Langage C avancé : Séance 10

Gestion de versions

Travail collaboratif

Marc TAJCHMAN
@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Versions du code

Un code, surtout s'il est écrit par plusieurs développeurs et/ou utilisé par plusieurs personnes, doit pouvoir évoluer dans le temps.

On rencontre souvent les situations suivantes (parmi d'autres):

- Les développeurs ajoutent de nouvelles fonctionnalités ou modifient des fonctionnalités existantes dans le code;
- Les utilisateurs trouvent des erreurs à l'exécution et les signalent aux développeurs;
- Un nouveau développement se révèle inutile ou produit des résultats faux et doit être abandonné;
- Plusieurs développeurs modifient la même partie du code, comment gérer les différentes modifications parfois contradictoires.

Il est important de pouvoir garder l'historique d'un code et de ses modifications. Cet historique se fait en définissant des versions du code.

Versions du code (2)

La façon de choisir les versions dépend de l'équipe de développement. On propose ici un exemple.

Chaque version est caractérisée par 3 nombres : numéro de version majeure, numéro de version mineure, numéro de correctif.

Exemples :

3.1.4 (majeur 3, mineur 1, correctif 4)
2025.06.5 (majeur 2025 (année), mineur 06 (mois), correctif 5)

On distingue souvent plusieurs niveaux de versions :

- Versions avec un nouveau numéro majeur ("stables"), diffusées à tous les utilisateurs

  Ce sont des versions avec des changements importants, qui sont bien testées. Toutes les modifications de la version précédente (mineures et correctifs) sont contenues dans la nouvelle version. Les autres numéros sont réinitialisés à 0.

Versions du code (3)

- Versions avec le même numéro majeur que la version précédente mais un nouveau numéro mineur, diffusées à certains utilisateurs "experts".
  C'est une version avec des changements plus localisés (une fonctionnalité ajoutée ou modifiée). Tous les correctifs de la version mineure précédente sont contenus dans cette version. Cette version est testée mais pas aussi bien qu'une version majeure.
  Le numéro de correctif est remis à 0.

- Versions avec les mêmes numéros majeur et mineur, mais un nouveau numéro de correctif, internes à l'équipe de développement
  C'est une version qui contient souvent la correction d'une erreur dans le code. En général, cette version n'est pas diffusée aux utilisateurs.

Bases de tests

Il est (très) utile de définir un ensemble de tests pour vérifier que les modifications du code conservent un fonctionnement normal de ce code.
Pour pouvoir effectuer une vérification efficace:

- Il faut tester toutes les fonctionnalités et passer par toutes les parties du code (couverture du code)
- Pour chaque valeur demandée à l'utilisateur, un test doit simuler tous les types de valeur.
- Quand il y a plusieurs choix possibles, un test doit utiliser chacun des choix.

Pour chaque test, il faut définir si c'est un test à utiliser pour chaque version majeure, mineure ou pour toutes les versions.
Si possible, mettre en place des procédures de vérification automatiques.

Rôles dans l'équipe de développement

Dans le cas où un code est développé par plusieurs personnes, il est (très) utile qu'une personne prenne le rôle de responsable du code, qu'une (ou quelques) personne(s) prenne(nt) le rôle d'intégrateur.

Le responsable du code répartit les modifications à réaliser aux différents développeurs en fonction des disponibilités et des compétences de chacun.

Chaque développeur fait une copie du code initial ou d’une version stable.

L'intégrateur est chargé d'examiner les modifications faites par les autres développeurs dans leur copie du code pour voir si les différentes modifications sont compatibles entre elles,

- si oui, il met les modifications dans le code en créant des versions internes
- Si non, il discute avec les développeurs impliqués, pour choisir quelles sont les modifications à garder, à modifier ou à abandonner.

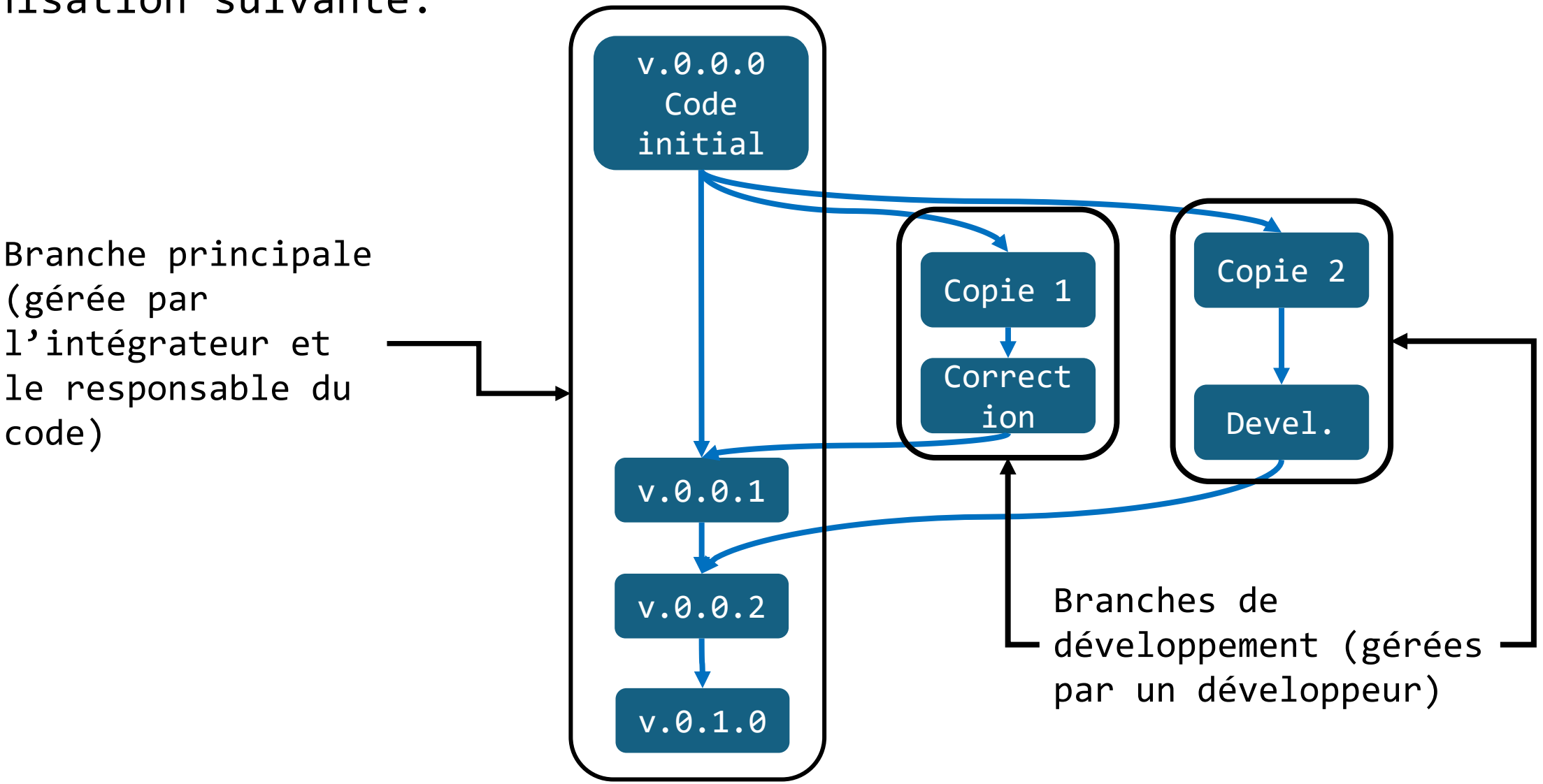
Quand le responsable de code juge que le code est suffisamment abouti, il publie une version majeure ou mineure.

Les développements suivants se feront sur cette nouvelle version majeure ou mineure.

Le fonctionnement décrit dans les pages précédente est un exemple d’organisation du développement, ce n’est pas une règle absolue !!

Arbre de développements

Chaque développement de code est particulier, on considèrera l'organisation suivante:



Outils de gestion de version

Outil de gestion de versions de code

Un outil logiciel de gestion de version est utile pour:

- Définir une version du code source
- Faire une copie du code source pour qu'un développeur y travaille
- Permettre à plusieurs développeurs de travailler des copies séparées du même code source sans se gêner les uns les autres
- Rassembler (fusionner) plusieurs copies différentes
- Revenir en arrière à une version antérieure du code
- …

Il existe plusieurs outils de ce type: rcs, cvs, git, mercurial, …

Dans la suite de l'exposé, on utilisera git qui est l'un des plus utilisés.

Exemple

Soit le code initial suivant:

main.c

```c
#include "data.h"

int main()
{
  data D;
  int n;
  n = f(&D) + g(&D);
  return 0;
}
```

data.h

```c
typedef struct s
{
  double x;
  double y;
} data;

int f(data *d);
int g(data *d);
```

f.c

```c
#include "data.h"

int f(data *d)
{
}
```

g.c

```c
#include "data.h"

int g(data *d)
{
}
```

Phase 0 : espace de référence

Pour pouvoir partager le code entre plusieurs développeurs, le responsable de code crée un répertoire vide (base_code) où vont être enregistrées toutes les versions:

```
git init --bare base_code
```

Pour le moment, la base de code (le répertoire base_code) ne contient aucun code source, seulement des fichiers utilisés par git lui-même.

Phase 1 : Code initial

L’intégrateur crée une première copie de la base de code

```
git clone chemin_vers_base_code main
```

(cette commande ne doit pas être tapée dans base_code)

Git crée un répertoire main qui va contenir la branche de développement principale.
L’intégrateur copie dans main le code source initial:

```
cp –rf chemin_vers_le_code_source_initial/* main
```

Dans main, il informe git que le répertoire src fait partie du code géré par git:

```
cd main
git add src
git commit –m "ajout initial"
```

Phase 1 : Code initial

Le responsable du code décide que ce code initial est la version 0.0.0 du code

```
git tag 0.0.0
```

L'intégrateur copie (« pousse ») le code initial dans la base git (et aussi le numéro de version)

```
git push origin 0.0.0
```

A ce moment, le code initial est disponible pour les autres développeurs

Phase 2 : Copie pour le premier développeur

Le développeur récupère une copie du code

```
git clone base_code dev1
```

Git crée un nouveau répertoire dev1 (git affiche un message d’erreur si dev1 existe déjà). Ce répertoire contient une copie de la branche principale.

Si le développeur veut faire des modifications, il doit créer une autre branche (sinon il va perturber la branche principale):

```
git checkout -b b1 0.0.0
```

Phase 3 : Copie pour le second développeur

Le développeur récupère une copie du code

```
git clone base_code dev2
```

Git crée un nouveau répertoire dev2 (git affiche un message d’erreur si dev2 existe déjà). Ce répertoire contient une copie de la branche principale.

Si le développeur veut faire des modifications, il doit créer une autre branche (sinon il va perturber la branche principale):

```
git checkout -b b2 0.0.0
```

Phase 4 : Le premier développeur fait des modifications dans sa branche

Le développeur modifie src/f.c, quand il tape :

```
git status
```

Git affiche la liste des fichiers modifiés, ajoutés ou supprimés:

```
On branch dev1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/f.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Pour enregistrer ses modifications locales, il tape

```
git add src/f.c
git commit –m 'modif de f'
```

Phase 4 : Le premier développeur fait des modifications dans sa branche (2)

Pour envoyer les premières modifications dans la base

```
git push --set-upstream origin b1
```

Cette commande crée une branche dans la base (de même nom b1 que la branche locale)

Pour envoyer les modifications suivantes, il suffira de taper

```
git push
```

Phase 5 : Le second développeur fait des modifications dans sa branche

Le développeur modifie src/g.c, quand il tape :

```
git status
```

Git affiche la liste des fichiers modifiés, ajoutés ou supprimés:

```
On branch dev1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/g.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Pour enregistrer ses modifications locales, il tape

```
git add src/g.c
git commit –m 'modif de g'
```

Phase 5 : Le second développeur fait des modifications dans sa branche (2)

Pour envoyer les premières modifications dans la base

```
git push --set-upstream origin b2
```

Cette commande crée une branche dans la base (de même nom b2 que la branche locale)

Pour envoyer les modifications suivantes, il suffira de taper

```
git push
```

Etat de la base et des copies

A ce stade, on a dans la base git, 3 branches « distantes » :

- La branche main avec le code initial
- La branche b1 avec les modifications du développeur 1
- La branche b2 avec les modifications du développeur 2

Dans la copie de l'intégrateur:

- La branche main locale avec le code initial

Dans la copie du développeur 1:

- La branche b1 locale avec les modifications du développeur 1

Dans la copie du développeur 2:

- La branche b2 locale avec les modifications du développeur 2

Report des modifications du développeur 1 dans la branche main

Se mettre dans la copie de l'intégrateur et taper

```
git origin/b1
```

pour intégrer les modifications de la branche b1

Examiner le fichier src/f.c pour voir s'il a été correctement modifié

Faire la même opération avec la branche b1 en tapant

```
git origin/b2
```

Remarque: les 2 développeurs ont travaillé sur les fichiers différents, la procédure simple ci-dessus fonctionne.

Il vaut mieux, à chaque étape, vérifier que les tests de la base de tests donnent toujours les mêmes résultats.

Hébergements de bases de code source gérées par git

L’exemple précédent utilisait une base locale.
Il existe plusieurs services (gratuits ou non, gérés par des entreprises ou des institutions : universités/centres de recherches)

Par exemple:

- GitHub
- Bitbucket
- GitLab

Tous ces services offrent un accès gratuit pour des codes de petite taille et des équipes réduites, mais sont payants sinon.
C’est un bon moyen pour se familiariser avec git.

Les institutions d’enseignement et/ou de recherche proposent aussi des service basés sur Git mais sont réservées en général à leurs membres.