TP C++

ESIREM 1A 2021-2022

Présentation générale

Le but de ces huit séances de TP va être de réaliser un projet de jeu Pacman simplifié afin de mettre en application les compétences acquises lors des CMs et des TDs. Le projet se réalise en groupes de deux étudiants et est suivi à l'aide du système de gestion de versions Git.

Les consignes doivent être scrupuleusement respectées afin d'éviter tout problème lors de la fusion des travaux des étudiants ou pour les modifications ultérieures.

Le travail rendu sera constitué du depôt Git (lien public ou archive zip), éventuellement complété d'un rapport sur les notions abordées et le recul sur l'expérience acquise (2 pages max). Le lien doit notamment être fait dans le rapport avec l'apport des notions abordées (principes SOLID, conception, gestion de versions).

En cas de difficultés avec Git, il est très fortement recommandé de réaliser une copie du dossier complet avant toute nouvelle manipulation, pour limiter le risque de pertes suite à une fausse manipulation.

Le sujet se découpe en deux parties, une par étudiant, le but étant de réaliser la fusion des différents codes sous Git.

Le développement se fait idéalement sous Qt Creator, pensez à vous mettre d'accord sur l'outil utilisé afin d'éviter des problèmes lors de la fusion dans Git. La procédure d'installation sur les ordinateurs personnels est détaillée à Annexe 4 :Installation de logiciels.

TP 1 : Prise en main

Conception (10 minutes)

Dans un premier temps, nous allons réaliser un travail de groupe afin de réfléchir à la conception UML d'un tel projet. La conception réellement utilisée sera fournie au fil des TPs. L'objectif ici est d'appliquer les notions de conception UML qui ont été vues durant les TDs pour pouvoir dégrossir le terrain et avoir une idée de la direction à prendre.

Initialisation du Git

Avant toute programmation, il est nécessaire de mettre en place le projet. Pour cela, vous allez créer un compte sur Github ou Gitlab. Ces sites permettent de conserver une copie de votre projet afin de travailler en équipe dessus.

Étudiant 1

Le premier étudiant va créer un nouveau projet sous Qt Creator (Fichier > Nouveau fichier ou projet). Dans le menu Projets, prendre Application (Qt) puis Qt Widgets Application. Mettre qmake en Build System. Donner un chemin pour le projet, préciser la plateforme de compilation.

QtCreator propose de suivre les modifications dans Git lors de la création du projet. Pour voir les différentes étapes, nous allons suivre la procédure dans le terminal.

- Garder <None> pour « Ajouter au gestionnaire de versions »
- Lancer Git Bash et se déplacer dans le répertoire du projet (commande cd).
- Lancer git init (affiche Initialized empty Git repository in...)
- Vérifier que le depôt soit bien créé en lançant git status.

Si la création du depôt n'a pas fonctionné, la commande devrait afficher un message d'erreur (fatal: not a git repository...). Sinon, la commande devrait afficher :

```
1   On branch master
2
3   No commits yet
4
5   Untracked files:
6     (use "git add <file>..." to include in what will be commited)
7   ...
8   ...
9
10  nothing added to commit but untracked files present (use "git add" to track)
```

Git status permet de s'informer sur l'état du dépôt local. Dans celui-ci il existe plusieurs types de fichiers possibles :

- Les fichiers ignorés par git (non affichés dans git status par défaut)

- Les fichiers non suivis (ici tout le projet)
- Les fichiers suivis

Les fichiers non suivis ou ignorés sont utiles pour avoir par exemple les fichiers de configuration qui sont propres à la machine. Les codes sources en revanche doivent être suivis par git pour être poussés sur le dépôt puis récupérés par l'autre étudiant.

Pour cela, on va tout d'abord configurer Git si ce n'est pas encore fait. Pour pouvoir savoir qui a réalisé quoi et comment le contacter, Git demande aux développeurs de fournir leur nom et leur adresse mail. Il est possible de configurer cela de manière locale (par projet) ou globale (pour l'ensemble des projets). La première solution est intéressante si vous avez plusieurs identités par exemple.

Gardez en tête que les informations fournies vont être synchronisées sur le dépôt Github/Gitlab et qu'il est assez difficile de modifier cela a posteriori. Je vous propose ici d'utiliser un pseudonyme et un faux mail pour ne pas avoir ses informations sur internet (sur de vrais projets, l'adresse mail reste importante pour contacter les développeurs).

```
11  git config --global user.name "Mon nom"
12  git config --global user.email "monadressemail@serveur.com"
```

L'option --global permet d'indiquer que ces informations sont communes à tous les nouveaux projets suivis sous Git. Cela évite d'avoir à ré-exécuter ces commandes lors de la création d'un nouveau projet. Si l'option est enlevée, le nom et l'adresse mail ne seront utilisés que pour les commits de ce projet.

Maintenant que Git est configuré, nous allons ajouter les fichiers de départ pour en garder une trace.

```
13  git add main.cpp mainwindow.* *.pro
```

On décide de suivre les codes sources ainsi que le fichier .pro qui permet la compilation. Les fichiers .pro.user sont spécifiques à l'ordinateur et il n'est donc pas utile de le suivre.

Si on refait git status, on voit que les fichiers ajoutés avec git add sont dans une nouvelle catégorie. Ils sont considérés comme « Changes to be commited / new file ». Ici on a deux informations sur les fichiers :

- Les fichiers suivis sont des nouveaux fichiers, ils n'ont pas été modifiés précédemment
- Les fichiers suivis sont marqués comme étant prêts à être enregistrés dans un commit

Si on exécute la commande git log, on remarque qu'il n'y a pas encore de commit réalisés. Dans git, il existe une zone intermédiaire dans laquelle on place les fichiers et les modifications que l'on prépare à enregistrer dans un commit.

Pour pouvoir réaliser le commit et donc avoir un instantané du projet, on exécute la commande git commit.

```
14  git commit -m "Mon premier commit"
```

L'option -m permet de donner le message du commit dans la commande (penser à prendre un nom qui a du sens), sinon un éditeur s'ouvre pour éditer le message du commit (utile notamment pour mettre plus d'informations).

Quelques informations s'affichent pour confirmer la création du commit, indiquer la branche sur laquelle le commit se place (master) et les fichiers enregistrés.

Si on refait git log, on remarque que le commit apparaît dans le résultat. On a également le nom de l'auteur, l'adresse mail, la date, le message, la branche sur laquelle le commit se place et un code alphanumérique qui représente l'identifiant du commit.

On souhaite maintenant envoyer ce dur labeur sur internet pour que le binôme puisse récupérer le code et commencer à travailler. Pour cela, on va créer un depôt git vide sur le site et récupérer l'url du dépôt (par exemple git@github.com:nomutilisateur/projet.git). Attention, il ne s'agit pas de l'url de la page.

On ajoute ensuite cette url dans le dépôt local pour pouvoir envoyer les modifications :

```
15  git remote add origin <mon url>
```

Cela va créer une sorte d'alias nommé origin qui va pouvoir être utilisée lors des envois (push) et des récupérations (fetch/pull). Cela permet d'avoir plusieurs depôts gits distants sans problème.

Si vous utilisez le lien en https, git demandera le nom d'utilisateur un token personnel pour pousser les modifications sur le dépôt. Si le lien utilisé est en ssh, alors il faut mettre en œuvre les clefs de chiffrement pour permettre à l'ordinateur d'accéder au dépôt. La première méthode est moins sécurisée mais un peu plus simple. Il faut aller sur GitHub, Settings, Developper Settings, Personnal Access Token et créer un nouveau token. Celui-ci est utilisé à la place du mot de passe au moment du push.

Pour utiliser ssh, on va dans un terminal et on exécute ssh-keygen, la commande va demander des informations (notamment l'endroit où enregistrer la clef et le mot de passe éventuel). Elle crée ensuite deux fichiers, le fichier clef et clef.pub. clef correspond à la clef privée et ne doit jamais être divulguée, clef.pub correspond à la clef publique, elle doit être copiée dans Github/Gitlab pour pouvoir autoriser les pushs. Dans Settings > SSH and GPG Keys > New SSH key. Lors du push il est nécessaire que la clef soit dans ~/.ssh et ait pour nom id_rsa (clef par défaut). Sinon il faut modifier le fichier de configuration de ssh pour utiliser la bonne clef ce qui sort un peu du cadre du TP. Vous pouvez regarder l'annexe Utilisation avancée de ssh pour la curiosité.

On peut vérifier que l'alias soit créé en exécutant git remote -v.

Enfin, on va envoyer le projet sur internet en utilisant la commande git push.

```
16  git push -u origin master
```

L'option -u n'est nécessaire que la première fois qu'on pousse une branche sur un depôt, elle sert à marquer master comme étant une branche d'origin. Par la suite, il suffit d'utiliser git push origin pour synchroniser toutes les branches qui ont subi cette opération.

On peut vérifier avec git log que la branche master a bien été envoyée sur origin, dans ce cas, il existe une nouvelle « branche », nommée origin/master, qui représente l'état du depôt distant depuis la dernière synchronisation.

- Ajouter le deuxième étudiant comme contributeur du dépôt pour qu'il puisse accéder et modifier le code.

Étudiant 2

Le deuxième étudiant va de son côté récupérer le projet directement sur le dépôt distant.

Il est possible de faire le travail de l'étudiant 1 de son côté avec un dépôt inutile (pour voir les commandes), ou assister l'autre étudiant afin de s'assurer que les notions soient bien comprises.

Lorsque le dépôt est en ligne et le deuxième étudiant ajouté au dépôt, le deuxième étudiant récupère une copie de son côté avec la commande git clone, en se mettant dans un répertoire de travail :

```
17  git clone <mon adresse de dépôt>
```

Si vous utilisez le lien en https, git demandera le nom d'utilisateur un token personnel pour pousser les modifications sur le dépôt. Si le lien utilisé est en ssh, alors il faut mettre en œuvre les clefs de chiffrement pour permettre à l'ordinateur d'accéder au dépôt. La première méthode est moins sécurisée mais un peu plus simple. Il faut aller sur GitHub, Settings, Developper Settings, Personnal Access Token et créer un nouveau token. Celui-ci est utilisé à la place du mot de passe au moment du push.

Pour utiliser ssh, on va dans un terminal et on exécute ssh-keygen, la commande va demander des informations (notamment l'endroit où enregistrer la clef et le mot de passe éventuel). Elle crée ensuite deux fichiers, le fichier clef et clef.pub. clef correspond à la clef privée et ne doit jamais être divulguée, clef.pub correspond à la clef publique, elle doit être copiée dans Github/Gitlab pour pouvoir autoriser les pushs. Dans Settings > SSH and GPG Keys > New SSH key. Lors du push il est nécessaire que la clef soit dans ~/.ssh et ait pour nom id_rsa (clef par défaut). Sinon il faut modifier le fichier de configuration de ssh pour utiliser la bonne clef ce qui sort un peu du cadre du TP.

Un nouveau dossier est créé avec les fichiers envoyés précédemment. Celui-ci peut enfin être ouvert dans Qt Creator.

Afin d'envoyer nos modifications, on va tout d'abord configurer Git si ce n'est pas encore fait. Pour pouvoir savoir qui a réalisé quoi et comment le contacter, Git demande aux développeurs de fournir leur nom et leur adresse mail. Il est possible de configurer cela de

manière locale (par projet) ou globale (pour l'ensemble des projets). La première solution est intéressante si vous avez plusieurs identités par exemple.

Gardez en tête que les informations fournies vont être synchronisées sur le dépôt Github/Gitlab et qu'il est assez difficile de modifier cela a posteriori. Je vous propose ici d'utiliser un pseudonyme et un faux mail pour ne pas avoir ses informations sur internet (sur de vrais projets, l'adresse mail reste importante pour contacter les développeurs).

```
18  git config --global user.name "Mon nom"
19  git config --global user.email "monadressemail@serveur.com"
```

L'option --global permet d'indiquer que ces informations sont communes à tous les nouveaux projets suivis sous Git. Cela évite d'avoir à réexécuter ces commandes lors de la création d'un nouveau projet. Si l'option est enlevée, le nom et l'adresse mail ne seront utilisés que pour les commits de ce projet.

Premières fonctions

Dans un premier temps, on va mettre de côté les fichiers mainwindow et supprimer le code présent dans le main afin d'avoir un programme console.

On enregistre les modifications dans un commit :

```
20  git add -u
21  git commit -m "Retrait interface graphique"
```

Étudiant 1

Le premier étudiant va réaliser une fonction nommée deplacer_personnage, qui prend deux entiers en lecture/écriture, ainsi qu'une chaîne de caractères représentant la commande à réaliser. Cette fonction retourne un booléen si le déplacement a eu lieu.

Pour pouvoir utiliser la fonction dans le main, il est nécessaire de la placer avant le main ou de la déclarer précédemment (de dire qu'elle existe, qu'elle prend tels paramètres et retourne tel type) :

```
22  bool deplacer_personnage(int x, int y, std::string cmd);
23
24  int main(int argc, char** argv)
25  {
26    int x=5, y=4;
27    deplacer_personnage(x, y, "UP");
28    return 0;
29  }
30
31  bool deplacer_personnage(int x, int y, std::string cmd)
32  {
33    // contenu
34  }
```

- Corriger la petite erreur présente dans la signature de la fonction. En cas de besoin, lire la suite

Lorsqu'une fonction est appelée, le comportement par défaut du C++ consiste à copier la valeur passée en paramètre dans une nouvelle variable. Celle-ci peut ensuite être modifiée et disparaît à la fin de la fonction. Cela pose deux problèmes : tout d'abord, les modifications ne sont jamais répercutées sur les variables de la fonction appelante (le x et y du main ne sont pas modifiées mais seulement les copies x et y appartenant à deplacer_personnage). L'autre problème se produit lorsqu'on passe des variables de grande taille comme des objets. Ceux-ci sont alors intégralement copiés ce qui peut être coûteux en terme de temps d'exécution. Pour éviter cela, on peut utiliser des pointeurs (int* x_addr = &x;), ou alors une version simplifiée proposée par le C++ appelé référence. Cette variable est une sorte de pointeur obligatoirement initialisée à sa déclaration par une autre variable et qui pointe sans arrêt sur cette case mémoire. C'est donc idéal pour les fonctions lorsqu'on souhaite référencer la variable dans la fonction appelante. Pour l'utiliser on remplace int x par int& x dans la signature de la fonction.

La fonction deplacer_personnage prend une commande qui vaut UP/DOWN/LEFT/RIGHT. Selon le cas, elle ajoute ou retire 1 à x ou y. La fonction ne peut pas déplacer le personnage si la commande n'est pas reconnue ou que l'on dépasse les limites possibles.

Pour pouvoir réaliser des tests, on va utiliser le bloc de commandes if/else :

```
35  if (condition)
36  {
37    // action à réaliser si condition est vraie
38  }
39  else if (condition2)
40  {
41    // action à réaliser si condition est fausse et condition2 est vrai
42  }
43  else
44  {
45    // action à réaliser si condition et condition2 sont faux
46  }
```

La fonction deplacer_personnage est donc une suite de tests if/else pour regarder si cmd correspond à une valeur attendue.

Pour simplifier la suite, on considère que l'origine des coordonnées est situé en haut à gauche de l'écran. C'est le système utilisé par la suite dans l'interface graphique.

- Écrire la fonction pour ajouter 1 à x si la commande vaut RIGHT, 1 à y si elle vaut DOWN, retirer 1 à x si la commande vaut LEFT et retirer 1 à y si elle vaut UP et ne rien faire si elle vaut IDLE. Retourner true si la variable a été modifiée, false sinon.

Rajoutons maintenant des limites autorisées, le personnage se déplace sur un plateau délimité, on a donc des valeurs xmin, xmax, ymin et ymax définies par le plateau. Pour pouvoir rendre le code plus lisible, on place ces valeurs dans des constantes. On peut soit utiliser un define, soit utiliser une variable constante.

```
47  // début de fichier
48  #define xmin 0
49  #define xmax 10
50  ...
```

Ou alors :

```
51  bool deplacer_personnage(...)
52  {
53    const int xmin=0;
54    const int xmax=10;
55    // suite
56  }
```

Les defines correspondent à une vieille façon de réaliser des constantes, toute occurrence de xmin est remplacée par 0 dans le code avant la compilation. À l'inverse, utiliser des variables constantes permet de n'avoir ces variables que dans le corps de la fonction et de vérifier le type, ce qui est un peu plus propre et facilite le debuggage.

- Mettre en place les constantes, on prendra xmin = 0, ymin = 0, xmax = 32 et ymax = 15.

Maintenant, pour chaque commande possible, on va tester si l'ajout ou le retrait de 1 fait sortir des bornes. Si ce n'est pas le cas on peut incrémenter x/y et retourner vrai, sinon il y a un problème et on retourne false.

- Ajouter des tests pour vérifier si les modifications sont possibles. Retourner false si elles sont impossibles.

La fonction est maintenant terminée, on souhaite la tester dans le main. Pour cela, l'idéal serait de pouvoir lire la commande saisie par l'utilisateur et lui afficher le résultat en retour. Un autre point intéressant serait de pouvoir répéter la saisie / affichage aussi souvent que nécessaire.

Pour afficher les messages dans la console, il est nécessaire de rajouter la ligne suivante dans le fichier .pro :

```
57  CONFIG += console
```

Pour lire une valeur saisie dans la console, il est nécessaire d'inclure le fichier iostream. Celui-ci fournit plusieurs fonctions et classes dont notamment std::cout et std::cin qui permettent la manipulation du terminal.

```
58  #include <iostream>
59  #include <string>
60
61  using namespace std;
62
63  int main(int argc, char** argv)
64  {
65    string saisie;
66    std::getline(cin, saisie);
67    cout << "Vous avez saisi " << saisie << endl;
68  }
```

La ligne using namespace std permet de « sortir » les objets de l'espace de nom std. Sans cette ligne, il est nécessaire de préfixer chaque objet de std avec std:: (par exemple std::string, std::cout, std::cin et std::endl).

Cin fonctionne avec l'opérateur >>, cependant pour une chaîne de caractères cela ne récupère qu'un seul mot et non une ligne entière. Pour pallier à ce comportement, on utilise généralement std::getline. L'opérateur >> peut être utilisé pour les entiers par exemple.

- Lire la saisie de l'utilisateur et l'utiliser dans déplacer, afficher les variables x et y après modification.

Jusque là tout va bien, cependant le code ne permet de modifier qu'une seule fois la position du personnage. On souhaite pouvoir effectuer cette modification autant de fois que nécessaire. Pour cela on va utiliser une boucle while :

```
69  while (condition)
70  {
71    // action à répéter tant que condition vaut vrai
72  }
73
74  do
75  {
76    // action à répéter
77  } while (condition); // si la condition est fausse, on sort
78
79  for (int i=0 ; i<10 ; i++)
80  {
81    // action à répéter 10 fois avec i qui vaut 0, 1, 2, 3..., 8, 9
82  }
```

Les trois structures proposées permettent toutes de répéter une action. La boucle while peut ne pas être exécutée si la condition est fausse au premier tour. La boucle do...while est forcément exécutée au moins une fois. La boucle for quant à elle, est généralement utilisée lorsque le nombre de répétitions est connu (par exemple pour parcourir un tableau).

- Utiliser une boucle while (true) pour demander indéfiniment la saisie utilisateur et déplacer le personnage. Bien mettre la déclaration et l'initialisation de x et y avant la boucle while.

On va maintenant ajouter les modifications dans le git pour préparer un commit. Tout d'abord, on va regarder l'état du dépôt avec la commande git status, puis regarder les modifications faites depuis le dernier commit avec git diff.

- Exécuter git status puis git diff

Pour ajouter les modifications, l'option -u de git add permet d'ajouter toutes les modifications des fichiers déjà suivis, c'est donc idéal dans le cas présent pour ne pas retaper tous les noms de fichiers.

- lancer un git add -u puis un git status

On remarque que les fichiers sont maintenant affichés en vert dans la section prêt à être commités. Cela veut dire que ces modifications seront ajoutées dans un commit si on exécute git commit (ne pas le faire maintenant).

On souhaite faire une dernière modification au code. En effet, la fonction retourne une valeur booléenne pour indiquer si le déplacement a été possible ou non. On souhaiterais pouvoir gérer le cas où le déplacement a été impossible de la façon la plus élégante possible. Pour cela, on va mettre en place des exceptions.

- Déclarer deux classes ExceptionBounds et ExceptionCommand vides avant le main.

Les exceptions sont des objets qui court-circuitent le bon fonctionnement du programme. Lorsqu'une exception est lancée, la fonction s'arrête et l'exception est envoyée à la fonction appelante. Si celle-ci est gérée le programme continue, sinon l'exception remonte la pile d'appel jusqu'à atteindre le main. Si l'exception n'est jamais gérée, le programme s'arrête. Le code utilisé pour les exceptions est le suivant :

```
83  void foo()
84  {
85    throw ExceptionBounds();
86  }
87
88  int main(int argc, char** argv)
89  {
90    try
91    {
92      foo();
93    }
94    catch (ExceptionBounds& eb)
95    {
96      // comportement si ExceptionBounds a été lancé
97    }
98    catch (ExceptionCommand& ec)
99    {
100     // comportement si ExceptionCommand a été lancé
101   }
102   catch (...)
103   {
104     // comportement si une autre exception est lancée
105   }
106   finally
107   {
108     // code systématiquement exécuté (permet de libérer la mémoire,
109     // fermer les fichiers...)
110   }
111   return 0;
112 }
```

Le code est un peu complexe mais nous n'utiliserons qu'une petite partie de celui-ci dans la fonction. Les trois commandes à retenir sont try, catch et throw. Le throw lance une exception, il est exécuté généralement s'il y a une erreur qui se produit. Le try indique que l'on exécute du code qui peut lancer des exceptions (il sert à les attraper). Le catch indique quoi faire si une exception d'un type donné est attrapé lors de l'exécution. Il est possible d'enchaîner les catch pour avoir un comportement différents pour chaque exception. Il est également possible d'ajouter des données comme un message d'erreur à la classe exception, afin d'avoir plus d'informations sur le problème (par exemple la commande qui a été tentée). Enfin, il est aussi possible de relancer une exception dans un catch (si l'exception ne peut pas être entièrement gérée à ce niveau). Bref, c'est un système extrêmement flexible et efficace pour gérer les erreurs.

- Lancer une exception ExceptionCommand si la commande n'est pas reconnue, afficher un message dans la console si l'exception est levée.
- Lancer une exception ExceptionBound si le déplacement se fait hors limites, afficher un message dans la console si l'exception est levée.

Bien entendu, vous vous demandez sans doute pourquoi ne pas tout simplement afficher ces messages dans la fonction deplacer_personnage. Ici notre but final est d'utiliser la fonction deplacer_personnage dans une interface graphique, afficher des messages dans la console aura alors moins de sens et il serait nécessaire de modifier le code de la fonction deplacer_personnage. Le fait de passer par une exception permet de ne pas avoir à déterminer

à priori de quelle façon on doit gérer l'erreur, et donc de pouvoir s'adapter à chaque situation efficacement (afficher un message console, une boite de dialogue, redemander une valeur, écrire dans un fichier log...).

Les exceptions peuvent être vues comme une erreur qui se produit dans une entreprise et la pile d'appel comme la hiérarchie. Si un employé tombe sur une situation à problèmes qu'il ne peut pas gérer, il demande à son supérieur. Si celui-ci peut gérer il le fait sinon il remonte encore le problème et ainsi de suite, en croisant les doigts pour que le haut de la pyramide ait une solution...

On a maintenant une fonction de très bonne qualité que l'on souhaite absolument sauvegarder dans le git pour en faire profiter notre binôme. Pour cela, on va d'abord admirer le travail réalisé avec git diff.

- Lancer git status puis git diff

Le git status montre deux fois le fichier main, une fois en rouge non prêt à être commité, et une fois en vert prêt à être commité. Pour comprendre la situation il faut voir ces lignes non pas comme des fichiers mais comme des modifications faites au fichier depuis le dernier commit. La ligne verte correspond aux anciennes modifications que l'on a rajoutées avec git add -u, la ligne rouge correspond aux modifications qui ont été faites depuis.

Le git diff va par défaut uniquement montrer les modifications faites en rouge. Pour pouvoir afficher les modifications faites en vert, on rajoute l'option --cached. Si on souhaite l'ensemble des modifications, on va exécuter git diff HEAD.

- Lancer git diff --cached, git diff et git diff HEAD, comparer les sorties
- Ajouter les modifications (git add -u)
- Faire un commit (git commit -m "Fonction deplacer_personnage")
- Envoyer les modifications (git push)
- Naviguer dans les commits sur le site internet en attendant que l'autre étudiant ait terminé sa partie et envoyé son travail

Étudiant 2

Le deuxième étudiant va réaliser une fonction detecter_collision qui indique si un ennemi se retrouve sur la même case que le personnage. Cette fonction prend en paramètre deux tableaux d'entiers de même taille contenant les positions x et y des ennemis, un entier représentant le nombre d'ennemis et deux entiers représentant la position du personnage. Elle retourne un booléen indiquant s'il y a une collision ou non.

Pour pouvoir utiliser la fonction dans le main, il est nécessaire de la placer avant le main ou de la déclarer précédemment (de dire qu'elle existe, qu'elle prend tels paramètres et retourne tel type) :

```
113 bool detecter_collision(int ennemis_x[], int ennemis_y[], int nb_ennemis,
114                         int x, int y);
115
116 int main(int argc, char** argv)
117 {
118   int x=5, y=4;
119   int enn_x[4] = {1, 2, 3, 4}, enn_y[4] = {1, 2, 3, 4};
120   if (detecter_collision(enn_x, enn_y, 4, x, y))
121   {
122     std::cout << "Collision" << std::endl;
123   }
124   return 0;
125 }
126
127 bool detecter_collision(int ennemis_x[], int ennemis_y[], int nb_ennemis,
128                         int x, int y, std::string cmd)
129 {
130  // contenu
131 }
```

On va maintenant ajouter les modifications dans le git pour préparer un commit. Tout d'abord, on va regarder l'état du dépôt avec la commande git status, puis regarder les modifications faites depuis le dernier commit avec git diff.

- Exécuter git status puis git diff

Pour ajouter les modifications, l'option -u de git add permet d'ajouter toutes les modifications des fichiers déjà suivis, c'est donc idéal dans le cas présent pour ne pas retaper tous les noms de fichiers.

- lancer un git add -u puis un git status

On remarque que les fichiers sont maintenant affichés en vert dans la section prêt à être commités. Cela veut dire que ces modifications seront ajoutées dans un commit si on exécute git commit (ne pas le faire maintenant).

On souhaite maintenant lancer une exception lorsque nb_ennemis n'est pas valide (inférieur ou égal à 0). Les exceptions sont des objets qui court-circuitent le bon fonctionnement du programme. Lorsqu'une exception est lancée, la fonction s'arrête et l'exception est envoyée à la fonction appelante. Si celle-ci est gérée le programme continue, sinon l'exception remonte la pile d'appel jusqu'à atteindre le main. Si l'exception n'est jamais gérée, le programme s'arrête. Le code utilisé pour les exceptions est le suivant :

```
132 void foo()
133 {
134   throw ExceptionBounds();
135 }
136
137 int main(int argc, char** argv)
138 {
139   try
140   {
141     foo();
142   }
143   catch (ExceptionBounds& eb)
144   {
145     // comportement si ExceptionBounds a été lancé
146   }
147   catch (ExceptionCommand& ec)
148   {
149     // comportement si ExceptionCommand a été lancé
150   }
151   catch (...)
152   {
153     // comportement si une autre exception est lancée
154   }
155   finally
156   {
157     // code systématiquement exécuté (permet de libérer la mémoire,
158     // fermer les fichiers...)
159   }
160   return 0;
161 }
```

Le code est un peu complexe mais nous n'utiliserons qu'une petite partie de celui-ci dans la fonction. Les trois commandes à retenir sont try, catch et throw. Le throw lance une exception, il est exécuté généralement s'il y a une erreur qui se produit. Le try indique que l'on exécute du code qui peut lancer des exceptions (il sert à les attraper). Le catch indique quoi faire si une exception d'un type donné est attrapé lors de l'exécution. Il est possible d'enchaîner les catch pour avoir un comportement différents pour chaque exception. Il est également possible d'ajouter des données comme un message d'erreur à la classe exception, afin d'avoir plus d'informations sur le problème (par exemple la commande qui a été tentée). Enfin, il est aussi possible de relancer une exception dans un catch (si l'exception ne peut pas être entièrement gérée à ce niveau). Bref, c'est un système extrêmement flexible et efficace pour gérer les erreurs.

- Créer une classe vide ExceptionSizeTab
- Lancer une exception ExceptionSizeTab si le nombre d'éléments est inférieur ou égal à 0, afficher un message dans la console si l'exception est levée.

Bien entendu, vous vous demandez sans doute pourquoi ne pas tout simplement afficher ces messages dans la fonction detecter_collision. Ici notre but final est d'utiliser la fonction detecter_collision dans une interface graphique, afficher des messages dans la console aura alors moins de sens et il serait nécessaire de modifier le code de la fonction detecter_collision. Le fait de passer par une exception permet de ne pas avoir à déterminer à priori de quelle

façon on doit gérer l'erreur, et donc de pouvoir s'adapter à chaque situation efficacement (afficher un message console, une boite de dialogue, redemander une valeur, écrire dans un fichier log...).

Les exceptions peuvent être vues comme une erreur qui se produit dans une entreprise et la pile d'appel comme la hiérarchie. Si un employé tombe sur une situation à problèmes qu'il ne peut pas gérer, il demande à son supérieur. Si celui-ci peut gérer il le fait sinon il remonte encore le problème et ainsi de suite, en croisant les doigts pour que le haut de la pyramide ait une solution…

Pour tester les exceptions, on va utiliser le débugger. Cliquer dans la marge, à gauche des numéros de lignes. Un point rouge apparaît. C'est un point d'arrêt. En mode debug, le programme va se mettre en pause au niveau de ce point ce qui va permettre de dérouler le fonctionnement du programme ligne par ligne et de voir l'état interne du programme (variables actuellement visibles, pile d'appel etc.). Il existe de nombreuses fonctionnalités de débug très utiles pour localiser un problème. On va rester ici sur une utilisation basique.

- Mettre en place deux blocs try catch dans le main, un exécutant detecter_collision sans erreur et un produisant une exception
- Positionner un point d'arrêt lors du premier appel à détecter_collision
- Lancer le programme en mode debug (la flèche verte avec l'insecte).

L'interface passe en mode debug. Sur la droite, un volet affiche les variables avec leur état courant. En bas, on trouve la pile d'appel contenant a minima main. Si plusieurs fonctions se trouvent dans la pile d'appel, il est possible de naviguer dedans pour voir les variables visibles dans une fonction donnée. Dans le volet du bas, on trouve plusieurs boutons utiles au debug :

Le premier bouton permet de continuer l'exécution jusqu'au prochain point d'arrêt, le deuxième sert à stopper le debug (et l'application). Le troisième permet d'avancer d'une instruction. Le quatrième permet d'avancer d'une instruction en mode détaillé (il rentre dans les fonctions). Le cinquième permet de continuer jusqu'à la fin de la fonction courante.

- En exécutant le programme pas à pas, étudier le comportement du code lorsque tout va bien et lorsqu'une exception est lancée. Bien penser à entrer à l'intérieur de detecter_collision.
- Quelles lignes sont exécutées en cas de bon fonctionnement ? En cas d'erreur ?

On a maintenant une fonction de très bonne qualité que l'on souhaite absolument sauvegarder dans le git pour en faire profiter notre binôme. Pour cela, on va d'abord admirer le travail réalisé avec git diff.

- Lancer git status puis git diff

Le git status montre deux fois le fichier main, une fois en rouge non prêt à être commité, et une fois en vert prêt à être commité. Pour comprendre la situation il faut voir ces lignes non pas comme des fichiers mais comme des modifications faites au fichier depuis le dernier

commit. La ligne verte correspond aux anciennes modifications que l'on a rajoutées avec git add -u, la ligne rouge correspond aux modifications qui ont été faites depuis.

Le git diff va par défaut uniquement montrer les modifications faites en rouge. Pour pouvoir afficher les modifications faites en vert, on rajoute l'option --cached. Si on souhaite l'ensemble des modifications, on va exécuter git diff HEAD.

- Lancer git diff --cached, git diff et git diff HEAD, comparer les sorties
- Ajouter les modifications (git add -u)
- Faire un commit (git commit -m "Fonction detecter_collision")
- Envoyer les modifications (git push)
- Attendre que l'autre étudiant ait terminé sa partie et envoyé son travail

À deux

Normalement, l'étudiant le moins rapide ne peut pas envoyer ses propres modifications. En effet, vous avez tous les deux travaillé sur le même fichier et sur les mêmes lignes. Mettons que le premier commit indique que x = 0, que le premier étudiant modifie le 0 en 1 et le deuxième étudiant le 0 en 2, quelle modification Git doit-il garder ? Git n'a aucun moyen de savoir quelle modification doit être conservée et laisse donc la possibilité aux deux étudiants de résoudre à la main le conflit de fusion.

Pour pouvoir régler ce problème, on va devoir fusionner les deux modifications dans un seul commit. Pour cela, on utilise la commande merge.

Tout d'abord, on s'assure d'avoir la dernière version du dépôt distant en exécutant :

```
162 git fetch origin
163 git log --graph --all --oneline
```

La deuxième commande permet de représenter de façon plus claire ce qui se passe dans le projet : on a les modifications courantes qui ont divergées par rapport au dépôt distant. On ne peut donc pas envoyer directement les modifications.

Pour pouvoir utiliser cette commande de façon plus simple, il est possible de définir un alias :

```
164 git config --global alias.graph 'log --graph --all --oneline'
```

En faisant cela, on peut exécuter git graph pour avoir le même résultat.

Une fois qu'on a récupéré la dernière version du dépôt, on tente une fusion avec git merge :

```
165 git merge origin/master
```

L'idée est de créer un commit qui a deux parents, master et origin_master et qui regroupe les modifications des deux côtés. Si la fusion est possible automatiquement elle est réalisée et le nouveau commit créé. Sinon, git passe dans un état particulier de fusion en cours, il faut alors passer par la commande git mergetool :

```
166 git status
167 git mergetool --tool=meld
```

Meld est un logiciel assez pratique pour réaliser les fusions, cependant il en existe de nombreux autres. Le logiciel est présenté sous la forme de trois fichiers textes mis côte à côte et des lignes surlignées. Le volet du centre correspond à l'ancêtre commun (l'état du fichier avant que les branches ne divergent). Le volet de gauche correspond à la version locale (_LOCAL) et le volet de droite à la version distante (_REMOTE). Les zones de couleur correspondent aux différences entre les fichiers (par exemple une ligne verte correspond à un ajout). Le but du jeu est de modifier le fichier central pour qu'il intègre intelligemment les modifications des deux autres fichiers. Pour cela il est possible de cliquer sur les flèches qui « poussent » les modifications sur l'autre fichier, ou de modifier directement à la main. Les touches Ctrl et Shift permettent de modifier le comportement de ces flèches (Ctrl => ajout, Shift => Suppression).

- Mettre toutes les modifications du fichier de gauche dans le fichier central.
- Le diff devient un peu trop confus pour fusionner le fichier de droite avec la même méthode, copier le texte à droite correspondant aux exceptions / fonctions réalisées et le rajouter dans le fichier du centre
- Maintenant, modifier le main pour que le programme positionne le personnage, demande une commande à l'utilisateur, déplace le personnage et teste les collisions en continu.
- Vérifier que la fusion s'est bien déroulée en compilant et en testant le projet
- Ajouter les modifications (git add -u, git status pour vérifier que tout soit ok).
- Exécuter git commit -m "Fusion du TP1" pour enregistrer les modifs
- Exécuter git push pour envoyer les modifications sur le serveur.

Il est possible pour la suite de ne pas avoir à rajouter --tool=meld en configurant Git correctement :

```
168 git config --global merge.tool=meld
```

Le comportement de Git peut être très finement choisi à l'aide de la commande config. Les paramètres sont stockés dans un fichier texte (.git/config) ce qui permet de facilement le transférer d'un projet à l'autre au besoin.

TP2 : Les classes

Il serait tout à fait possible de réaliser l'ensemble du programme en n'utilisant que des fonctions. Cependant le C++ permet l'utilisation de classes qui vont permettre de découper le code en unités cohérentes et donc de le rendre plus lisible et plus facile à maintenir.

Dans ce TP, nous allons déplacer le code réalisé précédemment dans des classes afin de faciliter le développement ultérieur du code.

Travail préliminaire

En deux groupes, réaliser une conception possible du jeu Pacman. En admettant que des classes puissent vous être fournies par l'enseignant, quelles caractéristiques attendez-vous d'elles ? Comment doivent-elles être liées ?

Dans la suite du TP, cette conception ne sera pas suivie pour pouvoir mieux gérer le temps, cependant être capable de déterminer les grandes fonctionnalités à implémenter reste une compétence vitale dans le développement.

Étudiant 1

- Réaliser une classe Personnage possédant une position pos_x et pos_y, un déplacement courant sous la forme d'une std::string et une méthode new_pos() qui calcule la position suivante. Bien respecter le nom des variables pour la suite.
- Ajouter un setter pour le déplacement.

On souhaite gérer élégamment le calcul de la collision avec un ennemi ou une pastille. On pourrait mettre tout le code dans Personnage, cependant d'autres classes vont avoir un comportement similaire. Pour pouvoir réutiliser facilement ce comportement, nous allons implémenter un design pattern Observateur.

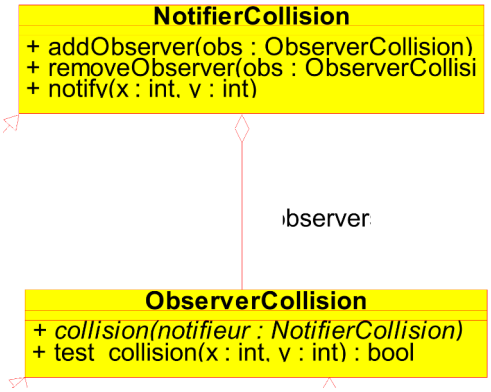
Les design patterns sont des façons de répondre à un problème de programmation de la façon la plus élégante et la plus fiable possible. Par exemple, si vous souhaitez réaliser une classe qui ne possède qu'une et une seule instance, il est possible de la construire en suivant le design pattern Singleton.

Le design pattern Observateur de son côté est utile lorsqu'une classe va « observer » la modification de l'état d'une autre classe. Une conception simple consisterait à vérifier régulièrement si la classe observée a été modifiée. Cependant cela va vite coûter beaucoup de ressources et la classe Observateur ne va pas réagir instantanément à un changement d'état.

Une façon de résoudre ce problème va consister à remplacer le fait que A observe B par le fait que B notifie A de ses modifications.

Cette manière d'écrire le code permet d'économiser les ressources et de réagir instantanément à un changement d'état de B. Cependant que se passe-t-il si plusieurs classes différentes doivent observer / notifier ?

Pour pouvoir résoudre ce problème, le comportement d'Observateur et de Notifieur est placé dans des classes dédiées, et les Observateurs et Notifieurs réels héritent de ces classes.



Voici une conception qui met en place le pattern Observateur dans le cadre du projet (le vrai pattern se trouve facilement sur internet). La classe Observateur « s'abonne » à un notifieur. Lorsque le notifieur est modifié, il appelle la fonction notify qui va appeler collision() sur chaque observateur en collision.

Pour n'appeler collision que sur les objets au contact, on rajoute une fonction test_collision qui permet de vérifier s'il y a une collision ou non.

La classe ObserverCollision sera réalisée par l'étudiant 2. Bien respecter les noms de fonctions pour éviter des problèmes lors de la fusion des travaux.

- Implémenter la classe NotifierCollision. notify() appelle test_collision sur chaque observateur et collision lorsque test_collision vaut vrai.
- Faire hériter Personnage de NotifierCollision.
- À chaque déplacement, appeler notify
- Envoyer le travail sur le dépôt Git, régler les éventuels conflits de fusion.
- Attendre que l'étudiant 2 ait fini l'interface Observer. Récupérer son travail via Git.

Si on récapitule, lorsque Personnage se déplace, il indique ce déplacement aux Ennemis qui vérifient s'ils sont sur la même position. En cas de collision, le joueur perd une vie (représenté pour le moment par un message console).

On souhaite maintenant réaliser le processus inverse : à chaque déplacement d'Ennemi, celui-ci doit également notifier Personnage pour que celui-ci teste une éventuelle collision.

- Faire hériter Personnage de ObserverCollision. Définir les méthodes.
- test_collision(x, y) vérifie si les paramètres correspondent à la position courante du Personnage
- collision() indique qu'il y a une collision (message console pour le moment).
- Envoyer le travail sur le dépôt Git, régler les éventuels conflits de fusion.
- Attendre que l'étudiant 2 ait fini la classe ObserverCollision, récupérer son travail via Git.
- Envoyer le travail sur le dépôt et récupérer le travail de l'autre étudiant une fois terminé.

Étudiant 2

- Réaliser une classe Ennemi possédant une position pos_x et pos_y, une méthode virtuelle pure new_pos() et une méthode deplacer(x, y) qui servira à forcer les valeurs pour les tests.
- Réaliser une classe Clyde héritant d'Ennemi, implémentant new_pos() qui calcule la position suivante. Le déplacement sera aléatoire pour le moment (voir snippet).

Pour calculer des valeurs pseudo-aléatoires en C++ :

```
169 #include <random>
170 #include <ctime>
171 srand(time(NULL));
172 int val = rand() % 10; // 0 à 9
```

On souhaite gérer élégamment le calcul de la collision avec un personnage. On pourrait mettre tout le code dans Ennemi, cependant d'autres classes vont avoir un comportement similaire. Pour pouvoir réutiliser facilement ce comportement, nous allons implémenter un design pattern Observateur.

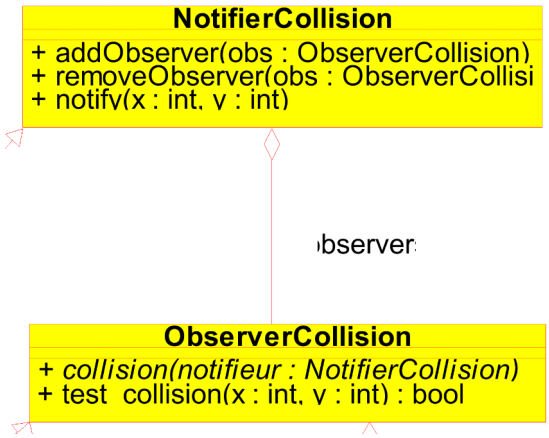
Les design patterns sont des façons de répondre à un problème de programmation de la façon la plus élégante et la plus fiable possible. Par exemple, si vous souhaitez réaliser une classe qui ne possède qu'une et une seule instance, il est possible de la construire en suivant le design pattern Singleton.

Le design pattern Observateur de son côté est utile lorsqu'une classe va « observer » la modification de l'état d'une autre classe. Une conception simple consisterait à vérifier régulièrement si la classe observée a été modifiée. Cependant cela va vite coûter beaucoup de ressources et la classe Observateur ne va pas réagir instantanément à un changement d'état.

Une façon de résoudre ce problème va consister à remplacer le fait que A observe B par le fait que B notifie A de ses modifications.

Cette manière d'écrire le code permet d'économiser les ressources et de réagir instantanément à un changement d'état de B. Cependant que se passe-t-il si plusieurs classes différentes doivent observer / notifier ?

Pour pouvoir résoudre ce problème, le comportement d'Observateur et de Notifieur est placé dans des classes dédiées, et les Observateurs et Notifieurs réels héritent de ces classes.



Voici une conception qui met en place le pattern Observateur dans le cadre du projet (le vrai pattern se trouve facilement sur internet). La classe Observateur « s'abonne » à un notifieur. Lorsque le notifieur est modifié, il appelle la fonction notify qui va appeler collision() sur chaque observateur en collision.

Pour n'appeler collision que sur les objets au contact, on rajoute une fonction test_collision qui permet de vérifier s'il y a une collision ou non.

La classe NotifierCollision sera réalisée par l'étudiant 1. Bien respecter les noms de fonctions pour éviter des problèmes lors de la fusion des travaux.

- Mettre en place l'interface ObserverCollision
- Faire hériter Ennemi de ObserverCollision. Définir les méthodes.
- test_collision(x, y) vérifie si les paramètres correspondent à la position courante d'Ennemi
- collision() indique qu'il y a une collision (message console pour le moment).
- Envoyer le travail sur le dépôt Git, régler les éventuels conflits de fusion.
- Attendre que l'étudiant 1 ait fini la classe Notifier, récupérer son travail via Git.

Si on récapitule, à chaque déplacement de Personnage, les ennemis sont informés de la nouvelle position via leur interface ObserverCollision. Ils vérifient s'ils sont sur la même case et si c'est le cas, le joueur perd une vie.

On souhaite maintenant réaliser le travail inverse : à chaque déplacement d'un ennemi, celui-ci va informer le Personnage de son déplacement et le Personnage va tester s'il se retrouve sur la même case que l'ennemi. Si c'est le cas, le joueur perd une vie.

- Faire hériter Ennemi de NotifierCollision.
- À chaque déplacement, appeler la méthode notify(x, y).
- Envoyer le travail sur le dépôt et récupérer le travail de l'autre étudiant une fois terminé.

À deux

Maintenant que les classes Personnage et Ennemi sont définies de façon à s'informer mutuellement de leurs déplacements, on va vérifier que le code fonctionne correctement.

- Assurez vous d'avoir bien synchronisé vos travaux avant de commencer.
- Dans le main, déclarer un Personnage et deux Clydes.
- Abonner Personnage aux deux Clydes.
- Abonner les deux Clydes à Personnage.
- Faire déplacer Personnage de façon à avoir une collision avec un Clyde. Est-ce que le message s'affiche ?
- Faire déplacer Clyde de façon à avoir une collision avec un Personnage. Est-ce que le message s'affiche ?
- Faire déplacer Clyde de façon à avoir une collision avec l'autre Clyde. Est-ce que le message s'affiche ?
- Synchroniser le travail sur le dépôt Git.
- Envoyer le lien du dépôt à l'enseignant si celui-ci est public, ou l'ajouter si celui-ci est privé (@DunLug sur Github et Gitlab, droits Reporter pour Gitlab).

TP3 : Des graphismes

Jusque là, nous avons mis en place une partie de la logique du jeu Capman. Cependant ce jeu ne va pas intéresser grand monde puisque le personnage et les ennemis sont déplacés par le code plutôt que par le joueur. De plus, un jeu en console va rapidement manquer de charme.

Pour aller un peu plus loin, nous allons mettre en place une interface graphique, lier les déplacements du personnage à des commandes clavier et les déplacements des ennemis à des timers.

Entre le TP2 et le TP3, vous avez dû voir apparaître une demande de pull request dans votre depôt Git. Une pull request permet de proposer des modifications à un code qui ne nous appartient pas. C'est une méthode très utilisée dans le logiciel libre puisqu'elle permet à n'importe qui d'apporter sa pierre à l'édifice, tout en laissant aux développeurs la possibilité d'accepter ou non les modifications (sinon ce serait le bordel).

La pull request envoyée par l'enseignant propose un ensemble de classes qui va gérer l'affichage graphique et l'interaction utilisateur. Le code est laissé en entier pour la curiosité mais ici, seules quelques classes vont être utilisées dans votre propre code :

- la classe Background sert de fond de fenêtre et de parents aux éléments ;
- la classe DrawableElem permet de représenter un élément fixe à une position donnée ;
- la classe Movable permet de déplacer cette image ;
- la classe RegularMove permet d'effectuer une action à chaque tour ;
- la classe Interactive permet de gérer le clavier ;

Précédemment, vous avez travaillé sur la même branche master pour réaliser le travail. Cela simplifie un peu la démarche mais pose deux problèmes :

- le code stable, prêt à être utilisé en production, n'est pas séparé des dernières modifications faites ;
- il est difficile de savoir à quoi correspond chaque commit.

Pour pouvoir résoudre ces problèmes, Git propose un système de « branches ». Une branche permet d'avoir plusieurs versions différentes du projet, de naviguer entre ces versions et de les fusionner.

Il est possible de créer un grand nombre de branches mais cela va rapidement rendre difficile le développement des nouvelles fonctionnalités et la fusion de celles-ci. Une manière

d'organiser ses branches est le « progressive stability branching ». Cette façon de faire sépare le code en trois grandes catégories de branches :

- la branche master contient le code stable, testé, prêt à être envoyé en production ;
- la branche develop regroupe les modifications non testées ;
- d'autres branches représentent les fonctionnalités en cours de développement.

Lors de l'ajout d'une nouvelle fonctionnalité ou de la correction d'un bug, on crée une nouvelle branche qui contient ces modifications. Le travail réalisé est donc séparé proprement des autres modifications et il est facile de revenir à une version antérieure s'il y a un problème ou si l'on souhaite se concentrer sur une autre fonctionnalité plus importante. Une fois la fonctionnalité prête, on la fusionne dans la branche develop. Après un certain nombre de modifications laissé à l'appréciation des développeurs, la branche develop est testée dans le détail, corrigée si besoin et fusionnée dans master. De ce fait il est facile de proposer une version stable du logiciel et une version beta.

Pour manipuler des branches, les commandes suivantes sont utiles :

```
173 git branch <nom_branche> # crée une branche en restant sur l'ancienne
174 git checkout -b <nom_branche> # crée une branche et va dessus
175 git switch -c <nom_branche> # crée une branche et va dessus
176 git branch -d <nom_branche> # supprime une branche fusionnée
177 git branch -D <nom_branche>  # force la suppression
178 git branch -vv # affiche le détail des branches
179 git checkout <nom_branche> # change de branche
180 git switch <nom_branche> # change de branche
```

Pour changer de branche, il est possible d'utiliser checkout ou switch. La commande checkout correspond à une ancienne commande de Git qui permet de changer de branche ou de remettre à zéro un fichier. Puisqu'elle est source de confusion, une nouvelle commande switch a été mise en place pour clarifier. De ce fait, la commande checkout est plus stable et présente dans toutes les versions de Git mais peut écraser votre travail en cas d'erreur de manip (impossible à récupérer), à l'inverse switch limite le risque de perte mais son comportement est susceptible d'évoluer par la suite.

Si vous utilisez git checkout, restez vigilant avant de lancer la commande pour ne pas perdre de travail. Les modifications pouvant être perdues en cas de fausse manip correspondent au modifications qui ne sont pas enregistrées dans un commit.

- Accepter la pull request et synchroniser le travail.

Étudiant 1

- Créer une branche develop depuis master et la pousser sur le dépôt.
- Créer une branche UI et switcher dessus.

On cherche à remettre le code de départ pour lancer une fenêtre graphique. Comme cette modification a été enregistrée dans Git, il est possible de la remettre en place

relativement facilement. Pour cela, on va rechercher le commit en question et l'inverser à l'aide de la commande git revert.

- Exécuter git log pour rechercher dans les premiers commits celui qui a retiré la classe MainWindow. Copier son identifiant.
- Vérifier les modifications à l'aide de git show -p <identifiant du commit>
- Renverser les modifications à l'aide de git revert <identifiant du commit>
- Résoudre les éventuels conflits de fusion (on garde le main graphique uniquement (version distante) et le mainwindow.ui de la pull request (version locale)).
- Enregistrer le travail dans un commit et revenir à la branche develop.
- Créer une branche personnage et switcher dessus.
- Modifier la classe Personnage pour qu'elle hérite de la classe Interactive. Supprimer les variables pos_x et pos_y dans la classe Personnage (présentes dans les classes mères).
- Définir les fonctions de gestion du clavier pour qu'elle modifie le déplacement du Personnage (fonctions void key_pressed(char key) et void arrow_pressed(string cmd))
- Modifier le constructeur pour qu'il initialise Interactive et utilise l'image du Personnage (set_background_color ou set_background_image).
- Définir la fonction appelée à intervalle régulier pour que celle-ci déplace le personnage, en fonction de la commande de l'utilisateur (void update_pos()).
- Enregistrer les modifications faites sur la branche Personnage et la pousser sur le dépôt.

Étudiant 2

- Récupérer le travail et switcher sur develop une fois celle-ci poussée.
- Créer une branche ennemi depuis develop et switcher dessus.
- Faire hériter Ennemi de RegularMove.
- Mettre à jour la position à chaque appel de void update_pos() dans Ennemi.
- Modifier le constructeur pour qu'il utilise l'image de Clyde (set_background_color ou set_background_image).
- Revenir sur la branche develop, créer une nouvelle branche pastille et basculer dessus.
- Créer une classe Pastille qui hérite de DrawableElem et de ObserverCollision.
- Afficher un message en cas de collision et supprimer la pastille (la supprimer des observateurs et la cacher avec hide()).
- Modifier le constructeur pour qu'il utilise l'image d'une pastille.
- Enregistrer les modifications faites sur la branche pastille et la pousser sur le dépôt.

À deux

Si vous regardez l'état de votre dépôt distant, vous devriez avoir six branches : la branche master qui contient le code stable, la branche develop et quatre branches de fonctionnalités. Nous allons maintenant fusionner les différentes modifications dans develop et tester pour pouvoir faire une fusion dans master.

- Synchroniser les travaux réalisés avec git fetch --all
- Exécuter git log --graph --all –oneline ou git graph

Les branches préfixées de origin/ correspondent à l'état du dépôt distant depuis la dernière synchronisation. Fetch ne permet pas de mettre à jour les branches locales pour correspondre aux branches origin/. Nous allons faire la fusion manuellement.

- Se placer sur la branche develop
- Exécuter git merge --no-ff origin/<nom_branche> avec personnage, ennemi, pastille et UI.
- Corriger les éventuels conflits de fusion.
- Vérifier l'état du dépôt local avec git log --graph --all --oneline.
- Vérifier que la compilation fonctionne correctement.

Pour terminer, on voudrait vérifier le fonctionnement du code avant de l'envoyer dans master. Nous allons instancier des objets pour les afficher et vérifier le bon fonctionnement des collisions. Les abonnements se font selon les règles suivantes :

– Personnage notifie les ennemis et les pastilles ;

– les ennemis notifient Personnage ;

- Dans le constructeur de MainWindow, créer un Personnage, une Pastille et un Clyde (allocation dynamique). L'élément background est défini par ui->background.
- Vérifier que Personnage change de direction lors d'un appui sur le clavier.
- Vérifier que Personnage se déplace régulièrement selon la direction donnée.
- Vérifier que Clyde se déplace régulièrement et de façon aléatoire.
- Ajouter les abonnements pour vérifier les collisions.
- Tester.
- Enregistrer les modifications dans un commit.
- Se placer dans la branche master.
- Fusionner la branche develop dans master.
- Synchroniser les modifications.

Nous avons terminé cette première version du jeu Capman. Je vous propose de regarder la conception présente dans l'annexe Conception du jeu.

- Comparer avec la conception faite au début, quels sont les points communs ? Les points qui améliorent la qualité du code ? Ceux qui pourraient être modifiés ?
- Comment cette conception aurait pu être modifiée pour respecter au maximum les principes SOLID ? Quels avantages cela apporte-t-il ?
- La classe Interactive ne doit pas être instanciée plus d'une fois, comment pourrait-on régler ce problème ?

TP4 : Pour aller plus loin

Le but de ce TP est cette fois de vous laisser rattraper un éventuel retard et d'avancer le projet selon votre propre convenance.

Voici les suggestions de modifications pouvant être apportées :

- Permettre au Personnage et aux Ennemis de traverser les bords pour se retrouver de l'autre côté de l'écran ;
- Créer d'autres ennemis avec chacun un comportement différent, la page Wikipédia de Pacman donne des exemples de comportements ;
- Afficher un compteur de score ;
- Modifier le compteur de score à chaque pastille mangée ;
- Afficher un compteur de vie ;
- Modifier le compteur de vie et réinitialiser le plateau lors d'une collision avec un ennemi ;
- Utiliser un fichier texte pour positionner les pastilles ;
- Mettre en place des murs empêchant le déplacement des ennemis et du personnage ;
- Mettre en place un screen de game over lorsque le compteur de vie atteint 0 ;

Annexe 1 : Cheatsheet Git

Les lignes grisées représentent des « bonus ».

Je souhaite créer mon dépôt

git init # Création d'un dépôt vide

ou

git clone <url> # Récupération d'un dépôt distant

git config [--global] user.name <mon nom> # visible par tous

git config [--global] user.email <mon email> # visible par tous

Je souhaite suivre de nouveaux fichiers

git add <fichier>

Je souhaite sauvegarder des modifications

git add -u / git add -p / git add <fichiers> # ajout des modifications

git commit -m <message de commit> # sauvegarde locale

git push # envoi des modifications sur le dépôt distant

Lors de la première poussée d'une branche, il est nécessaire de préciser la branche distante (exemple avec le dépôt origin et la branche master) :

git push -u origin master

Je souhaite récupérer le travail réalisé

git fetch --all # met à jour les branches distantes (origin/master etc.) sans toucher aux branches locales

git pull # met à jour les branches distantes et les fusionne avec les branches locale

Je souhaite travailler sur une version à part

git checkout -b <nouvelle branche> # crée une branche et bascule dessus

... # add, commit, push etc.

git checkout - / git checkout <branch> # retourne à la branche précédente (- est un raccourci).

Je souhaite fusionner les modifications

git checkout <branch> # se déplace sur la branche dans laquelle on récupère les modifs

git merge [--ff / --no-ff] <branch2> # récupère les modifications de branch2

git mergetool --tool=meld # résoudre les conflits de fusion

git commit # enregistre la résolution dans un commit de merge

Je souhaite récupérer des modifications

git cherry-pick <ref> # crée un nouveau commit contenant les modifications de ref

Je souhaite annuler des modifications

Modifications dangereuses, penser à faire une copie du dossier avant d'utiliser les commandes

git reset --hard # annule TOUTES les modifications non enregistrées dans un commit

git reset --hard <ref> # annule TOUTES les modifications faites après le commit

git checkout <fichier> # annule les modifications non enregistrées d'un fichier

git checkout -p # permet d'annuler finement les modifications

git revert <ref> # crée un nouveau commit qui « inverse » le précédent

git commit --amend # met à jour le dernier commit

Je veux chercher des informations dans le dépôt

git status # donne l'état courant du dépot (branche, modifications ajoutées, non ajoutées, fichiers non suivis...)

git log # donne la liste des commits depuis le commit courant

git log --graph --all --oneline --decorate # donne un graphe des commits

git log ... -- <fichier> # limite à des fichiers donnés (les renommages sont ignorés)

git diff # montre les modifications non ajoutées

git diff --cached # montre les modifications ajoutées mais non commitées

git diff HEAD # montre les modifications non commitées

git diff <ref 1> <ref 2> # montre les différences entre deux commits

git diff ... -- <fichier> # ne montre que les modifications faites sur les fichiers fournis

Options utiles pour le git log

--stat # donne les fichiers modifiés et leurs stats

-p # donne le diff pour chaque commit

--oneline # affiche une ligne par commit

-L:<fonction>:<fichier> # trace les modifications d'une fonction dans un fichier

--before=<date>/--after=<date> # filtre les commits par date

--author=<pattern> # filtre les commits par auteur

--pretty=<format> / --format=<format> # modifie le format (voir documentation)

-S<regex> --pickaxe-regex # recherche les commits qui modifient le nombre d'occurence de regex (--pickaxe-regex est inutile pour une chaîne fixe)

-G<regex> # recherche les commits dont les lignes modifiées contiennent regex

Options utiles pour le git diff

--summary # crée un résumé

--name-only # ne donne que les noms de fichiers modifiés

--name-status # ne donne que les noms de fichiers modifiés et leur statut

--word-diff=color # montre les différences par mots (une seule ligne)

--word-diff=porcelain # montre les différences par mots (plusieurs lignes)

-w # ignore les espaces, utile en cas de changement d'indentation

-W # donne toute la fonction en contexte

git branch -vv # montre les branches

git remote -vv # montre les dépôts distants

git bisect # fait une recherche dichotomique (regarder la documentation pour les détails)

git blame <fichier> # montre les commits ayant modifié en dernier chaque ligne

git grep # recherche des mots dans les commits (voir documentation pour les détails)

Annexe 2 : Utilisation avancée de ssh

SSH permet de chiffrer les communications entre deux ordinateurs à l'aide de clefs asymétriques. Un chiffrement asymétrique possède deux clés, une publique et une privée. La clef publique est utilisée pour chiffrer le message. La clef privée permet alors de le déchiffrer. On peut comparer ce comportement à celui d'une boîte aux lettres : n'importe qui peut déposer du courrier dedans mais seul le propriétaire de la clef peut lire le courrier.

Ici, la clef SSH permet de remplacer l'identification de l'utilisateur lors d'un envoi sur GitHub ou Gitlab. Le site connaît la clef publique de l'utilisateur et peut donc vérifier que celui-ci est bien la bonne personne (puisqu'il est le seul à avoir la clef privée). Pour cela, la clef publique doit être mise en place sur le site avant d'effectuer des push ou des pulls.

Il est possible d'avoir plusieurs clefs de chiffrement. Pour cela, on utilise la commande ssh-keygen. La commande est assez simple à utiliser et demande quelques informations pour créer une nouvelle clef. Deux fichiers sont alors crées. Le premier (par exemple clef), est la clef privée et ne doit pas être divulguée. Le deuxième (par exemple clef.pub), est la clef publique à transmettre à GitHub ou Gitlab.

Lorsqu'on a paramétré la clef sur le site, il arrive que la communication ne s'effectue pas malgré tout. SSH a besoin de savoir quelle clef utiliser. Pour cela, on peut ajouter un fichier ~/.ssh/config qui indique pour un hôte et un nom d'utilisateur la clef à employer.

```
181 Host nom
182   HostName gitlab.com
183   User git
184   IdentityFile c:/Users/utilisateur/.ssh/maclef
185   IdentitiesOnly yes
```

L'option IdentitiesOnly permet de n'utiliser que les clefs spécifiées dans le fichier de configuration et dans la ligne de commande. Sinon, SSH essaye d'utiliser les clefs enregistrées auprès de l'agent.

L'agent ssh est un programme auquel on ajoute des clefs. Lorsqu'une connexion est réalisée, l'agent essaye les clefs ajoutées pour réaliser la connexion.

Pour ajouter une clef, on utilise ssh-add.

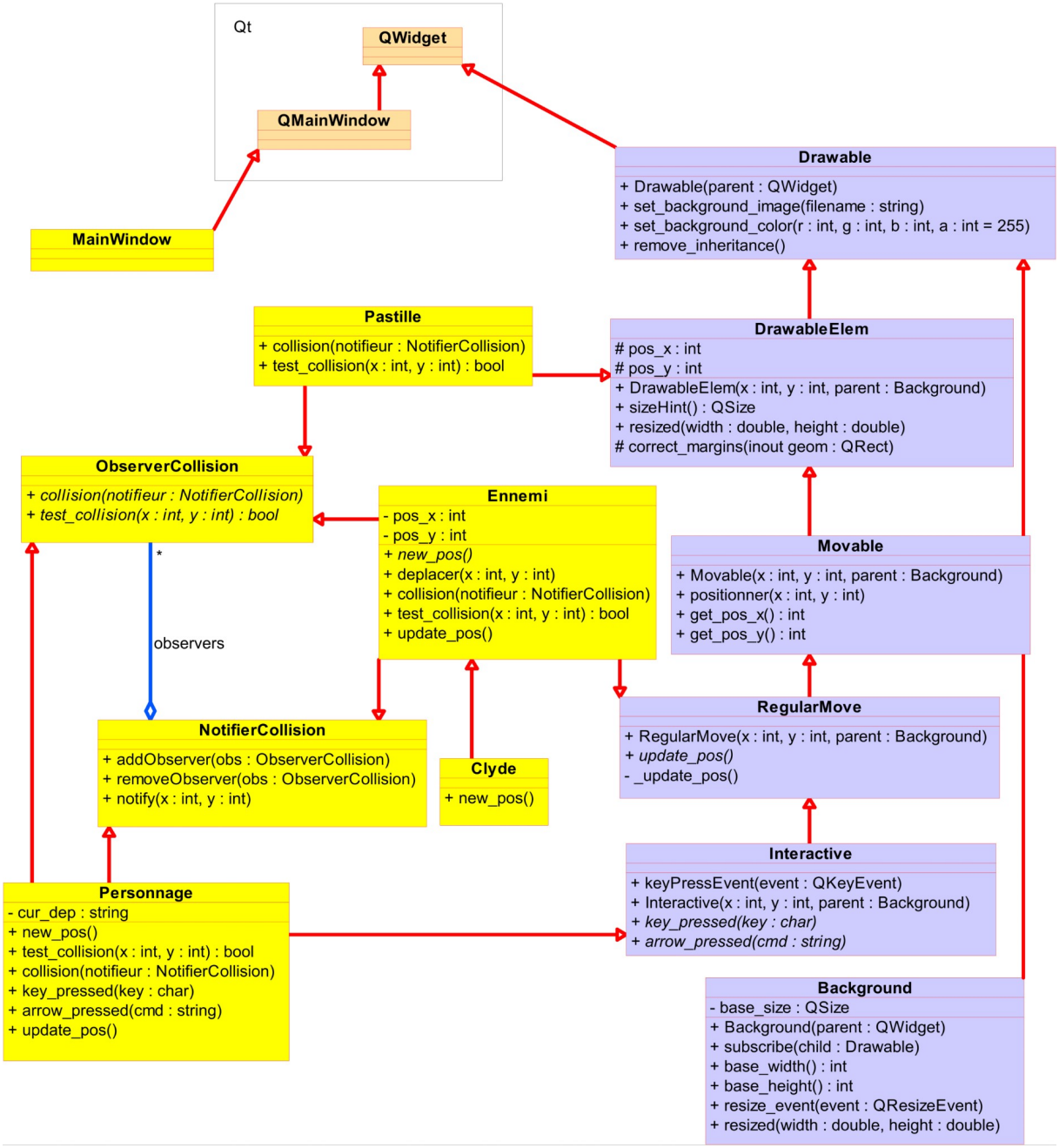
Il est possible de lister les clefs ajoutées avec l'option -l.

Si le message « Could not open a connection to your authentication agent » apparait, cela veut dire que l'agent n'est pas lancé. Pour lancer ssh-agent, on utilise la commande suivante :

```
186 eval `ssh-agent`
```

ssh-agent affiche sur la console des commandes pour définir les variables d'environnement utiles. Avec eval, ces affichages sont exécutés.

Annexe 3 : Conception du jeu



Annexe 4 : Installation de logiciels

QtCreator

Aller sur https://www.qt.io/download-open-source, cliquer sur Download the Qt Online Installer puis Download. Lancer l'exécutable.

Créer un compte sur Qt avec s'inscrire puis continuer dans l'installation.

Mettre C:\Qt comme répertoire d'installation (par défaut) et installation personnalisée. Faire suivant.

Dans le menu des composants, dérouler Qt, les options suivantes doivent être cochées au minimum (attention la taille augmente vite) :

- Qt <version 5 ou 6>
  - MSVC 2019 ou MinGW 11.2.0
  - Qt State Machine (optionnel)
- Developer and Designer Tools
  - Qt Creator 7.0.1 CDB Debugger Support
  - Debugger Tools for Windows
  - Qt Design Studio 3.3.0

Git

Aller sur https://git-scm.com/ et cliquer sur Download choisir le standalone installer ou la version portable.

Pour la version standalone, le logiciel demande un certain nombre d'options pour la configuration. Il est possible d'utiliser un éditeur graphique si vous n'utilisez pas vim habituellement.

Garder Let Git Decide pour le nommage des branches.

Configurer Use git from git bash only ou Git from the command line and also from 3rd-party software.

Use bundled SSH.

Pour la version portable, double cliquer sur Git Bash pour ouvrir le terminal.

Meld

Aller sur https://meldmerge.org/ et télécharger (Get It). Télécharger le MSI pour Windows et l'éxécuter.