Programmation multifichier

420-C21-IN Programmation II

Godefroy Borduas – Automne 2021

Module 6

1

Comprendre l'incidence de nos déclarations

- Principe de visibilité, de portabilité et de durée de vie des variables et fonctions
- Classe de mémorisation des variables et d'une fonction
- Les dangers d'une déclaration globale
- Division en fichier d'en-tête (*header*) et source (*cpp*)

2

Petite définition pour la route

Un programme est composé de modules. Ce dernier est décrit dans un fichier source Indépendant des autres modules.

Lorsque les modules sont réunis, nous appelons ceci un projet.

3

Principe de visibilité, de portabilité et de durée de vie des variables et fonctions

4

Petit rappel sur l'utilisation des variables et des fonctions

- Avant d'être utilisée, une fonction ou une variable doit être déclarée
- Pour une fonction, la déclaration est décrite par le prototype
- Pour une variable, la définition correspond à la déclaration
  - Demander à créer une variable sans l'instancier correspond à la déclaration
- Cette déclaration aura un impact sur le **comportement** de la variable en mémoire
  - Ceci influence la **durée de vie**, la **visibilité** et la **portée**

5

La durée de vie

- La durée de vie correspond au temps que la variable passe en mémoire
- Elle est **permanente** lorsque la variable existera en mémoire jusqu'à la fin du programme
- Elle est **temporaire** lorsque la variable existera en mémoire jusqu'à une fin donnée
  - Cette est annoncée par la fin du bloc où la variable est déclarée

6

Quelle est la durée de vie de chaque variable ?

Dans ce cas, toutes les variables sont **temporaires**.

```
int main(void) {
    int i = 42;
    if (i > 0) {
        int j = 0;
   (1)  i += j;
    } else {
        int k = 8;
        if (k < 10) {
            int l = 10;
            k *= l;
      (2)   i += k;
        }
      (3)
    }
   (4)
}
```

Variable	Fin de vie
i	Après (4)
j	Après (1)
k	Après (3)
l	Après (2)

7

Visibilité et portée

- La visibilité détermine où l'objet (variable ou fonction) est **utilisable**
- La portabilité détermine où l'objet est **connu**

Un objet ne sera jamais visible dans les endroits où il n'est pas connu.

8

Priorité de déclaration

La priorité sur une variable est toujours pour celle qui a été déclarée dans le bloc le plus proche

9

Comment déterminer la portée et la visibilité

- Deux caractéristiques sont nécessaires pour y parvenir

1. La position de la déclaration et de la définition
   - Dans quel bloc ?
   - Dans une fonction ?
   - Dans un fichier ?
2. La classe de mémorisation

10

Classe de mémorisation des variables et d'une fonction

11

Qu'est-ce qu'une classe de mémorisation ?

- Il existe quatre classes de mémorisation
  - Auto
  - Extern
  - Static
  - Register
- Chaque classe a un effet différent qui dépend de l'instruction
  - L'effet sera différent si c'est une définition ou une déclaration
  - L'effet sera aussi si c'est différent une variable ou une fonction
- Le niveau d'accès (interne au module ou externe) influence les classes de mémorisation
  - Il existe des comportements par défaut

12

Comportement par défaut

Type instruction	Niveau	Durée de vie	Visibilité
Définition de variable	Externe	Permanente	Dans tous les modules
	Interne	Temporaire	Bloc
Paramètre formel (variable)		Temporaire	Dans la fonction où elle est déclarée
Définition de fonction	Externe	Permanente	Dans tous les modules
Déclaration de fonction	Externe	Permanente	Module

13

La classe AUTO

- Classe par défaut de toute variable déclarée dans un bloc
- Les paramètres d'une fonction sont de type auto

Une déclaration externe est toujours faite à **l'extérieur** de **tous blocs**.

Type instruction	Niveau	Durée de vie	Visibilité
Définition de variable	Interne	Temporaire	Bloc
Paramètre formel	Interne	Temporaire	Dans la fonction où elle est déclarée

14

La classe REGISTER

- Accessible grâce au mot clé **register**
- Permets un accès rapide à la valeur de la variable
  - Pratique pour les variables à usage intense
- La variable sera affectée aux registres du processeur et non dans la pile mémoire
- La méthode est désuète. Les compilateurs appliquent des optimisations plus efficaces aujourd'hui.

Type instruction	Niveau	Durée de vie	Visibilité
Définition de variable	Interne	Temporaire	Bloc

15

La classe STATIC

- Accessible grâce au mot clé **static**
- Ne modifie pas la visibilité ou la porte
- Change la durée de vie d'une variable
- Permets de limiter la visibilité des variables globale au module uniquement

Type instruction	Niveau	Durée de vie	Visibilité
Définition de variable	Externe	Permanente	Module
	Interne	**Permanente**	Bloc
Définition d'une fonction	Externe	Permanente	Module

16

Comportement des classes AUTO, REGISTER et STATIC

Les classes **AUTO**, **REGISTER** et **STATIC** sont indépendantes et ils ne peuvent pas être appliqués à la même variable.

17

Classe de mémorisation des variables et d'une fonction

Le cas particulier de **EXTERN**

18

La déclaration de variable ou de fonction

La déclaration d'une variable ou d'une fonction (un objet) prépare le compilateur à la définition. Cette action réserve le type de l'objet. Ceci ne réserve pas l'espace mémoire.

La définition d'une fonction passe par son prototype.

19

La définition de variable ou de fonction

La définition permet de réserver l'espace mémoire pour une variable ou une fonction.

20

La classe EXTERN

- Accessible grâce au mot clé **extern**
- Permets de rendre une variable ou une fonction accessible à tous les modules

La définition d'une variable globale sert aussi de déclaration pour celle-ci jusqu'à la fin du fichier.

Type instruction	Niveau	Durée de vie	Visibilité
Déclaration de variable	Externe	Permanente	Module

21

Les dangers d'une déclaration globale

22

Tout le monde y a accès

- Dès que la variable est globale (au module ou au programme), son accès n'est plus sécurisé
- Toutes les fonctions peuvent modifier son code
- Ceci représente un danger de corruption des données
  - Voir le code **danger.cpp**

23

Division en fichier d'en-tête (*header*) et source (*cpp*)

Créer un fichier *header*

24

Le fichier *header*

- Contiens les prototypes de vos fonctions et les définitions de vos structures
- Réduis la taille de votre code principal
- Permets de réutiliser votre code et de partager une même fonction dans plusieurs fichiers sources
- Les fichiers *headers* ont toujours l'extension **h**
- Ajouter à un fichier source (cpp) grâce à l'instruction #include

25

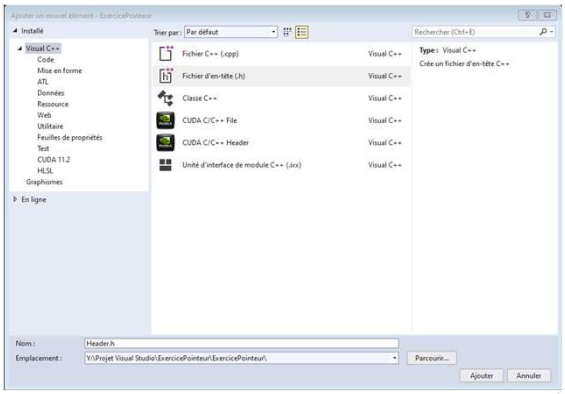
Ajouter un fichier d'en-tête

- Pour ajouter dans un projet, allez dans l'**Explorateur de source**. Sur le répertoire **Fichier d'en-tête** pour votre projet, faites un clic droit et choisissez **Ajouter/Nouvel élément**

26

Ajouter un fichier d'en-tête

- Sélectionner le type de fichier **Fichier d'en-tête**
- Donnez-lui un nom unique au projet
  - Le nom doit unique parmi tous les fichiers d'en-tête
- Cliquez sur ajouter



27

Exemple, ExtremeTableau.h

- Reprenons l'exercice 9 sur les pointeurs
- Dans notre fichier d'en-tête, nous ajoutons notre structure et le prototype de fonction maxmin
  - Regarder l'exemple **extremetableau.h**
- Votre fichier d'en-tête contient uniquement vos déclarations de fonction, les définitions de structure et les déclarations des variables globales
- Il est le « mode d'emploi » du fichier source.
- La définition de vos fonctions aura lieu dans votre fichier source.
  - Pour l'instant, notre seul fichier source est **main.cpp**

28

Ajouter son *header* au fichier source

- Maintenant que le fichier *header* est créé, il faut informer le compilateur que le fichier existe et que nous l'utilisons
- C'est un peu la même chose lorsqu'on crée un nouveau mot comme phallophoto, il faut préciser dans quel dictionnaire il est décrit.
- L'ajout est réalisé grâce à l'instruction de préprocessus `#include "nom et chemin du fichier header.h"`
- Toutefois, contrairement à iostream, nous devons utiliser les guillemets
  - Les guillemets indiquent au compilateur de rechercher le fichier dans le projet (ou à l'emplacement décrit) au lieu de la bibliothèque standard

29

Chemin relatif VS Chemin absolu

- Le chemin **relatif** permet de trouver un fichier à partir du dossier où se trouve votre application
  - Exemple de chemin relatif :
  - Votre application est dans le dossier :
    C:\User\ArthurDent\VS\projet
  - Votre fichier est dans le dossier :
    C:\User\ArthurDent\VS\projet\src\code\DontPanic.h
  - Le chemin relatif sera alors :
    C:\User\ArthurDent\VS\projet
    ~~C:\User\ArthurDent\VS\projet~~\src\code\DontPanic.h
    .\src\code\DontPanic.h
  - Le point représente le dossier de votre application
- Le chemin **absolu** permet de trouver un fichier à partir de la racine du disque (ex. le lecteur C)
  - Ceci est un chemin absolu
    C:\User\ArthurDent\VS\projet\src\code\DontPanic.h

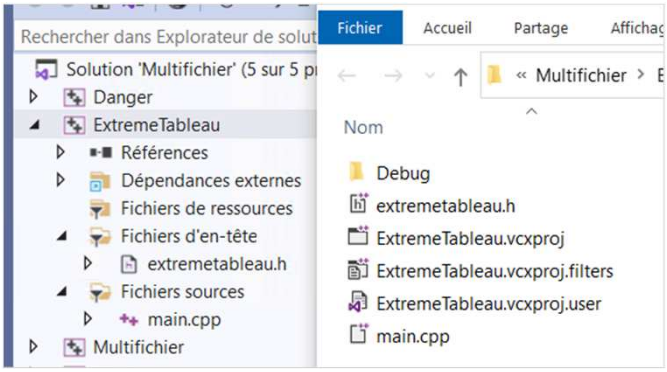
30

Avantage du chemin relatif

- Le chemin relatif ne dépend pas du disque et des configurations de votre système
- Il dépend uniquement de la structure de votre projet

31

La position des fichiers header créée avec Visual Studio



- Les fichiers sont ajoutés dans le même dossier que vos autres fichiers sources et *header* de votre projet
- Exemple : Le dossier du projet **ExtremeTablau**

32

Impact dans votre fichier source

- Après l'inclusion de **iostream** (et avant **usign namespace std**) ajouter l'instruction

```
#include "extremetableau.h"
```

  - Les autres inclusions de la bibliothèque standard (iostream, ctime, cstdlib, etc.) restent essentielles
- Comme les fichiers *headers* incluent uniquement les déclarations, vous devez définir vos fonctions dans le fichier source
  - Les fichiers sources ont toujours l'extension **cpp**
- Pour l'instant, notre fichier source est **main.cpp**

33

Division en fichier d'en-tête (*header*) et source (*cpp*)

Créer un fichier source

34

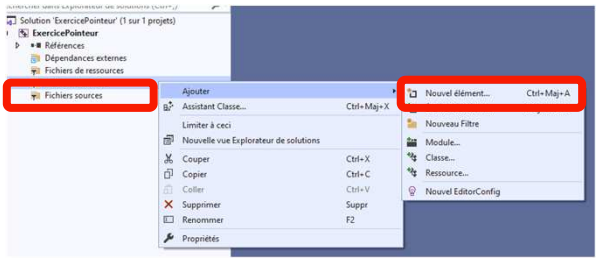
Le fichier source

- Contiens les définitions des fonctions
- Réduis la taille de votre code principal
- Permets de réutiliser votre code et de partager une même fonction dans plusieurs fichiers sources
- C'est le compilateur qui s'occupe de lier les fichiers sources ensemble
- Les fichiers sources ont toujours l'extension **cpp**

35

Ajouter un fichier source

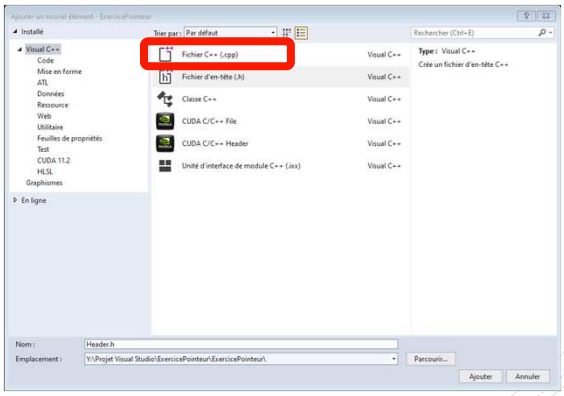
- Pour ajouter dans un projet, allez dans l'**Explorateur de source**. Sur le répertoire **Fichier source** pour votre projet, faites un clic droit et choisissez **Ajouter/Nouvel élément**



36

Ajouter un fichier source

- Sélectionner le type de fichier **Fichier C++**
- Donnez-lui un nom unique au projet
  - Le nom doit unique parmi tous les fichiers source
  - Par convention, le fichier source partage le même nom que le fichier d'en-tête lié
- Cliquez sur ajouter



37

Exemple, ExtremeTableau.h

- Reprenons l'exercice 9 sur les pointeurs
- Dans notre fichier source, nous ajoutons la définition de notre fonction maxmin
  - Regarder l'exemple **extremetableau.cpp**
- Votre fichier d'en-tête contient toutes les définitions de fonction et il contient les « actions » de votre fichier d'en-tête
- Si le fichier d'en-tête est un « mode d'emploi », le fichier source est « l'application » du mode d'emploi

38

Ajouter son *header* au fichier source

- Incluez toujours le fichier d'en-tête lié à notre fichier source
  - Cela permet au compilateur de vérifier le lien entre les définitions et les déclarations
- L'ajout est réalisé grâce à l'instruction de préprocessus **`#include "nom et chemin du fichier header.h"`**
- Toutefois, contrairement à iostream, nous devons utiliser les guillemets
  - Les guillemets indiquent au compilateur de rechercher le fichier dans le projet (ou à l'emplacement décrit) au lieu de la bibliothèque standard

39

Ajout des fichiers d'en-tête standard et autre

- Après l'inclusion des fichiers d'en-tête nécessaire pour le fichier source (et avant **usign namespace std**) ajoutez l'instruction

```
#include "extremetableau.h"
```

  - Les autres inclusions de la bibliothèque standard (iostream, ctime, cstdlib, etc.) restent essentielles
  - Vous ajoutez uniquement ceux qui sont utilisés **dans ce fichier source**
- Si votre fichier source utilise des déclarations provenant d'autre fichier d'en-tête, vous devez les inclure !

40

Protéger son fichier d'en-tête

- Lorsque le compilateur arrive sur une instruction d'inclusion (`#include`), ce dernier copie tout le fichier à l'intérieur du fichier source
- Cette opération est invisible
- Toutefois, le compilateur n'apprécie pas lorsqu'une définition est déclarée deux fois
- En somme, on ajoute les déclarations qu'une seule fois par problème.

41

L'instruction #define

- L'instruction de préprocesseur `#define` permet de définir une constante
- Après l'instruction `#define VALEUR 3`, le mot VALEUR sera remplacé par 3 dans le code

42

La paire d'instructions #ifndef - #endif

- L'instruction `#ifndef` permet de vérifier si une constante **n'a pas été** définie
  - L'instruction `#ifdef` permet de vérifier l'inverse
- La syntaxe est toujours

  #ifndef <Nom de la constante>
- Si l'instruction est vraie (c.-à-d. que la constante **n'est pas définie**), alors le compilateur va ajouter (ou compiler) toutes les instructions jusqu'à `#endif`
- `#endif` ferme le bloc commencer par `#ifndef`
  - `#endif` est **obligatoire** et toujours **après** le code

43

Protéger son fichier d'en-tête avec #define et #ifndef

- Ajouter ce code à votre fichier d'en-tête

```
#ifndef NOM_FICHIER_H
#define NOM_FICHIER_H
// Ajouter vos déclarations
#endif
// N'ajoutez rien ici
```

- Cette instruction empêche au compilateur d'ajouter deux fois les déclarations du fichier d'en-tête
- La première instruction vérifie que le NOM_FICHIER_H n'est pas défini
  - Si c'est vrai, alors on continue les autres instructions
- La deuxième instruction définie NOM_FICHIER_H
  - Nous n'avons pas besoin de connaître la valeur. Nous voulons seulement que le nom existe
  - Une fois définie, l'instruction précédente sera toujours fausse

44

L'impact

- Grâce au code précédent, le fichier d'en-tête sera toujours inclus qu'une seule fois
  - C'est la définition de MON_FICHIER_H après le #ifndef qui permet d'inclure le fichier d'en-tête qu'une seule fois
- Par convention, le nom de la constante sera le nom du fichier dans la notation SNAKE_CASE suivi de _H
- Regardez l'exemple complet **ExtremeTableauMultisource**

45

Petit mot sur #prama once

- Instruction **non standard** qui s'assure que le fichier ne soit inclus qu'une seule fois
- Il faut vérifier que le compilateur le supporte avant de l'utiliser
- Dans le cadre du cours, nous resterons avec la méthode standard (#ifndef)

46

Exercice

Utilisez la programmation multifichier

- Créer un fichier source qui gère un vecteur mathématique en 2 dimensions
- Les vecteurs sont décrits par deux nombres flottants (x et y)
- Les opérations sur les vecteurs vont comme suit
  - Addition : on addition les composantes x ensembles et les composantes y ensemble
  - Soustraction : on soustrait les composantes x ensembles et les composantes y ensemble
  - Produit scalaire : on multiplie les composantes x ensembles et les composantes y ensemble. On addition le tout après
  - Norme : (s'applique sur un seul vecteur) on utilise la formule suivante :

$$norme = \sqrt{x^2 + y^2}$$

47