Comprendre le découpage du code avec le TP du snake

Prenons l'exemple du TP `snake`.

Deux fichiers vous ont été fournis pour vous aider à gérer le terminal et la lecture des clés au clavier `keyboard-event.h` et `keyboard-event.cpp` Les extensions de ces fichiers (après le `.`) sont différentes: vous avez un `.h` et un `.cpp`.

Affichons le contenu du fichier `keyboard-event.h`

```
void backgroundClear();
int keyEvent();
void sleepOneLap(const int);
```

Vous n'avez là que des déclarations, ici les déclarations de trois fonctions (déclaration de fonction = fonction où le corps a été remplacé par `;`).

Si maintenant vous regardez le fichier `keyboard-event.cpp` vous trouvez (naturellement) la définition de ces trois fonctions. Vous y trouvez aussi deux variables `STDIN` et `initialized` qui ne servent que localement et sont donc déclarées locales (`static`) à ce fichier (en effet il serait inutile que les autres fichiers puissent y accéder voire nocif, si l'un de ces fichiers décidait de changer leur valeur en cours de route...).

Fichier d'entête versus fichiers d'implémentation

Dans les fichiers `.h` ou fichiers de *header* vous mettez ce que vous voulez montrer *à l'extérieur* et dans le fichier d'implémentation `cpp` vous *cachez* votre implémentation.

Le fichier `.h` doit être inclus dans les fichiers où vous voulez utiliser ces fonctions afin que `C++` puisse vérifier lors de la compilation, que les appels des fonctions sont cohérents (bien typés). Vous faites donc un `#include "keyboard-event.h"` dans `main_snake.cpp` où vous voulez utiliser les fonctions `backgroundClear`, `keyEvent` et `sleepOneLap`.

De la même manière, pour accéder aux fonctionnalités de la bibliothèque `C++` standard, par exemple aux `vector` vous devez inclure le fichier d'entête correspondant `#include <vector>`.

Et pour compléter le snake, vous avez le ou les fichiers que vous avez implémentés; pour l'instant disons un seul fichier `main_snake.cpp` qui contient toutes vos fonctions ainsi que la fonction `main`.

Compilation séparée

Lors de la compilation, seuls les fichiers d'implémentation sont compilés; les fichiers d'entête sont simplement inclus dans des fichiers d'implémentation ou d'autres fichiers `.h` suivant les besoins.

Pour créer l'exécutable du snake, il faut compiler et linker ensemble les deux fichiers d'implémentation `keyboard-event.cpp` et `main_snake.cpp`.

Nous pouvons créer l'exécutable en une seule fois

```
$ g++ main_snake.cpp keyboard-event.cpp -o snake
$ ./snake
--------|
|@......|
|.......|
|..oooO.|
|.......|
|.......|
---------
```

Mais, à chaque fois que vous modifier le code du snake, vous recompilez tous les fichiers, même le fichier `keyboard-event.cpp` qui pourtant ne change pas. Ce qui est totalement inutile. Si ce fichier était gros et long à compiler ça pénaliserait fortement vos temps de compilation pour rien du tout.

Quand les codes sont implémentés sur plusieurs fichiers, il devient nécessaire d'utiliser la compilation séparée, donc autant le faire tout de suite.

Cela consiste à produire des fichiers *objets* (les fameux `.o`) à partir des fichiers d'implémentation puis de *linker* ces fichiers objets ensemble pour former votre exécutable (*linker* s'appelle aussi l'édition de liens).

```
$ g++ -c main_snake.cpp
$ g++ -c keyboard-event.cpp
$ g++ -o snake main_snake.o keyboard-event.o
$ ./snake
```

Faire tout cela à-la-main surtout quand vous avez plein de fichiers est tellement fastidieux que, y'a très longtemps (dans les années 1970) quelqu'un a eu l'idée d'écrire une commande pour gérer les chaînes de compilation.

la commande `make`

Une chaîne de compilation va être décomposée en un ensemble de règles dans le genre:
pour fabriquer le fichier `main_snake.o` il faut compiler le fichier `main_snake.cpp` en faisant la commande `g++ -c`
ou encore *pour fabriquer l'exécutable `snake` il faut linker les fichiers `main_snake.o` et `keyboard-event.o` en faisant la commande `g++ -o snake main_snake.o keyboard-event.o`*.

Mais alors là attention ! avant de pouvoir linker les fichiers `main_snake.o` et `keyboard-event.o` il faut être sûr d'avoir compilé les fichiers `main_snake.cpp` et `keyboard-event.cpp`, sinon la commande échoue !

C'est pour cela qu'ils ont ajouté aux règles, des dépendances qui doivent être satisfaites avant que la règle puisse être appliquée. Vous allez dire: que la règle de fabrication de l'exécutable `snake` va dépendre de l'existence des fichiers `keyboard-event.o` et `main_snake.o`. Donc si ils n'existent pas, la commande `make` va fabriquer les fichiers `keyboard-event.o` et `main_snake.o` en appelant l'autre règle.

Et si les fichiers `keyboard-event.o` et `main_snake.o` existent ? La commande `make` est intelligente: elle ne va recompiler que ceux qui ont été modifiés depuis la dernière fois où ils ont été compilés, en regardant les dates des fichiers c'est pas sorcier.

écrire un makefile

Ces règles et leurs dépendances sont écrites dans un fichier `Makefile` ou `makefile` sous cette forme:

```
cible: dépendance
    action
```

Attention avant `action` y'a une **tabulation**.

Puis avec la commande `make` vous allez appeler cette règle comme cela:

```
make cible
```

Attention, la syntaxe des `makefile` est basique, rudimentaire et cryptique.

Ouvrez un éditeur de texte pour éditer le fichier `Makefile` et on y va.

En premier, on peut définir la variable indiquant le compilateur qui est aussi le linker.

```
CPP=g++
```

Pour accéder à une variable on fera `$(CC)`

On peut définir la règle générale pour compiler un `.cpp` en un `.o`

```
%.o:%.cpp
    $(CPP) -o $@ -c $<
```

Attention avant `$(CPP)` y'a une tabulation. Et là y'a des trucs cryptico-rudimentaires:
`%` indique le préfixe du fichier genre `main_snake`
`$@` indique la cible (la partie gauche des `:`) de la règle ici `%.o`
`$<` indique la dépendance ici `%.cpp`

donc si on décode, quand on fait:

```
make main_snake.o
```

`make` fait:

```
g++ -o main_snake.o -c main_snake.cpp
```

Essayez !

Maintenant on peut définir la règle pour linker tous vos `.o`:

```
snake : main_snake.o keyboard-event.o
    $(CPP) $^ -o $@
```

`$^` signifie toutes les dépendances

Et enfin, que manque-t-il ? De rajouter les dépendances entre fichiers objets et fichiers d'implémentation et de header.

Ici `main_snake.o` dépend de `main_snake.cpp` mais aussi de `keyboard-event.h` puisqu'ils l'inclut, et `keyboard-event.o` dépend de `keyboard-event.cpp` et de `keyboard-event.h`.

Donc on va rajouter des dépendances sans actions :

```
keyboard-event.o : keyboard-event.cpp keyboard-event.h
main_snake.o : main_snake.cpp keyboard-event.h
```

Ainsi si quelqu'un touche au fichier `keyboard-event.h` les fichiers `keyboard-event.cpp` et `main_snake.cpp` seront re-compilés ce qui est logique.

Vous mettez toutes ces règles à la suite dans le fichier `makefile`, vous mettez en première règle celle qui sera la règle par défaut (celle faite si `make` tout court est exécuté), vous rajoutez la règle pour nettoyer et vous avez votre premier `makefile` tout simple mais déjà pas trop mal.

```
CPP=g++
RM=rm -f

snake : main_snake.o keyboard-event.o
    $(CPP) $^ -o $@

keyboard-event.o : keyboard-event.cpp keyboard-event.h
main_snake.o : main_snake.cpp keyboard-event.h

%.o:%.cpp
    $(CPP) -o $@ -c $<

snake : main_snake.o keyboard-event.o
    $(CPP) $^ -o $@

clean:
    $(RM) main_snake.o keyboard-event.o snake
```

```
.PHONY: clean
```

et `PHONY` ? Vous remarquez que toutes les cibles sauf `clean` indiquent des noms de fichiers à fabriquer. Pour éviter que `make` ne se mette à la confondre avec un éventuel fichier `clean` avec la commande `clean`, on l'indique par `PHONY`.

Essayez de rajouter un fichier `clean` dans votre répertoire et d'enlevez la règle `PHONY` et regardez ce qui se passe.

problème des tabulations

Il y un problème comme les commandes sont introduites par des **tabulation**, quand vous coupez/collez des règles, certains éditeurs (dont vs-code) transforment les tabulations en espaces (genre 8) et la commande `make` ne voyant pas de `tabulation` fera une erreur dans le genre

```
$ make snake
makefile:8: *** missing separator (did you mean TAB instead of 8 spaces?).
Stop.
```

voir là https://stackoverflow.com/questions/34937092/why-does-visual-studio-code-insert-spaces-when-editing-a-makefile-and-editor-in/56060185
et/ou là https://github.com/microsoft/vscode/issues/8227