Immersion - Jour 3

Bonnes pratiques

Jonathan Weber et Germain Forestier

Automne 2023

Pour l'instant vous savez

- Manipuler et parcourir des fichiers dans le shell

Pour l'instant vous savez

- Manipuler et parcourir des fichiers dans le shell
- Compiler et exécuter un programme écrit en C

Pour l'instant vous savez

- Manipuler et parcourir des fichiers dans le shell
- Compiler et exécuter un programme écrit en C
- Ouvrir et traiter un fichier CSV en C

Pour l'instant vous savez

- Manipuler et parcourir des fichiers dans le shell
- Compiler et exécuter un programme écrit en C
- Ouvrir et traiter un fichier CSV en C

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

⇒ Découper en tâches plus restreintes

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

⇒ Découper en tâches plus restreintes

⇒ Se partager les tâches

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

⇒ Découper en tâches plus restreintes

⇒ Se partager les tâches

⇒ Utiliser un IDE commun de préférence (et un linter)

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

- ⇒ Découper en tâches plus restreintes
- ⇒ Se partager les tâches
- ⇒ Utiliser un IDE commun de préférence (et un linter)
- ⇒ Revoir le code existant et le nettoyer

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

- ⇒ Découper en tâches plus restreintes
- ⇒ Se partager les tâches
- ⇒ Utiliser un IDE commun de préférence (et un linter)
- ⇒ Revoir le code existant et le nettoyer
  - ⇒ Ajouter des commentaires

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

- ⇒ Découper en tâches plus restreintes
- ⇒ Se partager les tâches
- ⇒ Utiliser un IDE commun de préférence (et un linter)
- ⇒ Revoir le code existant et le nettoyer
  - ⇒ Ajouter des commentaires
  - ⇒ Revoir le nom des variables

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

- ⇒ Découper en tâches plus restreintes
- ⇒ Se partager les tâches
- ⇒ Utiliser un IDE commun de préférence (et un linter)
- ⇒ Revoir le code existant et le nettoyer
  - ⇒ Ajouter des commentaires
  - ⇒ Revoir le nom des variables
  - ⇒ Respecter un styleguide

Objectif de votre projet : Hex simplifié

⚠ Tâche complexe !!!

- ⇒ Découper en tâches plus restreintes
- ⇒ Se partager les tâches
- ⇒ Utiliser un IDE commun de préférence (et un linter)
- ⇒ Revoir le code existant et le nettoyer
  - ⇒ Ajouter des commentaires
  - ⇒ Revoir le nom des variables
  - ⇒ Respecter un styleguide
- ⇒ Le tester intensivement !

Diviser son code entre plusieurs fichiers

- Pourquoi ?

Diviser son code entre plusieurs fichiers

- Pourquoi ?
  - ⇒ Éviter le fichier unique de 15k lignes de code

Diviser son code entre plusieurs fichiers

- Pourquoi ?
  - ⇒ Éviter le fichier unique de 15k lignes de code
  - ⇒ Faciliter le travail à plusieurs sur un projet

Diviser son code entre plusieurs fichiers

- Pourquoi ?
  - ⇒ Éviter le fichier unique de 15k lignes de code
  - ⇒ Faciliter le travail à plusieurs sur un projet
  - ⇒ Faciliter la réutilisation du code

Diviser son code entre plusieurs fichiers

- Pourquoi ?
  - ⇒ Éviter le fichier unique de 15k lignes de code
  - ⇒ Faciliter le travail à plusieurs sur un projet
  - ⇒ Faciliter la réutilisation du code
  - ⇒ Faciliter la maintenance

Diviser son code entre plusieurs fichiers

- Pourquoi ?
  - ⇒ Éviter le fichier unique de 15k lignes de code
  - ⇒ Faciliter le travail à plusieurs sur un projet
  - ⇒ Faciliter la réutilisation du code
  - ⇒ Faciliter la maintenance

Diviser son code entre plusieurs fichiers

- Comment ?

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition
    - 1 fichier central `main.c` qui va gérer le programme

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition
  - 1 fichier central `main.c` qui va gérer le programme
  - 1 fichier `hex.c` qui va contenir les fonctions de gestion du jeu

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition
  - 1 fichier central `main.c` qui va gérer le programme
  - 1 fichier `hex.c` qui va contenir les fonctions de gestion du jeu
  - 1 fichier `network.c` qui va contenir les fonctions réseau

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition
  - 1 fichier central `main.c` qui va gérer le programme
  - 1 fichier `hex.c` qui va contenir les fonctions de gestion du jeu
  - 1 fichier `network.c` qui va contenir les fonctions réseau
  - 1 fichier `ai.c` qui va contenir les IA permettant de jouer automatiquement

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition
  - 1 fichier central `main.c` qui va gérer le programme
  - 1 fichier `hex.c` qui va contenir les fonctions de gestion du jeu
  - 1 fichier `network.c` qui va contenir les fonctions réseau
  - 1 fichier `ai.c` qui va contenir les IA permettant de jouer automatiquement
  - 1 fichier `interface.c` qui va contenir les fonctions gérant l'affichage et les entrées de l'utilisateur

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition
  - 1 fichier central `main.c` qui va gérer le programme
  - 1 fichier `hex.c` qui va contenir les fonctions de gestion du jeu
  - 1 fichier `network.c` qui va contenir les fonctions réseau
  - 1 fichier `ai.c` qui va contenir les IA permettant de jouer automatiquement
  - 1 fichier `interface.c` qui va contenir les fonctions gérant l'affichage et les entrées de l'utilisateur
  - 1 fichier `const.h` qui va contenir les constantes utilisées par tous les autres fichiers (ex : taille de la grille)

Diviser son code entre plusieurs fichiers

- Comment ?
- Un exemple de répartition
  - 1 fichier central `main.c` qui va gérer le programme
  - 1 fichier `hex.c` qui va contenir les fonctions de gestion du jeu
  - 1 fichier `network.c` qui va contenir les fonctions réseau
  - 1 fichier `ai.c` qui va contenir les IA permettant de jouer automatiquement
  - 1 fichier `interface.c` qui va contenir les fonctions gérant l'affichage et les entrées de l'utilisateur
  - 1 fichier `const.h` qui va contenir les constantes utilisées par tous les autres fichiers (ex : taille de la grille)
  - 1 fichier `header` pour chaque fichier C

Diviser son code entre plusieurs fichiers

- Fichier header ?

Diviser son code entre plusieurs fichiers

- Fichier header ?

  ⇒ Fichier qui contient les **prototypes** des fonctions

```c
int printBoard(int grid[NBR_LINES][NBR_CELLS]);
```

Diviser son code entre plusieurs fichiers

- Fichier header ?
  - ⇒ Fichier qui contient les **prototypes** des fonctions

```c
int printBoard(int grid[NBR_LINES][NBR_CELLS]);
```

    - Permet d'indiquer au compilateur les noms de fonctions, leurs paramètres et leur retour afin de pouvoir les appeler depuis un autre fichier.

Diviser son code entre plusieurs fichiers

- Fichier header ?

  ⇒ Fichier qui contient les **prototypes** des fonctions

```c
int printBoard(int grid[NBR_LINES][NBR_CELLS]);
```

  - Permet d'indiquer au compilateur les noms de fonctions, leurs paramètres et leur retour afin de pouvoir les appeler depuis un autre fichier.

  ⇒ On importe le fichier **header** avec la directive pré-processeur `include`

```c
#include "interface.h"
```

Diviser son code entre plusieurs fichiers

- Fichier header ?
  - ⇒ Fichier qui contient les **prototypes** des fonctions

```c
int printBoard(int grid[NBR_LINES][NBR_CELLS]);
```

    - Permet d'indiquer au compilateur les noms de fonctions, leurs paramètres et leur retour afin de pouvoir les appeler depuis un autre fichier.
  - ⇒ On importe le fichier **header** avec la directive pré-processeur `include`

```c
#include "interface.h"
```

  - ⇒ On peut également utiliser les fichiers **header** pour fixer des constantes utilisées par tous les fichiers

Utiliser un IDE

- IDE = Integrated Development Environment

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?

  ⇒ Coloration syntaxique

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion
  - ⇒ Débug, compilation, exécution, ...

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion
  - ⇒ Débug, compilation, exécution, ...
- Outil incontournable de développement

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion
  - ⇒ Débug, compilation, exécution, ...
- Outil incontournable de développement
- Lequel choisir ?

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion
  - ⇒ Débug, compilation, exécution, ...
- Outil incontournable de développement
- Lequel choisir ?
  - ⇒ Atom (et ses plug-ins, notamment `linter-gcc2`)

Utiliser un IDE

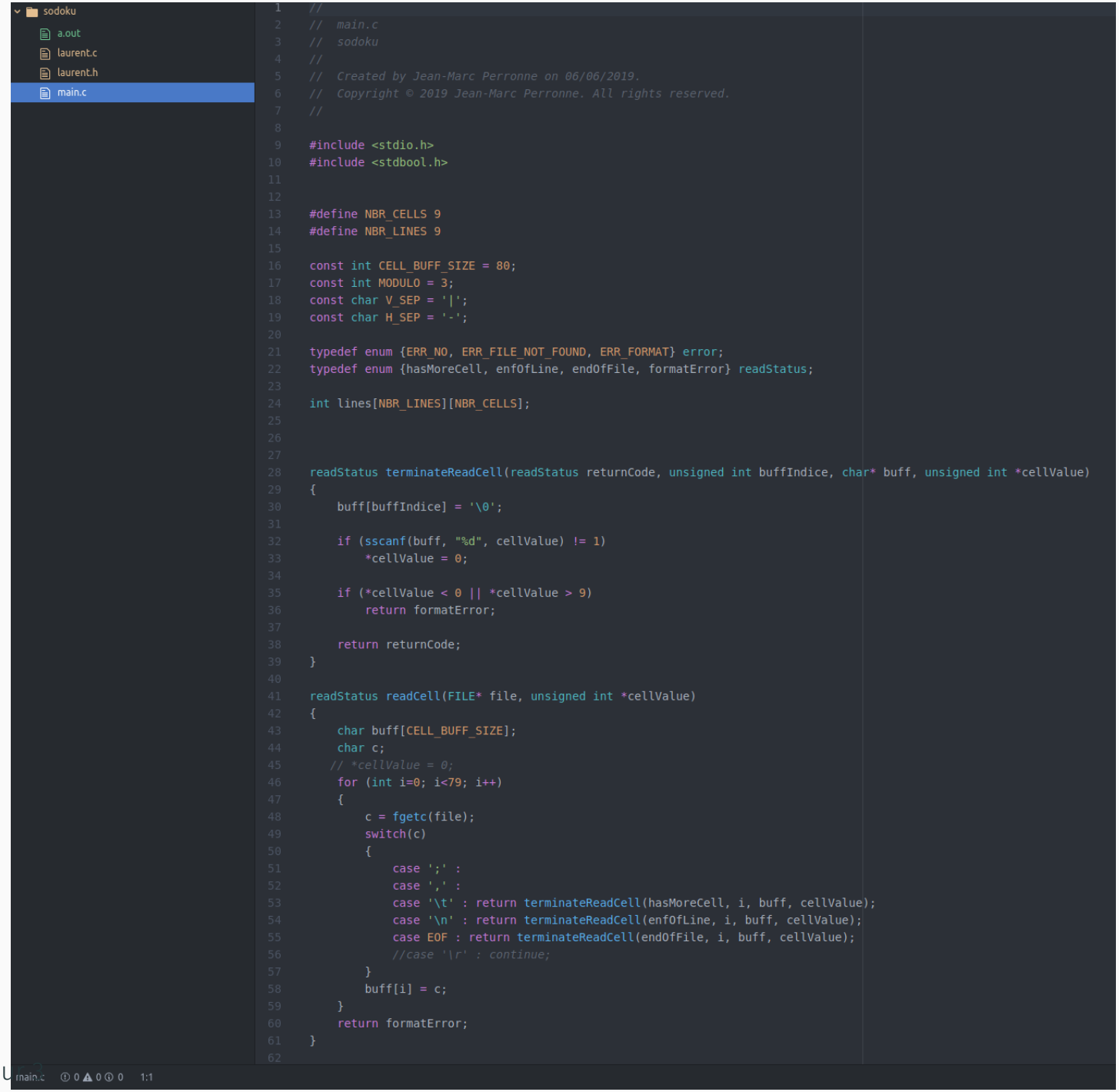
- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion
  - ⇒ Débug, compilation, exécution, ...
- Outil incontournable de développement
- Lequel choisir ?
  - ⇒ Atom (et ses plug-ins, notamment `linter-gcc2`)
  - ⇒ Visual Studio Code

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion
  - ⇒ Débug, compilation, exécution, ...
- Outil incontournable de développement
- Lequel choisir ?
  - ⇒ Atom (et ses plug-ins, notamment `linter-gcc2`)
  - ⇒ Visual Studio Code
  - ⇒ CLion

Utiliser un IDE

- IDE = Integrated Development Environment
- Intérêt ?
  - ⇒ Coloration syntaxique
  - ⇒ Analyse statique (linter)
  - ⇒ Auto-complétion
  - ⇒ Débug, compilation, exécution, ...
- Outil incontournable de développement
- Lequel choisir ?
  - ⇒ Atom (et ses plug-ins, notamment `linter-gcc2`)
  - ⇒ Visual Studio Code
  - ⇒ CLion
  - ⇒ ...


sodoku
a.out
laurent.c
laurent.h
main.c
//
// main.c
// sodoku
//
// Created by Jean-Marc Perronne on 06/06/2019.
// Copyright © 2019 Jean-Marc Perronne. All rights reserved.
//
#include <stdio.h>
#include <stdbool.h>
#define NBR_CELLS 9
#define NBR_LINES 9
const int CELL_BUFF_SIZE = 80;
const int MODULO = 3;
const char V_SEP = '|';
const char H_SEP = '-';
typedef enum {ERR_NO, ERR_FILE_NOT_FOUND, ERR_FORMAT} error;
typedef enum {hasMoreCell, enfOfLine, endOfFile, formatError} readStatus;
int lines[NBR_LINES][NBR_CELLS];
readStatus terminateReadCell(readStatus returnCode, unsigned int buffIndice, char* buff, unsigned int *cellValue)
{
buff[buffIndice] = '\0';
if (sscanf(buff, "%d", cellValue) != 1)
*cellValue = 0;
if (*cellValue < 0 || *cellValue > 9)
return formatError;
return returnCode;
}
readStatus readCell(FILE* file, unsigned int *cellValue)
{
char buff[CELL_BUFF_SIZE];
char c;
// *cellValue = 0;
for (int i=0; i<79; i++)
{
c = fgetc(file);
switch(c)
{
case ';' :
case ',' :
case '\t' : return terminateReadCell(hasMoreCell, i, buff, cellValue);
case '\n' : return terminateReadCell(enfOfLine, i, buff, cellValue);
case EOF : return terminateReadCell(endOfFile, i, buff, cellValue);
//case '\r' : continue;
}
buff[i] = c;
}
return formatError;
}
main.c
0 0 0
1:1

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code

  $\Rightarrow$ Maintenance du code

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code
  - ⇒ Maintenance du code
  - ⇒ Reprenabilité du code

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code
  - ⇒ Maintenance du code
  - ⇒ Reprenabilité du code
- Syntaxe ?

```
// Oh le beau commentaire sur une ligne
/* Oh le beau commentaire
sur plusieurs lignes */
```

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code
  - ⇒ Maintenance du code
  - ⇒ Reprenabilité du code
- Syntaxe ?
  - ⇒ Commentaire sur une ligne avec //

```
// Oh le beau commentaire sur une ligne
/* Oh le beau commentaire
sur plusieurs lignes */
```

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code
  - ⇒ Maintenance du code
  - ⇒ Reprenabilité du code
- Syntaxe ?
  - ⇒ Commentaire sur une ligne avec `//`
  - ⇒ Commentaire sur plusieurs lignes avec `/*  */`

```
// Oh le beau commentaire sur une ligne
/* Oh le beau commentaire
sur plusieurs lignes */
```

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code
  - ⇒ Maintenance du code
  - ⇒ Reprenabilité du code
- Syntaxe ?
  - ⇒ Commentaire sur une ligne avec `//`
  - ⇒ Commentaire sur plusieurs lignes avec `/*  */`

```
// Oh le beau commentaire sur une ligne
/* Oh le beau commentaire
sur plusieurs lignes */
```

- Comment ?

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code
  - ⇒ Maintenance du code
  - ⇒ Reprenabilité du code
- Syntaxe ?
  - ⇒ Commentaire sur une ligne avec `//`
  - ⇒ Commentaire sur plusieurs lignes avec `/*  */`

```
// Oh le beau commentaire sur une ligne
/* Oh le beau commentaire
sur plusieurs lignes */
```

- Comment ?
  - ⇒ En mettre suffisamment pour que le code soit compris

Bonnes pratiques : commentaire

- Objectif : Améliorer la compréhension du code
  - ⇒ Maintenance du code
  - ⇒ Reprenabilité du code
- Syntaxe ?
  - ⇒ Commentaire sur une ligne avec `//`
  - ⇒ Commentaire sur plusieurs lignes avec `/*  */`

```
// Oh le beau commentaire sur une ligne
/* Oh le beau commentaire
sur plusieurs lignes */
```

- Comment ?
  - ⇒ En mettre suffisamment pour que le code soit compris
  - ⇒ Ne pas mettre de commentaire évident comme ci-dessus :

```
i=i+1;//Ajoute 1 a i
```

Bonnes pratiques : Nommage

- Donner des noms de variable explicites

```
int toto;     // non !
int var27;    // non !
int CPT_MM;   // non !

int compteur; // oui !
```

Bonnes pratiques : Nommage

- Donner des noms de variable explicites

```
int toto;     // non !
int var27;    // non !
int CPT_MM;   // non !

int compteur; // oui !
```

- Donner des noms de fonction explicites et lisibles

```
void process()              // non !
void TRAITEMENT_12()        // non !
void resolutiondelagrille() // non !

void resolutionDeLaGrille() // oui !
```

Bonnes pratiques : Indentation

- Indenter correctement :

```c
#include <stdio.h>
   main(){printf("Vous␣êtes␣12␣élèves␣;␣");printf("chacun
doit␣10␣euros\n");printf("Je␣gagne␣120␣euros␣!\n");}
```

```c
#include <stdio.h>

int main(){
   printf("Vous␣êtes␣12␣élèves␣;␣");
   printf("chacun␣doit␣10␣euros\n");
   printf("Je␣gagne␣120␣euros␣!\n");
}
```

Bonnes pratiques : Styleguide

- Harmoniser les pratiques

Bonnes pratiques : Styleguide

- Harmoniser les pratiques

⇒ Homogénéiser le code et améliorer sa lisibilité

Bonnes pratiques : Styleguide

- Harmoniser les pratiques

⇒ Homogénéiser le code et améliorer sa lisibilité

- Quelques coding style en C :
  - Projet GNOME C coding style
  - University of Maryland C coding style
  - CS 50 C coding style

Bonnes pratiques : Styleguide

- Harmoniser les pratiques

⇒ Homogénéiser le code et améliorer sa lisibilité

- Quelques coding style en C :
  - Projet GNOME C coding style
  - University of Maryland C coding style
  - CS 50 C coding style
- Choisissez en un de la liste ou un autre, voire créez en un si vous le souhaitez et respectez le.