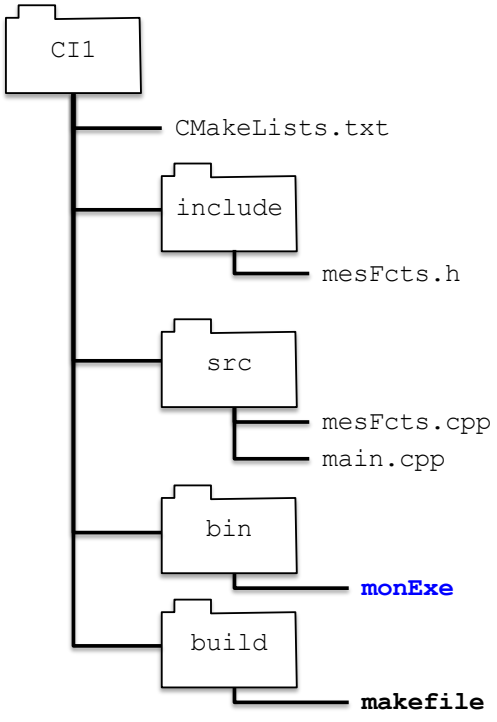
- Depuis un terminal
  - Se placer dans le répertoire `build` en utilisant la commande `cd`
  - Lancer la commande `cmake ..`
  - Quels sont les fichiers/répertoires créés ?



Exercice 1. Génération d'un exécutable
Marche à suivre générale …

5) Générer le fichier exécutable correspondant à votre programme à l'aide de `make` depuis le répertoire contenant le fichier *makefile*

- Depuis un terminal
  - Si nécessaire, se placer dans le répertoire `build` en utilisant la commande `cd`
  - Lancer la commande `make`
  - Quels sont les fichiers/répertoires créés ?



Exercice 1. Génération d'un exécutable
Marche à suivre générale …

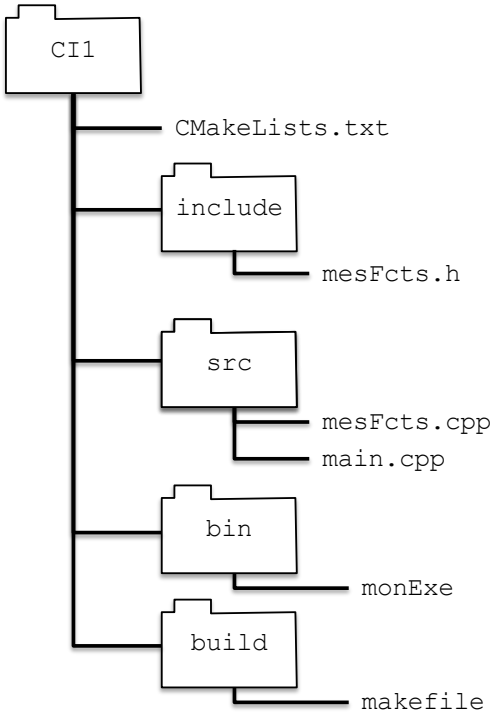
6) Depuis un terminal, exécuter le programme *monExe* généré précédemment
   - Lancer la commande `./monExe` depuis le répertoire `bin`

   ou

   la commande `../bin/monExe` depuis le répertoire `build`

Exercice 1. Génération d'un exécutable
Que faire si ?

- Je modifie le code de l'un des fichiers source (*.cpp* ou *.h*) de mon programme
- Je veux retirer ou ajouter un fichier source (*.cpp* ou *.h*) à mon programme
- Je veux changer les options de compilation de mon programme
- Je veux écrire un nouveau programme



Exercice 2. Ajout de fichier et de fonction
Définir et appeler une nouvelle fonction

- Ecrire le code source de la fonction `moyenne` suivante, et le répartir dans deux nouveaux fichiers *mesAutresFcts.h* et *mesAutresFcts.cpp*
  - `moyenne`

    Calcule la moyenne des éléments d'un tableau de réels

    Entrées:

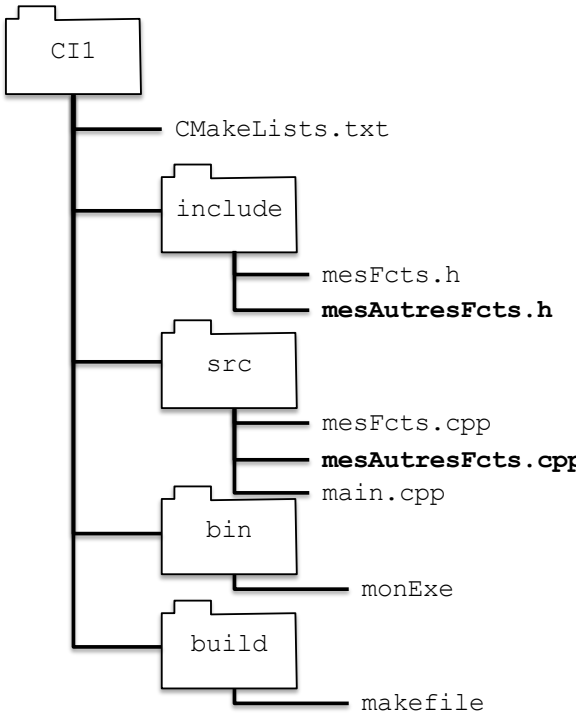
    *- p : **adresse** de début du tableau*

    - N : nombre d'éléments du tableau

    Sortie:

    - Moyenne des éléments

- Mettre à jour l'arborescence du projet
- Modifier le code du fichier *main.cpp* afin d'exploiter la fonction `moyenne`
- Générer l'exécutable et tester



Exercice 3. Passage par référence
Définir et appeler une nouvelle fonction

- Ecrire le code source de la fonction `rechercheMin` suivante, et le répartir dans les fichiers *mesAutresFcts.h* et *mesAutresFcts.cpp*
  - `rechercheMin`

    Recherche le plus petit élément d'un tableau ainsi que l'indice de sa première occurrence

    Entrées:

    *- p : **adresse** de début du tableau*

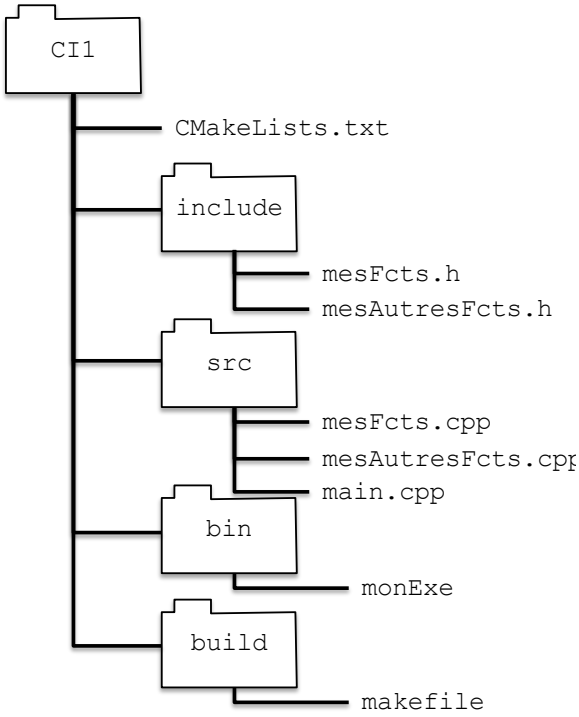
    - N : nombre d'éléments du tableau

    *- min : **référence** sur le plus petit élément*

    *- indMin : **référence** sur l'indice de la première occurrence du plus petit élément*

    Sortie: Aucune

- Modifier le code du fichier *main.cpp* afin d'exploiter la fonction `rechercheMin`
- Générer l'exécutable et tester



Exercice 4. Surcharge de fonction
Définir et appeler une nouvelle fonction

- Ecrire le code source de la fonction `rechercheMin` suivante, et le répartir dans les fichiers *mesAutresFcts.h* et *mesAutresFcts.cpp*
  - `rechercheMin`

    Recherche le plus petit élément d'un tableau, l'indice de sa première occurrence, ainsi que son nombre d'occurrences.

    Entrées:

    *- p : **adresse** de début du tableau*

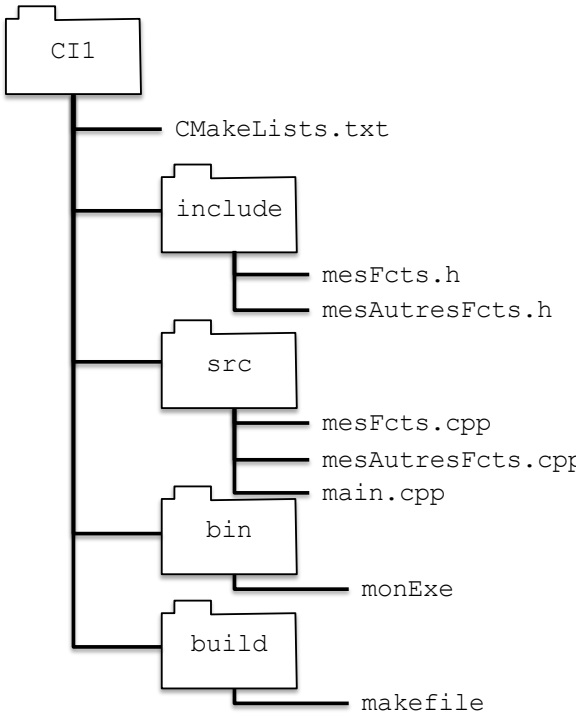
    - N : nombre d'éléments du tableau

    *- min : **référence** sur le plus petit élément*

    *- indMin : **référence** sur l'indice de la première occurrence du plus petit élément*

    *- nb : **référence** sur le nombre d'occurrences du plus petit élément*

    Sortie: Aucune
- Modifier le code du fichier *main.cpp* afin d'exploiter la fonction `rechercheMin`
- Générer l'exécutable et tester



Exercice 5. Argument par défaut
Modifier et appeler une fonction

- Modifier le code source de la fonction `affichage` de telle sorte qu'elle puisse, au choix, afficher également la moyenne des éléments du tableau
  - `affichage`

    Affiche à l'écran successivement tous les éléments d'un tableau (avec leur indice), puis éventuellement leur moyenne

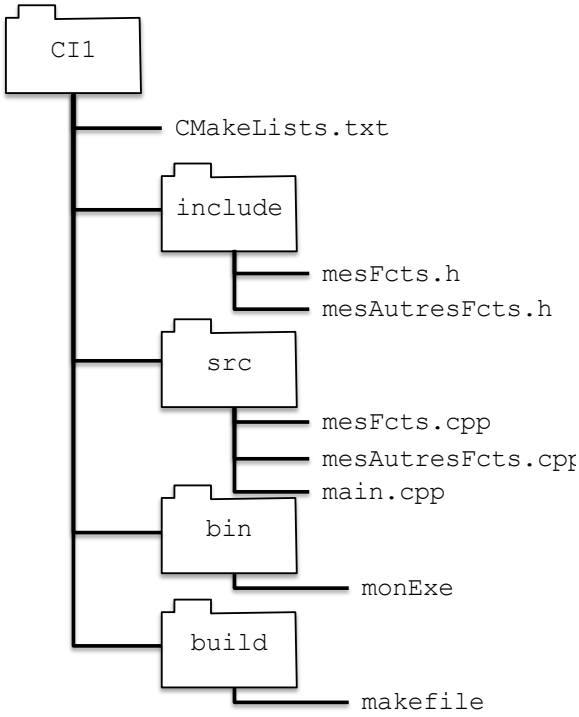
    Entrées:

    *- p : **adresse** de début du tableau*

    - N : nombre d'éléments du tableau

    - visu : booléen sélectionnant le calcul et l'affichage de la moyenne (par défaut : `false`)

    Sortie: Aucune
- Modifier le code du fichier *main.cpp* afin d'exploiter la fonction `affichage`
- Générer l'exécutable et tester



Exercice 6. Allocation dynamique de mémoire
Définir et appeler une nouvelle fonction

- Ecrire le code source de la fonction `copieTab` suivante, et le répartir dans les fichiers *mesFcts.h* et *mesFcts.cpp*
  - `copieTab`

    Crée et retourne une copie d'un tableau de réels passé en entrée

    Entrées:

    *- p : **adresse** de début du tableau*

    - N : nombre d'éléments du tableau à copier

    Sortie:

    *- **adresse** de début du tableau créé*
- Modifier le code du fichier *main.cpp* afin d'exploiter la fonction `copieTab`
  - Libérer la mémoire allouée à la fin du programme
  - Comment créer une copie partielle d'un tableau (i.e. uniquement les éléments d'indices compris dans une plage d'indices) ?
- Générer l'exécutable et tester