PRG1 C++ - 2024 S1

Colin Stefani

Table des matières

1. Introduction

1.1. Bref historique du C/C++

Langage originaire de FORTRAN, un bordel sans nom où tout le monde veut améliorer le langage de l'autre, 3 mecs (Thompson, Ritchie, Kernighan) finissent par créer B qui deviendra C après la « traduction » de UNIX qui a engendré des ajouts de fonctionnalités.

Entrée dans le standard ANSI (American National STandards Institute), puis adopté par ISO.

Un mec super sympa de chez AT&T (Bjarne Stroustrup) ajoute des fonctionnalités de Simula (un des premiers langages de programmation OOP) à C++

Branching de C et et C++, mais C++ permet la programmation sous + de paradigmes (procédurale, orientée objet, générique), père de plein d'autres langages de haut niveau.

1.2. 1er programme en C++

Avec l'exemple suivant:

```c
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
  // intro
  cout << "Bienvenu(e)s au cours PRG1" << endl;
  // traitement ...
  // fin de programme
  cout << "fin de programme";
  return EXIT_SUCCESS;
}
```

Tout programme doit:

- Avoir une fonction `main()`
- Cette dernière doit retourner un entier `int`, représentant le statut de retour du programme (0 = success, anything else = error)

L'instruction `cout` permet d'écrire du texte sur le stream `stdout` du programme. L'instruction `endl` effectue un retour à la ligne. L'opérateur `<<` permet la transmission au flux (stream) `std::cout`

Des constantes de `<cstdlib>` existent pour les valeurs de retour: `EXIT_SUCCESS` (0) ou `EXIT_FAILURE` (-1). Ces constantes sont notamment ici pour éviter les « magic numbers », ces valeurs constantes (numériques, strings) qui rendent le code illisible.

Les fichiers **d'en-têtes** permettent l'utilisation de librairies externes, comme par exemple `<iostream>` ou `<cstdlib>`.

Les espaces de nommage (`namespace`) existent pour éviter la collision des noms de fonctions ou constantes, il peut être importé avec le mot-clé `using namespace NOM_DU_BORDEL`

Les instructions d'une fonction, se terminant par un `;`, sont placées entre des accolades `{...}`, qui forment un **bloc de code**.

1.3. Compilation

Le compilateur génère du **code objet**, normalement compréhensible par la machine.

```sh
g++ -c main.cpp
```

Il va ensuite faire passer tout le bazar dans l'éditeur de lien (« linker », e.g. `ld`), qui va lier tous les fichiers, toutes les librairies, permettant enfin à la machine de lancer l'éxécutable produit.

```sh
g++ main.o -o main
```

Les deux commandes précédentes peuvent être combinées avec `g++`:

```sh
g++ main.cpp -o main
```

D'autres arguments comme `--pedantic` (normes ISO), `-Wxx` (warnings) ou `-std=c++20` (standard C++) peuvent être passés à `g++` pour configurer son comportement à la compilation et au *linkage*.

2. Utilisation de `git`

La commande `git` permet la gestion **versionnée** de projets (VCS, Version Control System). Cela fonctionne particulièrement bien en programmation lorsqu'on travaille à plusieurs sur un seul et même projet, car plusieurs personnes travaillent sur leur propre version (**branche**), et on fusionne ensemble toutes les modifications (**merge**).

2.1. Création du dépôt

L'intégration de git à un projet se fait via la commande:

```sh
git init
```

Ceci va créer un répertoire `.git` « vide » dans le dossier actuel. Ce dernier est chargé de stocker toutes les version précédentes ainsi que toutes les métadonnées liées au projet.

> ⚠ **Avertissement**
>
> On confond souvent le service GitHub avec l'outil `git`. GitHub est simplement une plateforme avec laquelle on interragit via `git` ! Ce dernier est uniquement un client suivant des standards pour le versionnage de projets. On pourrait très bien utiliser `git` avec un autre service en ligne tel que GitLab ou BitBucket.

La première étape est de relier notre projet avec sa version en ligne afin de pouvoir les synchroniser. On commence par créer un dépôt (**repository**) sur GitHub, puis on l'ajoute à git:

```sh
git remote add origin https://github.com/colinstfni/labo01.git
```

La commande `git remote add` permet d'ajouter un référentiel, ce qui permet à `git` de savoir où envoyer et récupérer les versions lors de la synchronisation. On ajoute ici le dépôt « lab01 » de l'utilisateur « colinstfni », ces valeurs sont évidemment à changer.

2.2. Authentification avec GitHub

On a deux options:
- le CLI de GitHub: `gh`
- pour les masos, à la main avec les clés SSH (chiant)

Pour les gens normaux, avec le CLI, une fois téléchargé et installé correctement:

```sh
gh auth login
```

On séléctionne « GitHub.com » pour le type de compte, « HTTPS » pour le protocole et enfin « Login with a web browser ». Le CLI va vous donner un code:

```
! First copy your one-time code: ABCD-1234
```

On copie le code comme demandé et on appuie sur Entrée, ce qui devrait ouvrir un navigateur, si jamais il ne s'ouvre pas: https://github.com/login/device

Une fois connecté et le code collé, la commande devrait s'être terminée, et en tapant `git auth status` on devrait avoir un truc du genre:

```
❯ gh auth status
github.com
  ✓ Logged in to github.com account colinstfni (keyring)
  - Active account: true
  - Git operations protocol: https
  - Token: gho_************************************
  - Token scopes: 'gist', 'read:org', 'repo', 'workflow'
```

Il faut maintenant configurer `git` correctement avec la commande:

```sh
gh auth setup-git
```

Et pouf c'est magique ça marche!

Dans le cas où vous voulez travailler avec plusieurs comptes GitHub sur le même ordinateur, vous pouvez répéter la procédure précédente et changer de compte avec:

```sh
gh auth switch -u <USER>
```

2.3. Rédaction de changements

On peut maintenant effectuer des changements qui seront traqués par `git`. Si on crée un fichier « pouet »:

```sh
touch pouet
git status --short
```

```
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  pouet
```

On voit que `git` détecte notre fichier, mais nous annonce qu'il n'est pas encore inclus « dans ce qui sera commit ».

> **Info**
>
> Un « commit » est un peu comme un carton dans lequel on met tous nos changements avant de le sceller et de l'envoyer. Lorsqu'un commit est créé, on lui assigne un **hash** (e.g. *a08dbe8*) qui permet de l'identifier.

On concocte donc notre emballage (notre commit en gros):

```sh
git add pouet # On ajoute le fichier "pouet" au prochain commit
git commit -m "Ajout de pouet" # On crée le commit avec son message
```

```
[main (root-commit) ce7cba2] Ajout de pouet
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 pouet
```

Notre carton est prêt, plus qu'à l'envoyer !

```sh
git push
```

Cette dernière commande lance la synchronisation avec le remote tout juste configuré, en **poussant** (push) les changements. La synchronisation dans le sens inverse peut se faire avec la commande `git pull` (tirer).

> ⚠ **Avertissement**
>
> **Toujours** `git pull` avant de `git push` ;) ça vous évitera des problèmes

2.4. Concept de branches

Les branches sont une notion commune dans les VCS, ils servent à continuer de travailler sur le projet sans altérer la « main-line ». Dans `git`, un projet est représenté de manière linéaire:

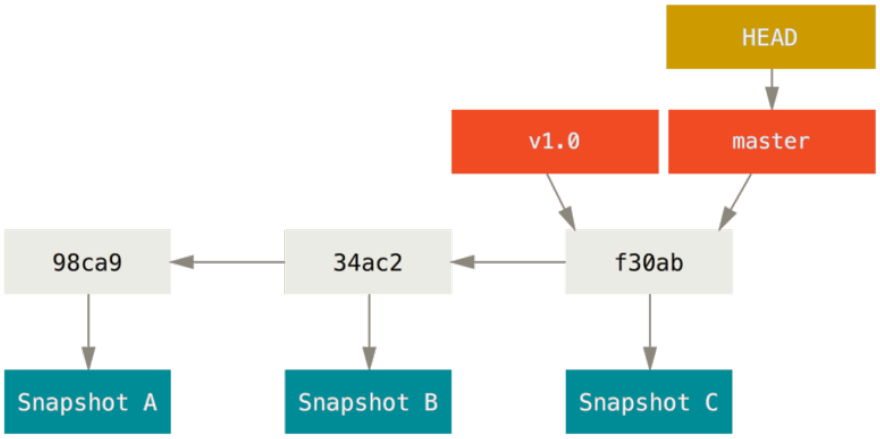

Fig. 1. – Une branche et son historique de commits, Scott Chacon and Ben Straub

Au fur et à mesure des commits, le projet change. Dans le cas de figure où deux personnes travaillent sur le même projet et l'une veut implémenter une nouvelle fonctionnalité pendant que l'autre continue de travailler sur le projet lui-même, la création d'une branche serait pertinente:

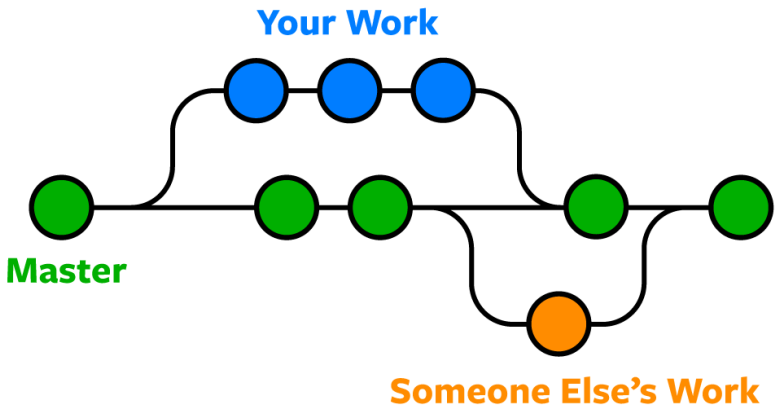

Fig. 2. – Plusieurs branches git fusionnées ensemble

Afin de savoir sur quelle branche on se trouve actuellement, `git` utilise également un pointeur spécial `HEAD`. Par exemple, si on se trouve actuellement sur `master`et qu'on crée une branche `testing`:

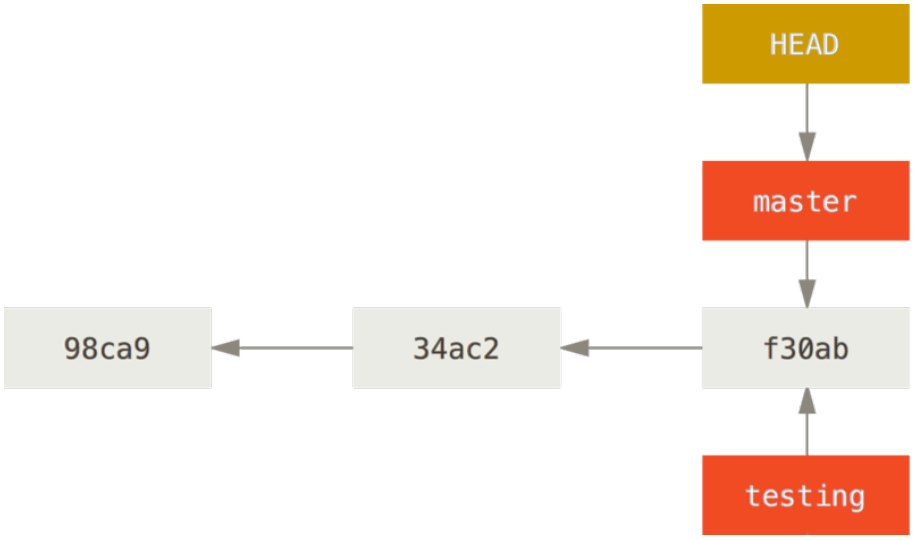

Fig. 3. – Le pointeur `HEAD` pointant sur `master`, Scott Chacon and Ben Straub

Et lorsqu'on change de branche avec `git switch testing` ou `git checkout testing`:

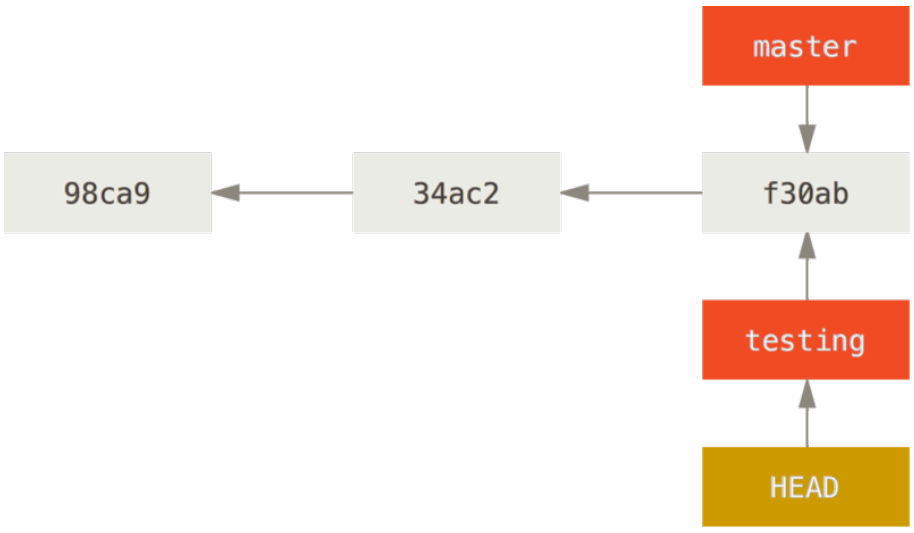

Fig. 4. – Le pointeur `HEAD` pointant sur `testing`, Scott Chacon and Ben Straub

Lorsqu'un commit sera fait sur la branche testing, le pointeur `HEAD` le suivra, on peut donc le considérer comme un « stack pointer » des commits sur la branche actuelle. Ce pointeur peut être vu lorsqu'on exécute la commande `git log --oneline`:

```sh
git log --oneline
```

```
b460714 (HEAD -> main, origin/main) Add stuff
e55f0dd Update compile.yaml
50bcf7d Update README
16f7123 Add a README
```

Ici, `HEAD` pointe vers `main`, avec le dernier commit étant `b460714`.

2.5. Résolution de conflits

Des fois on se retrouve avec des trucs du genre:

```
CONFLICT (content): Merge conflict in main.cpp
```

1. Ne pas paniquer, ni tout supprimer ou même jeter son ordi par la fenêtre
2. Les clients `git` desktop sont vos amis dans ces cas là.

La résolution de conflits à la main juste avec un éditeur de texte est largement faisable, mais tellement plus simple avec les bons outils. Les plus puristes diront que c'est de la triche mais bon…

VSCode inclut par défaut une interface `git` très facile d'utilisation, la plupart des autres IDEs également. Les conflits viennent du fait que les deux versions du projet que vous avez tenté de synchroniser ont des différences trop complexes pour avoir été fusionnées automatiquement, ça arrive souvent quand on bosse à plusieurs sur un même fichier.

Pour comprendre un minimum comment ça fonctionne derrière, il est conseillé de faire un premier merge à la main comme un(e) vaillant(e).

Les conflits sont dans le format texte suivant:

```
1  <<<<<<<
2  Les changements sur HEAD (la branche actuelle, current)
3  |||||||
4  La dernière version commune
5  =======
6  Changement de votre branche qui diffère (incoming)
7  >>>>>>>
```

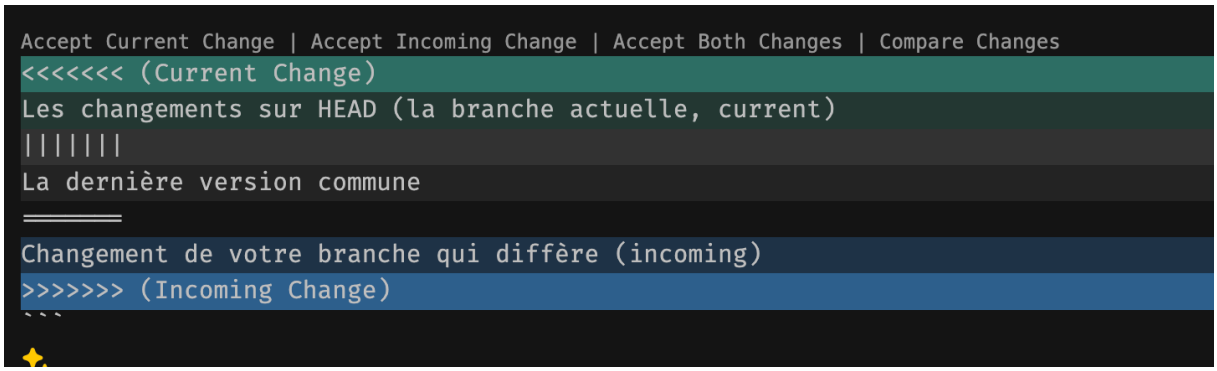

Fig. 5. – L'aide de VSCode lors d'un conflit de fusionnage

Au delà de résoudre des conflits, le mieux c'est quand même de ne pas en créer: pour ce faire, il est préférable de « modulariser » le projet en créant différents fichiers pour chaque fonctionnalité.

3. Bases de C++

3.1. Déclaration de variables

Chaque variable en C++ doit être déclarée à l'aide de son **type**. On peut lui assigner une valeur directement: `int x = 9;` ou simplement la déclarer: `int x;`.

Les variables sont utiles pour:
- Stocker des valeurs dynamiques (input utilisateur, résultat d'une opération)
- Faciliter la lecture et modification d'un programme

Avec l'exemple suivant:

```cpp
cout << "Nb bouteilles dans un pack : " << 6;
cout << "Volume d'un pack(l) = " << 0.33 * 6;
cout << "Poids d'un pack(g) = " << 13.2 * 6;
cout << "Entrer le Nb de pack à expédier :";
cout << "Le poids de votre colis(g) :" << 13.2 * 6 * nb_pack;
```

On pourrait le réécrire avec des variables:

```cpp
int nb_bouteilles = 6;
cout << "Nb bouteilles dans un pack : " << nb_bouteilles;
int vol_bouteille = 33;
cout << "Volume d'un pack(cl) = " << vol_bouteille * nb_bouteilles;
int poids_bouteille = 13;
int poids_pack = poids_bouteille * nb_bouteilles;
cout << "Poids d'un pack(g) = " << poids_pack;
cout << "Entrer le Nb de pack à expédier :";
int nb_pack = 0;
cin >> nb_pack;
cout << "Le poids de votre colis(g) :" << poids_pack * nb_pack;
```

Info

Une variable peut être initialisée de plusieurs manières en C++:
- « Comme en C » `int age = 6;`
- « Par constructeur » `int age(6)`
- « Uniforme » `int age{6}` (11+)

Une variable:
- **doit** commencer par _ ou une lettre
- **peut** contenir des lettres, des chiffres et _
- **ne peut pas** contenir d'espaces ou de caractères spéciaux

- **ne peut pas** être un mot réservé C++ (`while`, `true`)

3.2. Core guidelines

Des règles de nomenclature sont définies dans le C++ Core book:

- **NL5** : Le nom d'une variable ne doit pas mentionner son type (i.e. n'utilisez pas la notation hongroise)
- **NL7** : La longueur d'un nom de variable doit être ± proportionnelle à sa portée (distance entre ses utilisations)
- **NL8** : Utilisez une nomenclature consistante : snake_case, camelCase, PascalCase, ...
- **NL9** : N'utilisez pas `TOUT_EN_MAJUSCULE` pour les identificateurs autres que les macros (vues en PRG2)
- **NL10** : Préférez le style snake_case, c'est le style utilisé par la Standard Template Library. Utilisez éventuellement Majuscule_initiale pour les types que vous définissez vous-même, comme Bjarne Stroustrup.

3.3. Constantes

Les variables constantes ne changent **pas** de valeur, on dit qu'elle sont **immutables**.

On les déclare de la manière suivante:

```cpp
const int meaning_to_life = 42;
```

Son initialisation à la déclaration est **obligatoire**.

> **Info**
>
> C++ Core guideline **CON.1**: Par défault, toutes les variables doivent être constantes. (Rust-like idiom)

3.4. Types de base

En C++, toutes les données sont **typées**. Un type définit:

- Comment la donnée est stockée en mémoire (taille)
- Les opérations possibles

Les types fondamentaux fournis par le language permettent de stocker les données les plus basiques:

- Caractères (`char`, 1 byte, code ASCII)
- Nombres entiers (`int`)
- Nombre réels (`float`, `double`, le `double` est **2 fois** plus précis que le `float`)
- Booléens (`bool`)
- Chaînes de caractères (`string`, classe)

Le mot-clé `auto` peut être utilisé pour laisser le compilateur déduire automatiquement le type de la variable.

3.5. Expressions

En C++, tout ce qui correspond à une valeur est une expression. On dit qu'elle « renvoie » une valeur.

La plus simple des expressions est une constante exprimée littéralement:

```cpp
'a' // char
"sdkfjgshdkg" // string
42 // int
3.14 // double
```

> **Info**
>
> La `lvalue` est la valeur de **localisation**, par exemple le nom de la variable. La `rvalue` est la valeur de **résultat**, par exemple un littéral constant, les résultats d'expression (qui ne se résolvent pas en `lvalues`, e.g. `a+b`).

3.6. Opérateur d'affectation

L'opérateur `=` copie la valeur de l'expression de droite à gauche. Il renvoie également lui-même une `lvalue`. Toute expression d'affectation s'effectue de **droite à gauche**.

```cpp
x = y = 42;
```

Ici, d'abord `y = 42;`, puis `x = y;`, donc `x = 42;`

3.7. Arithmétique sur les réels/entiers

Les types **réels** (`float`, `double`), disposent des opérateurs `+, -,*,/` qui se comportent comme en maths.

Les types **entiers** (`int`, …) disposent des opérateurs `+, -, *, /, %`

> **Avertissement**
>
> La division `/` entière renvoie un nombre **entier**. On a toujours l'égalité suivante:
>
> $$\left(\frac{a}{b}\right) * b + (a\%b) = a$$
>
> $$a \equiv r \pmod{b} \Rightarrow \left(\frac{a}{b}\right) * b + r = a$$
>
> Le modulo `%` renvoie le reste de la division euclidienne, son signe correspond au signe de a pour $a \equiv r \pmod{b}$ ou `a % b`