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:

```cpp
#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.

```
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.

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

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

```
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:

```
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:

```
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:

```
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:

```
gh auth setup-git
```

Et pouf c'est magique ça marche!

2.3. Rédaction de changements

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

```
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):

```
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 !

```
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. 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:

```
<<<<<<<
Les changements sur HEAD (la branche actuelle, current)
|||||||
La dernière version commune
=======
Changement de votre branche qui diffère (incoming)
>>>>>>>
```

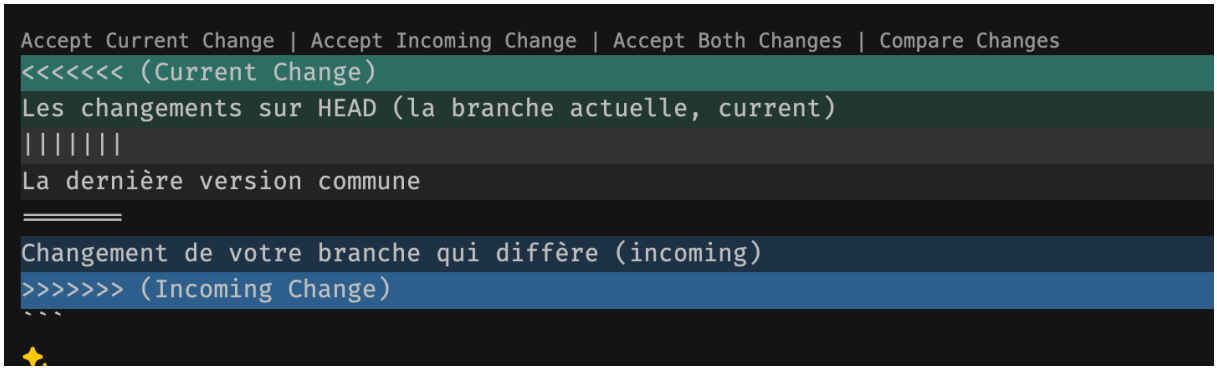

Fig. 1. – L'aide de VSCode lors d'un conflit de fusionnage

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.