Techniques et outils pour la programmation

PG110 Projet programmation

Philippe SWARTVAGHER
ph-sw.fr

BORDEAUX INP Enseirb-Matmeca

À propos de ce cours

Organisation

- 3 séances de CI ($3 \times 2 \times$ 1h20)
  Outils et techniques pour la programmation
- 7 séances de projet ($7 \times 2 \times$ 1h20)
  Réalisation d'un Pac-Man

Objectifs

- Être plus à l'aise en programmation C
- Savoir utiliser les outils pour débugguer des programmes C

Évaluation

- Projet : code, rapport et soutenance

Pré-requis

- Avoir suivi les cours de PG109 et IF110

Ressources et crédits

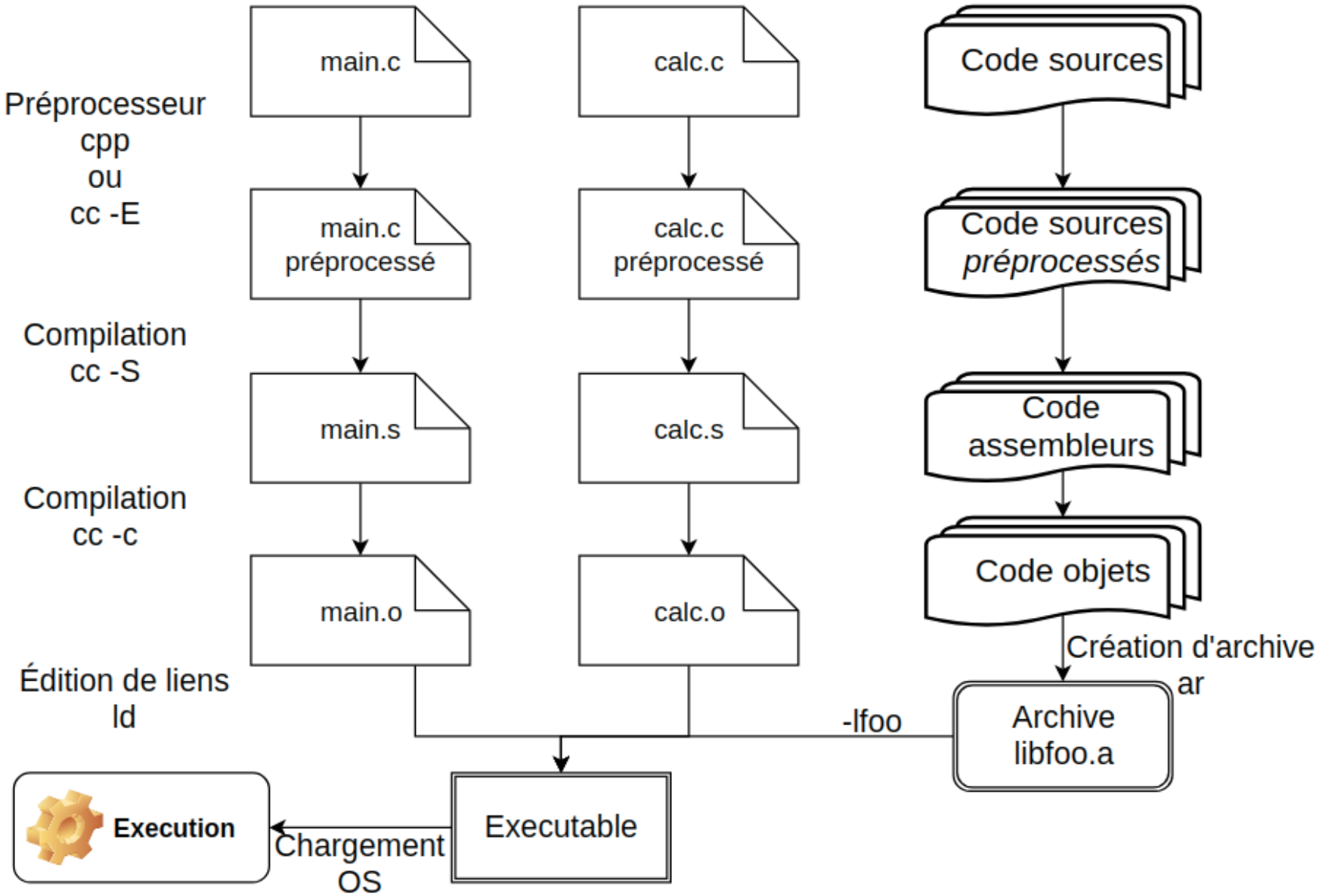
Ce cours est basé, entre autres, sur :

- Le cours de Julien Allali : https://www.labri.fr/perso/allali/?page_id=511
- Les cours de Laurent Réveillère et François Pellegrini
- Les liens donnés à la fin de chaque partie

Sommaire

Construction de bibliothèques

Rappel : chaîne de compilation



Bibliothèques logicielles

- Grouper un ensemble de fichiers objets (`*.o`) dans un unique fichier.
- Cet unique fichier est une *bibliothèque* logicielle.
- Il est ainsi possible d'avoir une unique copie de la bibliothèque, qui est utilisée par plusieurs programmes différents.
- Facilite la manipulation, les mises à jour, ...

Dans l'exemple précédent, `calc.o` pourrait être transformé en bibliothèque.

Deux types de bibliothèques :

- Statiques
- Dynamiques

En français, on dit « *bibliothèque* », pas « *librairie* » ou « *library* » !

Bibliothèques statiques

- Nommées `lib*.a`
- À la compilation (plus précisément : lors de l'édition de liens) :
  - Si un des symboles de la bibliothèque est utilisé, la bibliothèque est incluse dans le programme
    $\Rightarrow$ Peut augmenter (grandement) la taille des exécutables
- À l'exécution :
  - L'exécutable n'a plus besoin de la bibliothèque pour fonctionner

```
gcc -c ticket_tab.c -o ticket_tab.o # construit le .o

# construit la bibliothèque statique
# (peut contenir plusieurs .o)
ar rcs libticket_tab.a ticket_tab.o

# compilation du programme avec la bibliothèque
# -L. : cherche aussi les bib. dans le dossier courant
# -lticket_tab : lie avec libticket_tab.a
gcc -o cafeteria_tab cafeteria.c -L. -lticket_tab
```

Bibliothèques dynamiques

- Nommées `lib*.so` sous Linux, `lib*.dylib` sous MacOS, `*.dll` sous Windows
- À la compilation (plus précisément : lors de l'édition de liens) :
  - Si un des symboles de la bibliothèque est utilisé, un lien est fait vers la bibliothèque
- À l'exécution :
  - La résolution de symboles (`ld.so`) cherche une bibliothèque correspondant au lien et l'utilise
- Une bibliothèque dynamique est installée en un exemplaire, puis peut être utilisée par différents programmes
  $\Rightarrow$ Bibliothèques dynamiques aussi appellées bibliothèques *partagées*

```
# construit la bibliothèque dynamique
# -shared : construit une bib. partagée
# -fPIC : Position Independant Code : le code de la
    bibliothèque peut être chargé n'importe où en mé
    moire
gcc -shared -fPIC ticket_tab.c -o libticket_tab.so

# compilation du programme avec la bibliothèque
# -lticket_tab : lie avec libticket_tab.so
gcc -o cafeteria_tab cafeteria.c -L. -lticket_tab
```

Manipulation de bibliothèques

- `nm $binaire` : liste les symboles d'un binaire
- `ldd $binaire` : liste les bibliothèques dynamiques requises par un binaire

- Pour la compilation, on peut omettre l'option `-L` en utilisant la variable d'environnement `LIBRARY_PATH` [1].
- Pour l'exécution, on peut préciser où chercher les bibliothèques dynamiques avec la vairable d'environnement `LD_LIBRARY_PATH` [2].

Forcer l'utilisation d'une bibliothèque statique (la bibliothèque partagée étant utilisée en priorité [3]) :

```
gcc -o cafeteria cafeteria.c -L. -l:libticket_tab.a
# ou, plus simplement :
gcc -o cafeteria cafeteria.c libticket_tab.a
```

Placez les options pour lier les les bibliothèques (`-L`, `-l`) comme derniers paramètres du compilateur !

1. https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/Environment-Variables.html
2. `man ld.so`
3. `man ld`

L'importance des fichiers d'en-tête

(Rappel) Fichiers `.h` contenant (surtout) les prototypes des fonctions définies dans les fichiers `.c`

Lors de l'utilisation de bibliothèques, on n'a pas forcément accès aux sources (fichiers `.c`), mais on a besoin d'avoir les fichiers d'en-tête :

- À la compilation : vérification de la cohérence des paramètres (nombre, types, ...) et l'existence (présumée) des fonctions appelées grâce aux déclarations dans les fichiers d'en-tête.
  Une fonction inconnue produira *seulement* un `warning: implicit declaration of function`.
- Lors de l'édition de lien (compilation et exécution) : vérification seulement de l'existence des fonctions (pas de vérification de la cohérence des paramètres).

Distribution des bibliothèques et fichiers d'en-tête

Exemple : un programme dynamiquement lié avec la SDL, système Debian

- Pour l'exécuter : seulement besoin de `libSDL2-2.0.so.0`
  $\Rightarrow$ fournie par le paquet `libsdl2-2.0-0`
- Pour le compiler : besoin en plus des fichiers d'en-tête, *eg* `SDL.h`
  $\Rightarrow$ fournis par le paquet `libsdl2-dev`

pkg-config

Où trouver les bibliothèques installées sur le système ? Quels paramètres précisément passer à `gcc` ?

1. RTFM
2. `pkg-config` peut aider

En plus des fichiers d'en-tête, les bibliothèques[4] fournissent des fichiers `.pc` pour que le programme `pkg-config` puisse répondre à ces questions.

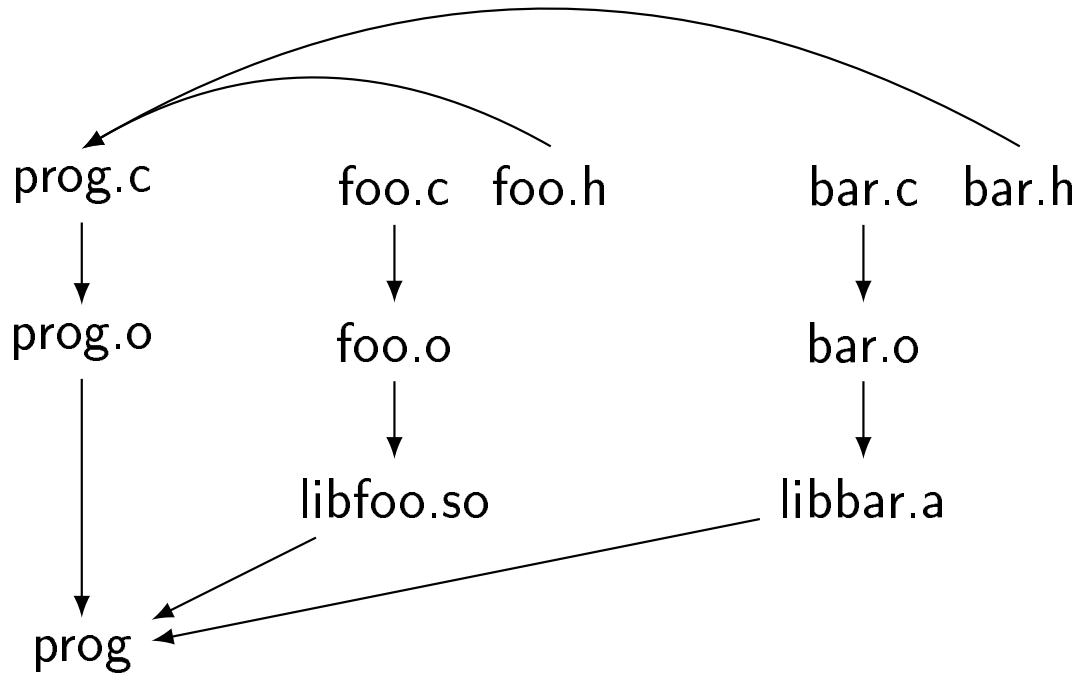
```
pkg-config --list-all | grep -i sdl
sdl12_compat  sdl12_compat - An SDL-1.2 compatibi...
sdl2          sdl2 - Simple DirectMedia Layer is ...
SDL2_image    SDL2_image - image loading library ...

pkg-config --cflags sdl2
-I/usr/include/SDL2 -D_REENTRANT

pkg-config --libs sdl2
-lSDL2
```

4. Celles qui sont bien faites...

Quelques questions

Avec le graphe de dépendences suivant, que faut-il recompiler si je modifie...

1. `prog.c` ?
2. `foo.c` ?
3. `foo.h` ?
4. `bar.c` ?
5. `bar.h` ?



Exercice

Reprenez votre code (ou la correction) du TP5 de PG109, notamment `cafeteria.c` et `ticket_tab.c`.

1. Ajoutez dans la fonction `ticket_nouveau()` l'affichage d'un message
2. Compilez `ticket_tab.c` en une bibliothèque dynamique `libticket.so`
3. Compilez `cafeteria.c` en utilisant `libticket.so`
4. En utilisant `LD_LIBRARY_PATH=$(pwd)`, exécutez `./cafeteria`
5. Changez le message affiché dans la fonction `ticket_nouveau()`
6. Compilez `ticket_tab.c` en une bibliothèque dynamique `libticket.so` **dans un autre dossier**
7. En utilisant `LD_LIBRARY_PATH=/autre/dossier`, exécutez `./cafeteria`

Qu'observez-vous ? Pourquoi ?

À vous de jouer !

Makefile

Motivation

- Marre de chercher et écrire ces longues lignes de commandes pour compiler ?
- Marre d'oublier de recompiler quelque chose ?
- Comment appliquer les mêmes options de compilation à l'ensemble du projet ?

Vous allez aimer `make` et ses `Makefiles` !

Règles make

`make` est un programme qui génère des fichiers s'ils n'existent pas ou s'ils sont obsolètes, selon un ensemble de règles définies.

Exemple de règle :

```
cible: source1 source2
    commande1
    commande2
```

Se lit : « si le fichier `cible` n'existe pas ou s'il est plus ancien que `source1` ou `source2`, exécute les commandes `commande1` puis `commande2` ». L'exécution de ces commandes est censée générer `cible`.

Les commandes à exécuter sont des commandes shell

Attention ! Chaque commande est préfixée d'une tabulation ! Pas d'espaces !

Fichier Makefile

Ces règles se placent dans un fichier nommé (obligatoirement) `Makefile` :

```
foo.o: foo.c
  gcc -c foo.c -o foo.o

bar.o: bar.c
  gcc -c bar.c -o bar.o

prog: prog.c foo.o bar.o
  gcc prog.c foo.o bar.o -o prog
```

Exécution :

```
make prog
```

`make` est récursif : si `foo.o` n'existe pas, il va exécuter les commandes nécessaires pour le construire.

Commande `make`

`make` invoqué sans paramètre va par défaut générer la première cible définie dans le `Makefile`.

Astuce : utiliser comme première règle :

```
all: prog
```

« Pour construire `all`, tu as besoin de `prog`, puis il n'y aura rien d'autre à faire. »

Option `-j` pour parallèliser l'exécution des différentes règles à exécuter.

Variables

Possibilité de définir des variables :

```
NOM=VALEUR
```

Utilisation : `$(NOM)`

Les variables peuvent être modifiées par les arguments de `make` :

```
make foo NOM=autre_valeur
```

Les variables d'environnement sont disponibles dans un `Makefile`, mais sont écrasées par les définitions comme ci-dessus. Pour garder les valeurs des variables d'environnement et avoir une valeur par défaut :

```
NOM?=VALEUR
```

```
NOM=autre_valeur make foo
```

Variables particulières

À utiliser dans les commandes à exécuter par les règles :

- `$@` : nom de la cible
- `$^` : toutes les dépendences
- `$<` : première dépendence
- RTFM : https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

```
foo.o: foo.c
    gcc -c $^ -o $@

bar.o: bar.c
    gcc -c $^ -o $@

prog: prog.c foo.o bar.o
    gcc $^ -o $@
```

Variables usuelles

Parfois prédéfinies.

- `CC` : compilateur C (`C Compiler`)
- `CFLAGS` : options à passer au compilateur (`-Wall -Werror -I ...`)
- `LDFLAGS` : options à passer à l'éditeur de lien (`-L ...`)
- `LDLIBS` : les bibliothèques avec lesquelles se lier (`-lfoo ...`)
- RTFM : `https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html`

Règles génériques

```
%.o: %.c
	gcc -c $^ -o $@
```

« Pour construire un fichier `.o`, tu as besoin d'un fichier `.c` du même nom et exécuter la commande `gcc ...` »

Séparation des dépendances et des règles :

```
prog.o: prog.c foo.h bar.h
foo.o: foo.c foo.h
bar.o: bar.c bar.h
%.o:
	gcc -c $< -o $@
```

Règles usuelles

- `all` : déjà présentée
- `clean` : supprime tout ce qui a été généré (`.o`, exécutables, ...)
- `install` : installe les binaires dans les répertoires système
- `check` ou `test` : lance les tests

Ces règles ne produisent pas de fichier, on peut l'indiquer à `make`, notamment dans le cas où il existe un fichier qui porte le nom de ces règles :

```
.PHONY=all clean install check
```

Règles implicites

Si un fichier `foo.c` existe, par défaut (sans règle explicite), `make foo` exécute :

```
$(CC) $(CFLAGS) $< -o $@ $(LDLIBS)
```

Pour connaître toutes les règles et variables connues de `make` :

```
make -p
```

Pour ne pas se faire influencer par un éventuel `Makefile` qui serait présent dans le dossier courant :

```
make -p -f /dev/null
```

Génération de listes de fichiers

Pour stocker dans une variable tous les fichiers `.c` :

```
SRC=$(wildcard *.c)
```

Pour changer l'extension des fichiers contenus dans une variable :

```
OBJ=$(SRC:.c=.o)
```

```
CC=gcc
CFLAGS=-Wall -Werror -std=c99
EXEC=hello
LDFLAGS=
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

all: $(EXEC)

hello: $(OBJ)
	$(CC) -o $@ $^ $(LDFLAGS)

%.o: %.c
	$(CC) -o $@ -c $< $(CFLAGS)
```

Stocker le résultat de commandes dans une variable

Par exemple pour faire appel à `pkg-config` :

```
CFLAGS=-Wall -Werror -std=c99 \
       $(shell pkg-config --cflags sdl2)
LDFLAGS=$(shell pkg-config --libs-only-L sdl2)
LDLIBS=$(shell pkg-config --libs-only-l sdl2)
```

Commandes silencieuses

Par défaut, `make` affiche les commandes qu'il exécute.
Dans le cas d'une commande `echo`, par exemple, cela peut faire doublon.
On peut indiquer à `make` de ne pas afficher la commande en la préfixant par `@` :

```
hello:
	@echo "Building $@..."
```

Forcer la reconstruction d'une cible

Ne devrait jamais être nécessaire...

```
make -B foo
```

Ou bien simuler une modification d'un fichier source :

```
touch foo.c
```

Pour aller plus loin

Quelques limites

- Règles spécifiques à l'environnement (OS, architecture, ...) ?
- Tester si une bibliothèque / fonction est disponible ?
- (Dés)Activer la compilation de certaines fonctionnalités du programme ?
- Complexité d'un fichier `Makefile` rapidement grandissante...

D'autres outils plus poussés

- Autotools : ensemble d'outils (`autoconf`, `automake`, ...)
- CMake
- Meson
- ...

Autotools et CMake produisent des fichiers `Makefile` !

Exercice

Reprenez votre code (ou la correction) du TP5 de PG109, notamment `cafeteria.c` et `ticket_tab.c`.

Créez un fichier `Makefile` qui construit les fichiers suivants :

- `libticket.so` à partir de `ticket_tab.c`
- `cafeteria` à partir de `cafeteria.c` en le liant à `libticket.so`

Ajoutez les règles suivantes :

- `all`, première règle, qui construit `cafeteria`
- `clean`, qui supprime tous les binaires

À vous de jouer !

Quelques liens

- https://gl.developpez.com/tutoriel/outil/makefile/
- https://renenyffenegger.ch/notes/development/make/index
- https://sed-bso.gitlabpages.inria.fr/cmake/

Déverminage

Erreurs de programmation

Les fameux *bugs*...

- Erreurs statiques : détectées à la compilation, par le compilateur (syntaxe, types, ...)
- Erreurs dynamiques : à l'exécution, comportement non désiré (mauvais comportement, voire arrêt de l'exécution, ...)

Déverminage « au `printf` »

Parsème le code de `printf` pour comprendre où se situe le bug, afficher le contenu de variables...

Avantages

- Simple et facile à mettre en œuvre (vous savez déjà le faire !)
- Flexible : on peut afficher ce qu'on veut
- Redirections possibles

Inconvénients

- Nécessite de changer le code, recompiler, réexécuter... plusieurs fois !
- Peut générer des sorties très verbeuses

Déverminage « au `printf` » en pratique

Écrire sur la sortie d'erreur pour s'assurer que les données sont immédiatement affichées :

```
fprintf(stderr, "foo = %d\n", foo);
```

On peut ensuite rediriger la sortie d'erreur :

```
./prog 2> erreur.out
```

Ou, au contraire, n'afficher que la sortie d'erreur :

```
./prog > sortie.out
./prog > /dev/null
```

Des outils plus efficaces existent (mais on utilise quand même tous aussi cette méthode...).

Préparer son programme pour des outils de déverminage

Compiler avec les options suivantes :

- `-O0` (lettre o majuscule suivie d'un zéro) : désactive les optimisations du compilateur, pour mieux faire correspondre le code assembleur au code C)
- `-g` : inclut les symboles de débogguage dans l'exécutable

Parenthèse sur les optimisations

Le compilateur peut réaliser de nombreuses optimisations :

- dérouler des boucles
- inclure directement le code des fonctions là où elles sont appelées
- réordonner des instructions
- faire de la vectorisation
- ...

On peut contrôler ces optimisations à l'aide du paramètre `-O` (toujours la lettre o en majuscule)[5] :

- de `-O0` (par défaut) à `-O3` : pas d'optimisation à optimisations maximales
- `-Os` : optimise pour réduire la taille du code compilé
- `-Ofast` : optimise pour la vitesse d'exécution du programme, quite à ne plus respecter certains standards

5. RTFM : `man gcc` (fourni par le paquet `gcc-doc` sous Debian)

GDB

GNU Debugger : programme qui exécute un autre programme, et qui permet de :

- Exécuter le programme ligne par ligne
- Afficher le contenu des variables, de la mémoire
- Afficher le code source
- Afficher la pile d'appels
- ...

Utilisation de GDB

Lancement d'un programme dans GDB :

```
gdb $programme
```

Puis lancement du programme depuis l'invite de commande de GDB :

```
run $arguments_du_programme
```

Chaque commande GDB a un raccourci : `run` $\rightarrow$ `r`

Pour recharger le programme à exécuter (après recompilation dans un autre terminal, par exemple) :

```
file $programme
```

Interruption de l'exécution

L'exécution du programme peut être interrompue de plusieurs façons :

- Une erreur dans le programme cause son arrêt (erreur de segmentation, `assert()` invalide, ...)
- utilisation du raccourci clavier `Ctrl+C`
- un point d'arrêt (*breakpoint*) est atteint

Dans tous les cas, on retrouve une invite de commande GDB.

Points d'arrêt

Indique à GDB qu'il doit s'arrêter lorsque l'exécution atteint une ligne spécifique du code source

Définir un point d'arrêt :

```
break fonction
br 32 // s'arrête à la ligne 32
br fichier.c:32
```

Astuce : touche Tab pour avoir l'autocomplétion !

Lister les points d'arrêt :

```
info break
```

Désactiver/réactiver un point d'arrêt :

```
disable $br_id
enable $br_id
```

Supprimer un point d'arrêt :

```
delete $br_id
```

Exécution interrompue : où est-on dans le programme ?

- `backtrace` (`bt`) : affiche la pile d'appels des fonctions
- `list` (`l`) : affiche quelques lignes de code source
  - Appel suivant : affiche le code qui suit
  - `l -` : affiche le code qui précède
  - `l 28,38` : affiche les lignes 28 à 38
  - `tui enable` : affiche en permanence le code (`tui disable` pour... vous avez compris ; raccourci clavier : `Ctrl+X A`)

Exécution interrompue : qu'y a-t-il dans ces variables ?

- `print (p)` : affiche le contenu d'une variable :

```
p variable
p tableau[2]
p *ptr
p structure.membre
p structure->membre
```

- `info local` : montre toutes les variables définies dans le contexte actuel et leurs valeurs
- `info args` : montre les arguments de la fonction actuelle et leurs valeurs

Exécution interrompue : remontons la pile d'appels

- `bt` : affiche la pile d'appels des fonctions

- `up` : remonte d'une fonction dans la pile d'appels
- `down` : descend d'une fonction dans la pile d'appels
- `frame $id` : passe directement au niveau `$id` de la pile d'appels, indiqué par `bt`

Exécution interrompue : continuons pas-à-pas

... ou pas pas-à-pas !

- `continue` (`c`) : reprend l'exécution du programme
- `step` (`s`) : exécute l'instruction suivante en rentrant dans les appels de fonction
- `next` (`n`) : exécute l'instruction suivante sans rentrer dans les appels de fonction
- `unroll` (`u`) : exécute jusqu'à la sortie de la boucle
- `finish` (`f`) : exécute jusqu'à la sortie de la fonction

`step $n` ou `next $n` exécute *n* fois `step` ou `next`

Astuce : Touche `Enter` sans avoir saisi de commande répéte la commande précédente.

Autres commandes

Il y en a beaucoup...

- `watch` : provoque l'interruption du programme quand une zone mémoire est modifiée
- `display` : affiche une variable automatiquement, à chaque interruption de GDB
- `tbreak` : comme `break`, mais actif qu'une seule fois
- `help $commande` : affiche l'aide de la commande
- `set variable $var = $expression` : modifie la valeur de la variable
- `print $fonction($arguments)` : appelle la fonction
- Points d'arrêt conditionnels (s'arrêter seulement après le $n^{\text{ème}}$ passage, seulement si une variable a telle valeur, ...)
- Affichage de zones mémoires, de tableaux, ...)
- ... *cf* les liens plus loin

Quitter GDB

- Commande `quit` ou `exit`
- `Ctrl+D`

Analyse d'un core dump

Sans GDB, en cas d'erreur (entre autres) de segmentation, le système peut sauvegarder la mémoire du processus dans un fichier `core`

Cette fonctionnalité peut être contrôllée par la commande `ulimit` :

```
ulimit -c # doit renvoyer autre chose que 0
ulimit -c unlimited # pour définir la taille maximale
    d'un fichier core
```

Attention, les fichiers `core` peuvent être volumineux !

Analyser un fichier `core` avec GDB :

```
gdb $programme $core
```

S'arrête sur la ligne fautive.

Autre outil de débbugage : Valgrind

Émulation du code machine, pour faire des vérifications

- Peut rendre l'exécution beaucoup plus lente
- Sérialise les programmes multithreadées

En réalité, un ensemble d'outils (à préciser avec `--tool=`) :

- `memcheck` (outil par défaut, le plus souvent utilisé) : vérifications mémoires (utilisation de mémoire non initialisée, dépassement de tableau, accès à de la mémoire déjà libérée, ...)
- `cachegrind` : analyse de l'utilisation des caches
- `callgrind` : donne les graphes d'appel des fonctions et le temps passé dans chaque fonction
- `helgrind` : analyse d'exécutions concurrentes pour les programmes multithreadés
- `massif` : analyse de l'occupation du tas

Utilisation de Valgrind

```
valgrind $programme $arguments_du_programme
```

Pour avoir plus d'informations sur les problèmes rapportés par Valgrind :

```
valgrind \
    --leak-check=full \
    --show-leak-kinds=all \
    --track-origins=yes \
    $programme $arguments_du_programme
```

Erreurs rapportées par Valgrind

- Accès à des zones mémoires interdites (en-dehors des tableaux, zone mémoire déjà libérée, ...)
- Lecture d'une zone mémoire pas encore initialisée
- ...

Rapport en fin d'exécution

Classement en différentes catégories des blocs mémoire qui n'ont pas été libérés :

- `definitively lost` : l'adresse du bloc mémoire à libérer n'est plus stockée nul part
- `indirectly lost` : l'adresse du bloc mémoire est stockée dans une zone mémoire à laquelle on n'a plus accès
- `possibly lost` : une adresse à l'intérieur du bloc mémoire est toujours stockée quelque part (dans le cas d'opération sur les pointeurs : `ptr++` par exemple)
- `still reachable` : l'adresse du bloc mémoire est encore dans une variable dans le tas (variable `static` par exemple)

Correction : rajouter les `free` manquants...

Passer le relais à GDB

Valgrind s'arrête à chaque erreur et permet de passer le relai à GDB :

```
valgrind \
    --vgdb-error=1 \
    $programme $arguments_du_programme
```

Lancer ensuite GDB dans un autre terminal en suivant les instructions de Valgrind.

$\Rightarrow$ permet d'inspecter le contexte de l'erreur plus en détails

Sanitizers

- Comme Valgrind, pour vérifier l'utilisation de la mémoire
- Surcoût plus faible qu'avec Valgrind, peut parfois détecter plus d'erreurs
- Meilleur support du multi-threading
- Instrumente le code à la compilation
- RTFM : https://github.com/google/sanitizers/wiki

```
gcc -g -O0 -fsanitize=address prog.c -o prog
export ASAN_OPTIONS=detect_stack_use_after_return=1
./prog # affichera les erreurs trouvées
```

En résumé

Choix des outils

- Une erreur de segmentation ? $\Rightarrow$ Valgrind
- GDB pour comprendre un comportement erroné du programme
- (l'un n'exclut par l'autre...)

Recommendations

(... ou *Obligations* ?)

- Tout problème rapporté par Valgrind doit impérativement être corrigé !
- Exécuter ses programmes avec Valgrind de temps en temps, même sans erreur mémoire apparente, pour s'assurer qu'il n'y a vraiment pas d'erreur
- Traiter les erreurs dans leur ordre d'apparition (comme pour les erreurs du compilateur)

Exercice

Récupérez le fichier `bugs.c`.
Comme son nom l'indique, ce programme est truffé de bugs.

Compilez le programme et débugguez-le avec GDB puis Valgrind et évenutellement les sanitizers.
Corrigez les erreurs au fur et à mesure[6].

À vous de jouer !

6. Le code est relativement court est simple, toutes les erreurs peuvent se corriger juste en lisant le code. Mais jouez le jeu, le but est de manipuler les outils !

Quelques liens

- https://sed-bso.gitlabpages.inria.fr/formations/gdb_2019.html
- https://dept-info.labri.fr/~thibault/gdb.html.fr
- https://sed-bso.gitlabpages.inria.fr/formations/valgrind.html
- https://gitlab.inria.fr/lcirrott/horror
- Beej's Quick Guide to GDB https://beej.us/guide/bggdb/

Gestion de version avec Git

Développement à plusieurs

(« *plusieurs* » pouvant être soi-même et le soi-même du futur...)

Déjà vu en PG109...

- Commenter le code
- Définir et respecter des conventions de codage

Mais comment faire pour...

- Communiquer les changements dans le code ?
  - Envoyer tout le code en indiquant qu'il s'agit de la nouvelle version ?
  - Envoyer un mail expliquant les modifications à apporter ?
  - Autre approche ?
- Garder un historique des modifications ?
  - Pour savoir qui a fait quoi quand et pourquoi
  - Pour pouvoir revenir à une version antérieure du code

diff

Programme qui compare deux fichiers (textuels) et affiche les différences :

```
diff a.txt b.txt
```

Options :

- `-y` : affiche les fichiers côte-à-côte
- `--suppress-common-lines` : n'affiche pas les lignes identiques
- `-r` : compare récursivement deux dossiers

Le logiciel Meld est un `diff` avec une interface graphique

patch

On peut sauvegarder les différences trouvées par `diff` dans un fichier qu'on nomme alors un *patch* :

```
diff -rupN original new > modification.patch
```

... on envoie le fichier `modification.patch` à la personne qui doit appliquer la modification...

La modification est appliquée avec la commande `patch` :

```
cd original
patch < ../modification.patch
```

Génération, envoi et application de modifications dans le code : ✓
Gestion de l'historique des modifications : ✗

Gestionnaire de sources

Un gestionnaire de sources (traduction approximative de VCS : *version control system*) permet de :

- conserver l'historique des modifications apportées à des fichiers
- travailler à plusieurs sur du code source

Chaque modification enregistrée est appelée un *commit*

Quelques gestionnaires de version :

- Git : le plus connu et plus utilisé, vu dans la suite de ce cours
- SVN (Subversion) : toujours utilisé par de vieux projets
- CVS (*Concurrent Versions System*) : encore plus ancien que SVN
- Mercurial
- Bazaar
- Microsoft a aussi le sien

Ces logiciels reposent sur les principes de `diff` et `patch`.

Git

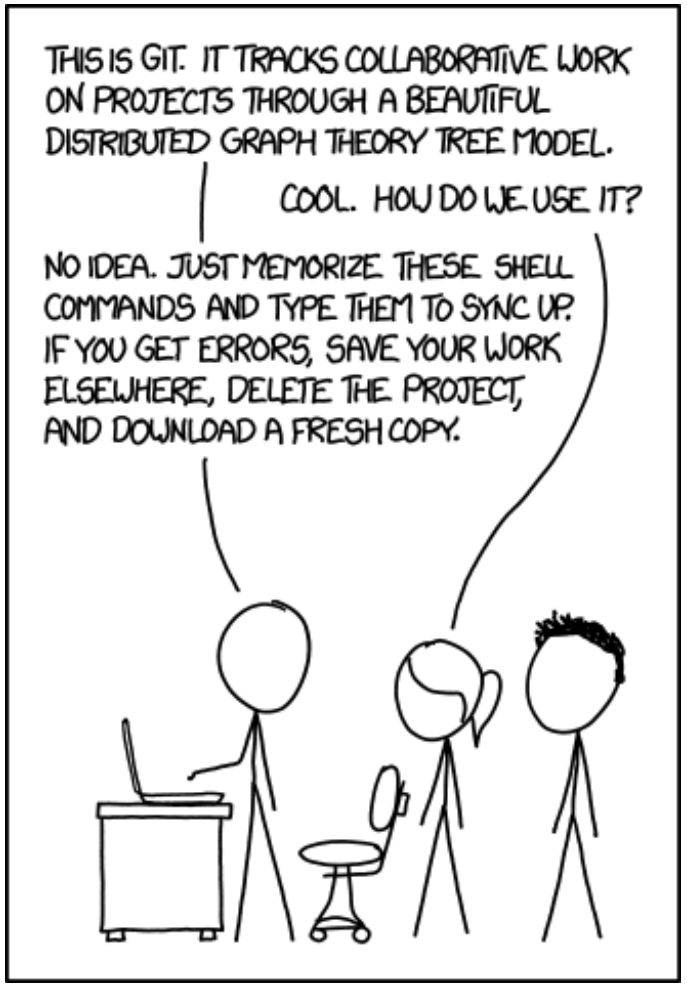
- Créé en 2005 par Linus Torvalds pour remplacer le gestionnaire de version utilisé alors pour le développement de Linux

Manipulation

- En ligne de commande : `git`
- Avec une interface graphique : `git gui` et `gitk`, `gitg`, Ungit, SourceTree, et beaucoup d'autres...
- Intégré dans votre éditeur

Objectifs de ce module Git

- La maîtrise de Git peut être considérée comme difficile
- **Objectifs :**
  - Connaître les concepts de base de Git
  - Savoir utiliser les fonctionnalités de base de Git



https://xkcd.com/1597/

Dépôts

Un code source est versionné par Git au sein d'un *dépôt* (*repository*).

Un dépôt Git est un dossier contenant n'importe quoi, mais surtout un dossier `.git` où Git stocke toutes les informations concernant ce dépôt (l'historique des modifications apportées aux fichiers du dossier, ...)

Créer un dépôt :

```
mkdir mon-super-projet
cd mon-super-projet
git init
```

Il est possible de *cloner* un dépôt Git déjà existant ailleurs :

```
git clone https://login@forge.fr/adresse/depot
cd depot
```

On récupère alors le contenu et l'historique du dépôt distant.

Configuration

Deux niveaux de configuration :

- Configuration globale : stockée dans `$HOME/.gitconfig`
- Configuration par dépôt : stockée dans `.git/config`, a la priorité sur la configuration globale

Manipulation de la configuration :

```
git config --global user.name "Gérard MENVUÇA"
git config --global user.email gerard.menvuca@ecole.fr

# En étant dans un dépôt :
git config user.email gerard.menvuca.du.33@perso.fr
```

Pour obtenir la valeur d'un paramètre :

```
git config user.name
```

Obtenir de l'aide

```
git $cmd --help
man git-$cmd
```

Exemple pour la commande `clone` :

```
git clone --help
man git-clone
```

Documentation officielle : https://git-scm.com

Exercice

Chaque étudiant du binôme :

1. Clonez depuis Thor le dépôt que vous allez utiliser pour le projet
   - Utilisez l'adresse HTTPS du dépôt, et non SSH
2. Si ça n'a pas été déjà fait en IF110 : configurez Git en suivant les instructions de la section *Configurer Git* de la page `https://thor.enseirb-matmeca.fr/ruby/docs/repository/git`
3. Vérifiez que vos nom et email sont bien renseignés dans Git

À vous de jouer !

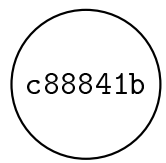
Commit

Un *commit* désigne deux choses :

- une modification de fichier(s)
- l'état de l'ensemble des fichiers une fois cette modification appliquée

Un commit est constitué de :

- la modification de fichier(s) [7]
- un message
- une date
- un auteur (nom et adresse mail)
- un commit parent (qui est le commit précédent)



Tous ces attributs vont permettre d'en construire un dernier : l'identifiant du commit, qui correspond à un hachage `SHA1`.

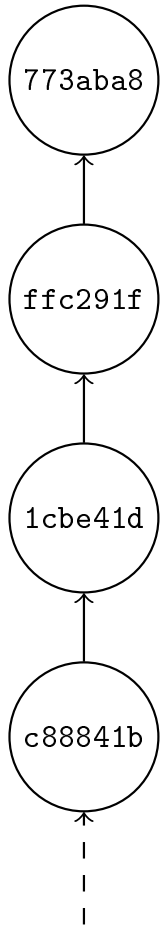
Si un de ces attributs est changé, l'identifiant du commit change également.

7. Bien qu'on pourrait croire que c'est le `diff` des modifications qui est stocké, Git garde toutes les versions de chaque fichier.

Historique

Chaque commit ayant un commit parent, on peut former un graphe composé des commits.

On peut alors se balader dans ce graphe pour consulter l'historique des modifications, revenir en arrière, annuler des modifications, *etc*.



Génération, envoi et application de modifications dans le code : ✗
Gestion de l'historique des modifications : ✓

Les trois états possibles des fichiers et des modifications

`Modifications qui seront validées` `Changes to be committed`

Modifications marquées comme à committer

`Modifications qui ne seront pas validées` `Changes not staged for commit`

Modifications pas marquées comme à committer

`Fichiers non suivis` `Untracked files`

Fichiers non suivis par Git

La commande dont il faut abuser

`git status`

Permet de connaître l'état de votre dépôt, des fichiers qui le composent et des modifications en cours.

Quels fichiers versionner ?

- Git peut versionner n'importe quel type de fichiers (textes, images, binaires, vidéos, sons, ...)
- Règle : **on ne versionne pas ce qui peut être généré à partir de ce qui est dans le dépôt** (fichiers `.o`, exécutables, bibliothèques, fichiers PDF, ...)
- On évite de versionner des fichiers volumineux (pensez à celui qui fera `git clone`). En cas de besoin : regardez du côté de Git LFS.
- En général : y réfléchir à deux fois avant de commiter un fichier qui n'est pas textuel
- On ne versionne pas les fichiers spécifiques à votre environnement (fichiers `.swp` de Vim, fichiers `#truc#` de Emacs, fichiers de votre IDE, ...)
- Git ne versionne pas les dossiers ! Astuce pour forcer la présence d'un dossier vide : y placer un fichier vide `.gitkeep` et versionner ce fichier.

Exclure des fichiers

Pour que Git ignore certains fichiers, notamment dans `git status`, on peut créer (et versionner) un fichier `.gitignore` :

```
*.pdf # tous les fichiers PDFs
*.o # tous les fichiers .o
mon_programme
```

Ce fichier concerne le dossier où il se trouve et ses sous-dossiers.

Quand committer quoi ?

*Un commit réalise **un** changement et il le fait **bien**.*

Un changement

Le changement répond à un unique problème.

- Exemple correct : correction d'une fuite mémoire
- Exemple incorrect : correction d'une fuite mémoire et ajout d'une option pour faire pousser des radis

Pensez au cas où vous voudriez annuler seulement l'ajout de l'option...

Un commit ne doit pas être vu comme une sauvegarde d'une modification inachevée !

Il le fait **bien**

Une fois le commit récupéré, le problème est complétement résolu.

- Exemple correct : ajout d'une option pour faire pousser des radis
- Exemple incorrect : début du `if` pour faire tester si une option pour faire pousser des radis est donnée
- Exemple incorrect extrême : faire un commit qui casse (consciemment) le code. Ou alors il est clairement indiqué que le commit est non-fonctionnel !

Message de commit

- Comme pour le style de code : toujours garder le même style (langue, format, ...)
- Langue : français, anglais, ... à vous de choisir
- Phrase sans sujet :
  - *Ajout d'une option écrire à l'envers*
  - *Add option to write backwards*
- On doit pouvoir dire :
  - *En appliquant ce commit, je vais avoir...*
  - *If I apply this commit, it will...*
- Dans certains projets, préfixé du type de modification (`fix`, `feat`, `doc`, ...)
- Idéalement moins de 50 caractères

Messages incorrects

- *tmp*
- *fix*
- *Écriture à l'envers*

Faire un commit

1. Inspection de la situation :

```
git status
```

Modificatior

Modificatior

Fichiers nor

8. On peut modifier l'éditeur utilisé avec `git config core.editor`

Faire un commit

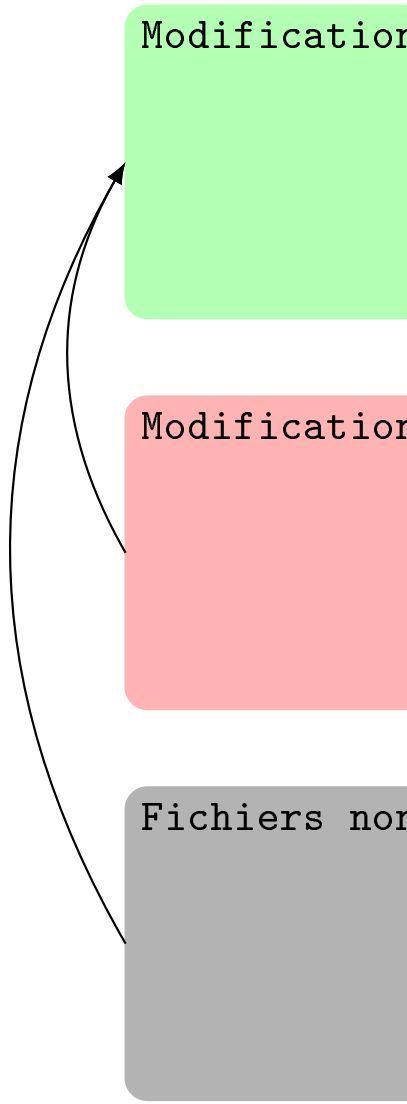
1. Inspection de la situation :

```
git status
```

2. Ajouter les modifications à committer :

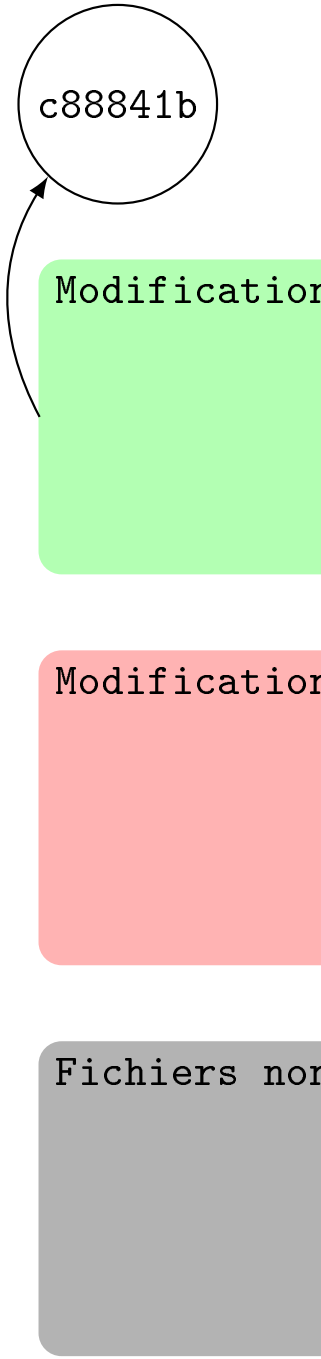
```
git add fichier.txt
```

Dans ce cas, toutes les modifications apportées au fichier `fichier.txt` feront partie du commit.



8. On peut modifier l'éditeur utilisé avec `git config core.editor`

Faire un commit



1. Inspection de la situation :

```
git status
```

2. Ajouter les modifications à committer :

```
git add fichier.txt
```

   Dans ce cas, toutes les modifications apportées au fichier `fichier.txt` feront partie du commit.

3. Committer :

```
git commit
```

4. Saisir le message de commit dans l'éditeur qui s'est ouvert [8]
5. Sauvegarder et quitter
6. Et voilà !

8. On peut modifier l'éditeur utilisé avec `git config core.editor`

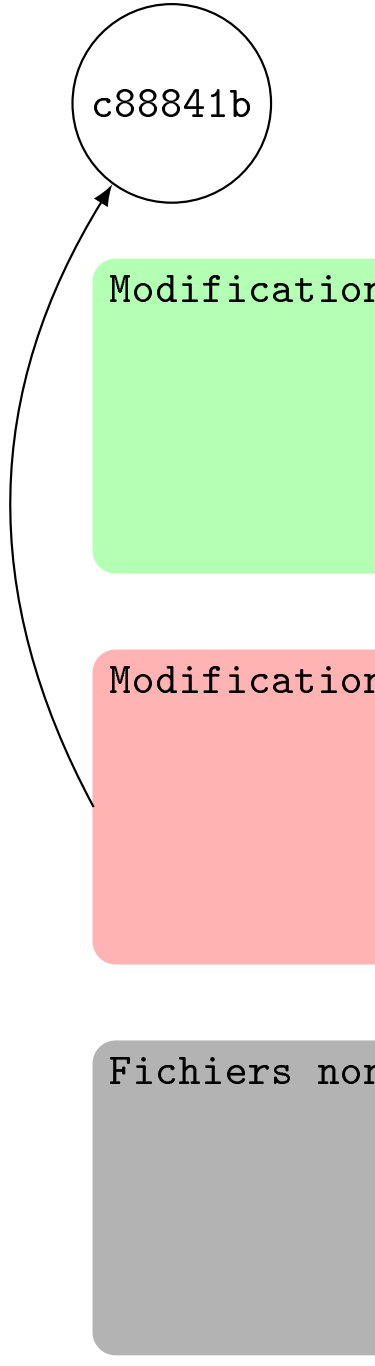
Quelques raccourcis pour committer

Pour inclure dans le commit les modifications de tous les fichiers déjà suivis, sans passer par `git add` :

```
git commit -a
```

Pour saisir le message de commit sans passer par l'éditeur :

```
git commit -m "Message"
```



Voir les modifications pas encore committées

Modifications qui seront validées — Changes to be committed

```
git diff --staged [$fichier]
```

Modifications qui ne seront pas validées — Changes not staged for commit

```
git diff [$fichier]
```

Fichiers non suivis — Untracked files

Exercice

Chaque étudiant du binôme :

1. Vérifiez que votre nom et adresse mail sont bien configurés. Si ce n'est pas le cas, *cf* exercice précedent
2. Si ce n'est pas déjà fait, configurez Git pour qu'il utilise l'éditeur `nano`
3. Modifiez le fichier `README.md` en ajoutant votre nom sur une nouvelle ligne à la fin du fichier
4. Regardez les modifications avec `git diff`
5. Committez cette modification

À vous de jouer !

Consulter l'historique

Afficher tous les commits (le plus récent en premier) :

```
git log
```

Afficher tous les commits qui ont modifié un fichier en particulier :

```
git log $fichier
```

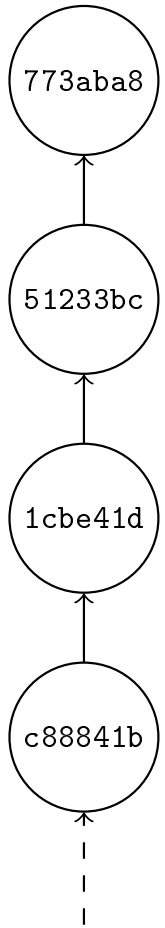
Voir un commit spécifique :

```
git show $commit_id
```

Astuce : pas besoin d'utiliser l'ID du commit en entier, seulement assez de caractères pour identifier un unique commit

Désigner des commits

- Le commit ID en entier : `51233bc38e0f3575765e8459f2c3a4d61c5e0370`
- **Astuce** : pas besoin d'utiliser le commit ID en entier, seulement assez de premiers caractères pour identifier un unique commit : `51233bc`
- Le commit parent : `51233bc~`
- Le commit grand-parent : `51233bc~~`



Comparer des commits

Les commits désignant aussi des états de l'ensemble des fichiers, on peut obtenir les différences dans les fichiers entre différents états :

- Entre l'état actuel est un commit particulier :

```
git diff 51233bc
```

  Peut être vu comme toutes les différences entre l'état des fichiers au commit 51233bc et l'état actuel

- Entre deux commits différents :

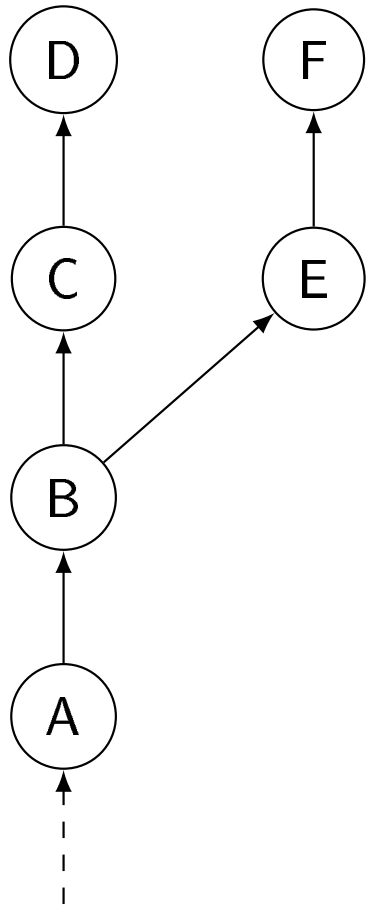
```
git diff 51233bc~~..51233bc
```

  Peut être vu comme la somme des modifications apportées par les commits 51233bc~ et 51233bc

Question : quelle est la différence entre `git diff 51233bc~~..51233bc` et `git diff 51233bc..51233bc~~` ?

Branches

- Concept important des systèmes de gestion de versions
- Une branche est une suite de commits formant un historique
- On peut avoir plusieurs branches en parallèle qui évoluent indépendemment
- Deux branches partagent un historique de commits commun, puis divergent à partir d'un commit [9]
- Une branche peut être fusionnée dans une autre



9. On peut aussi (rarement) avoir des branches orphelines.

Cas d'utilisation des branches

Pouvoir avoir différentes modifications dans le code indépendantes, simultanément.

Exemple de cas d'utilisation :

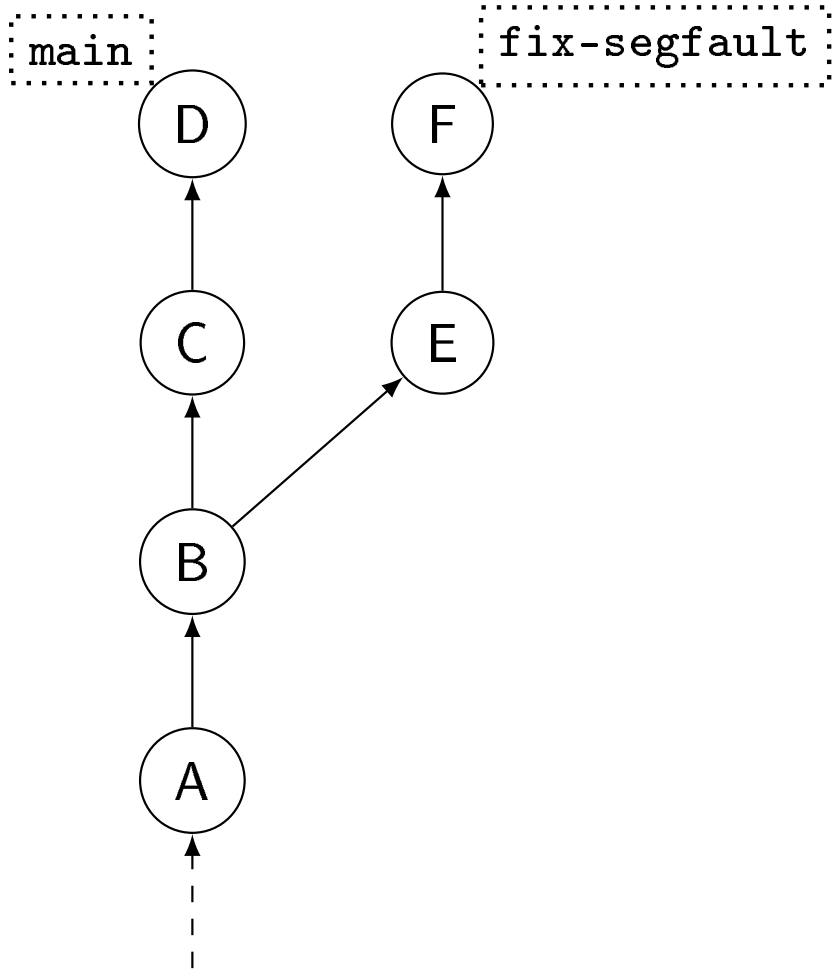
- Une branche principale contient le code approuvé
- Une branche pour chaque correction de bug
- Une branche pour chaque nouvelle fonctionnalité

Permet aux personnes intéressées / concernées de tester la correction de bug ou la nouvelle fonctionnalité, sans perturber le code déjà stable et et les autres développements en cours.

En général, à terme, chaque branche est fusionnée dans la branche principale.

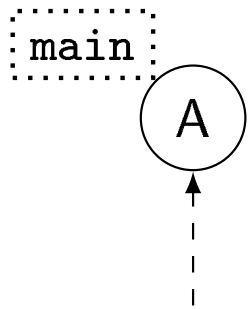
Nom de branches

- Pour désigner plus facilement les différentes branches, chaque branche a un nom
- Le nom d'une branche n'est qu'un alias de son dernier identifiant de commit
- Branche par défaut : la seule que vous récupérez quand vous clonez un dépôt
  - Nommée `master`, `main`, `dev`, `trunk`, ...



Utilisation de branches (1)

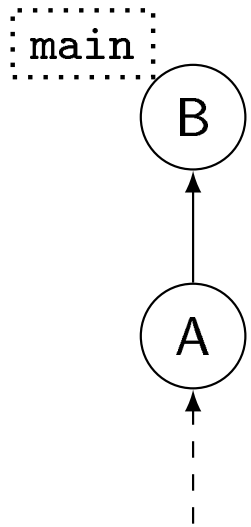
État initial :



Utilisation de branches (2)

Création d'un commit :

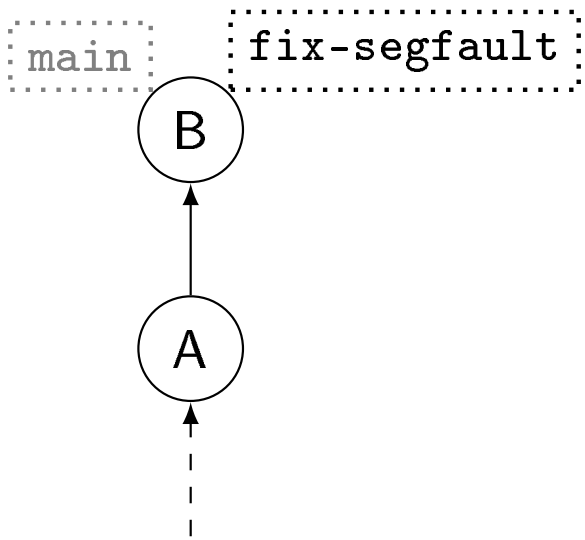
```
git commit
```



Utilisation de branches (3)

Création d'une branche et basculement sur cette nouvelle branche :

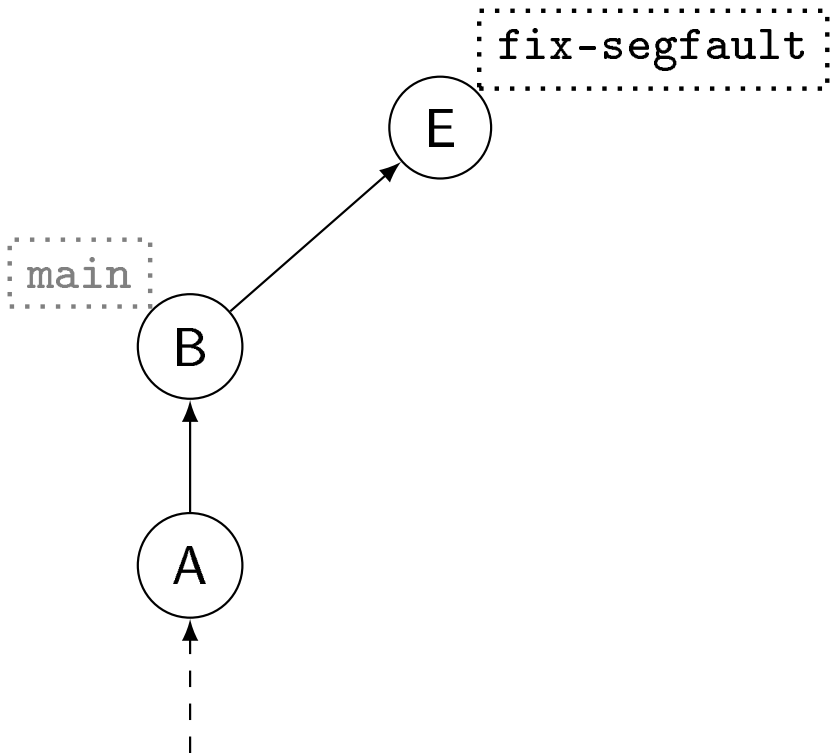
```
git switch -c fix-segfault
```



Utilisation de branches (4)

Création d'un commit :

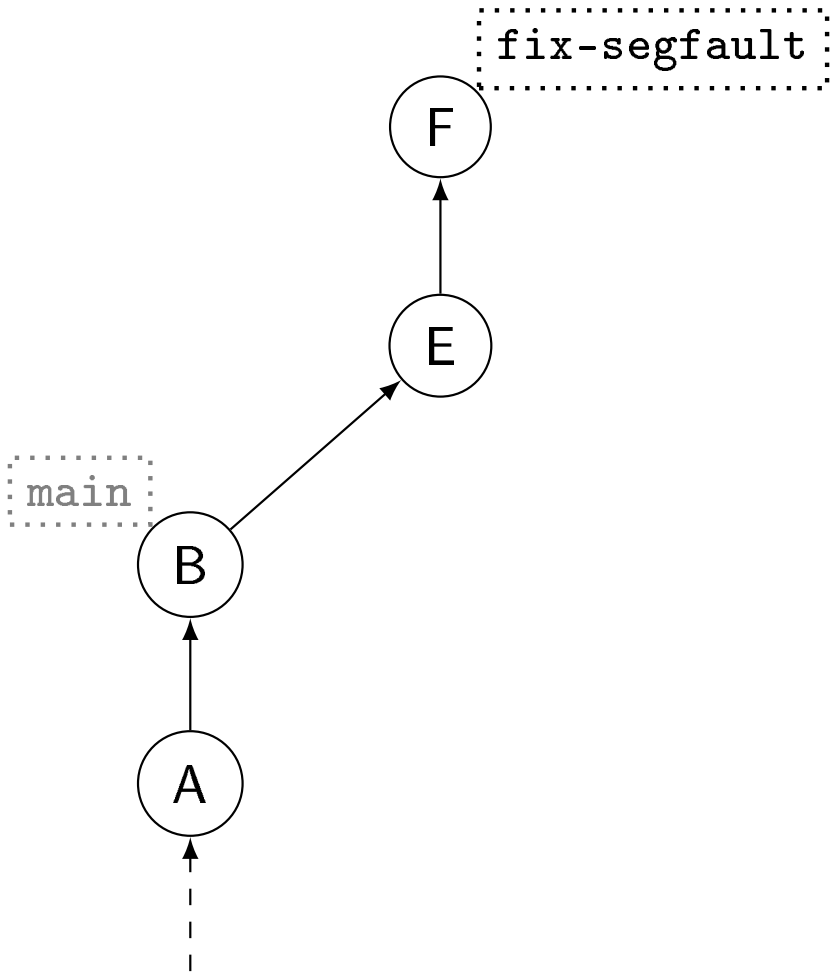
```
git commit
```



Utilisation de branches (5)

Création d'un commit :

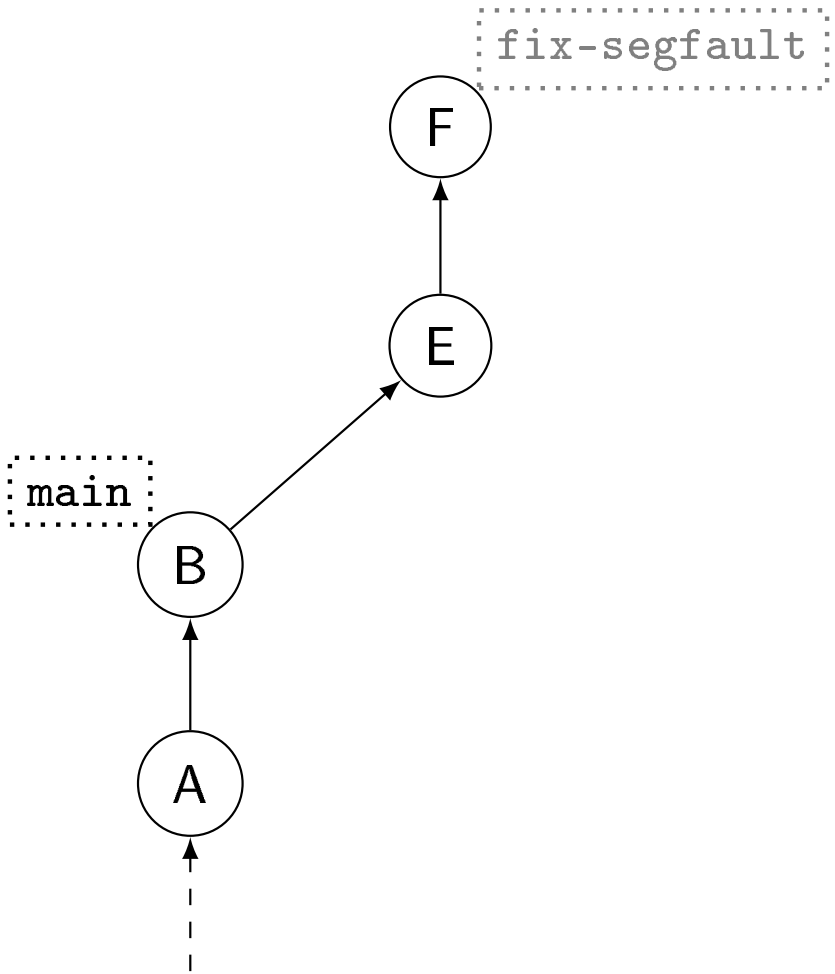
```
git commit
```



Utilisation de branches (6)

Basculement sur la branche `main` :

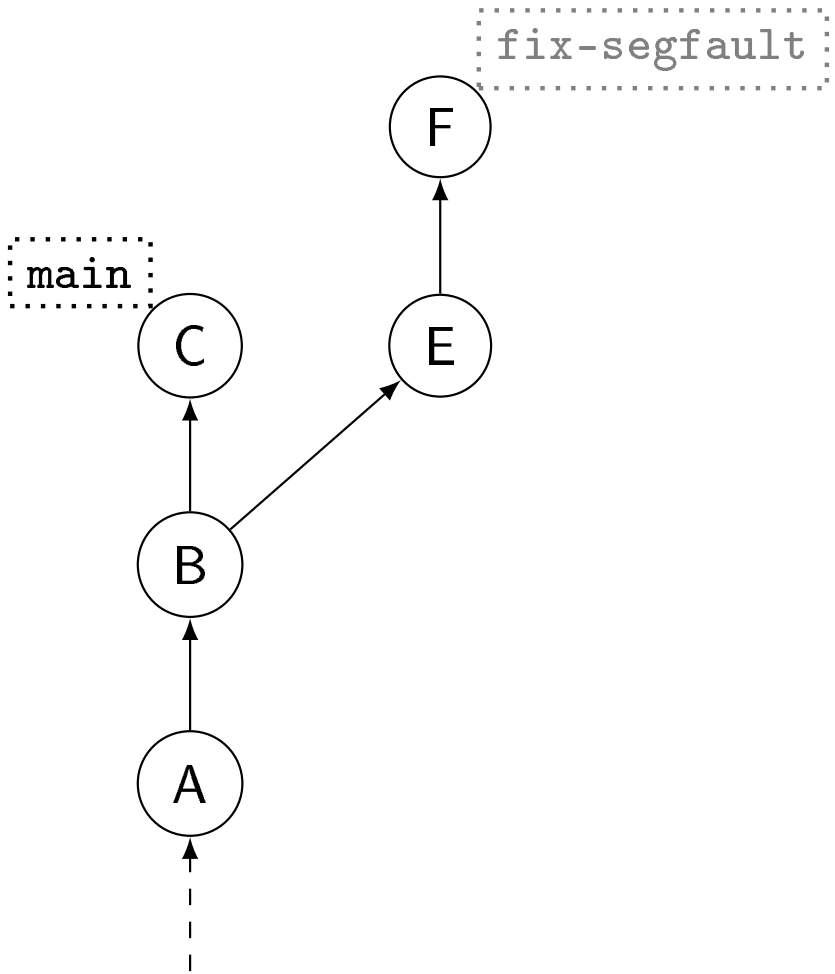
```
git switch main
```



Utilisation de branches (7)

Création d'un commit :

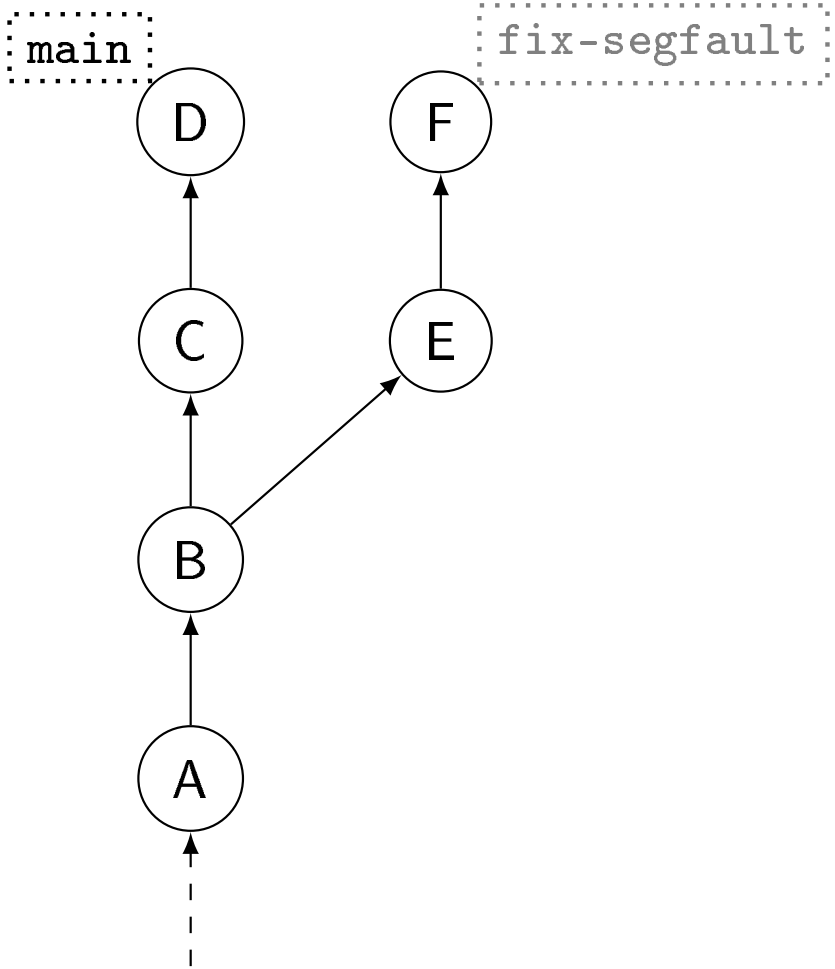
```
git commit
```



Utilisation de branches (8)

Création d'un commit :

```
git commit
```



Fusion de branches

Récupère les modifications d'une branche dans une autre

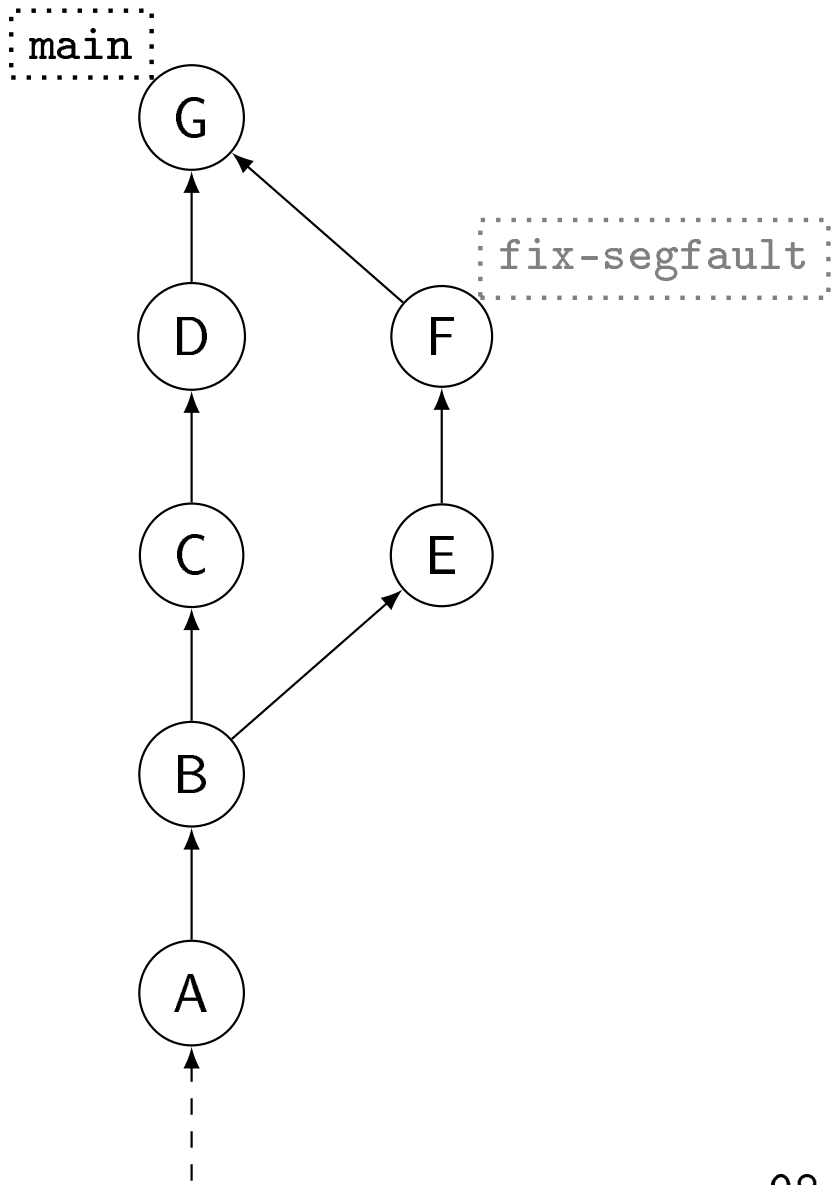
1. Se placer sur la branche qui va recevoir les modifications de l'autre branche :

```
git switch main
```

2. Fusionner :

```
git merge fix-segfault
```

- Crée un **commit de fusion** avec deux parents
- On peut toujours continuer d'ajouter des commits à la branche `fix-segfault` !



Manipulation de branches

- Visualiser le graphe des commits :

```
git log --oneline --graph --decorate=full
```

- Lister les branches existantes :

```
git branch
```

- Supprimer une branche :

```
git branch -d $branche
```

Conflits

Dans le cas d'une fusion, si des modifications venant des deux branches portent sur les mêmes portions de fichiers, et que Git ne sait pas choisir quelle modification conserver, se produit un **conflit**

- La fusion est interrompue
- Git vous indique les fichiers qui contiennent des conflits (on peut les retrouver avec `git status`)
- À vous de corriger manuellement les conflits en éditant les fichiers concernés

Résolution de conflits

1. Les conflits sont délimités ainsi dans les fichiers :

```
<<<<<<
Contenu du fichier dans la branche foo
======
Contenu du fichier dans la branche bar
>>>>>>
```

2. Vous devez supprimer les délimiteurs du conflit et adapter le contenu du fichier tel que vous voulez qu'il soit
3. Sauvegardez le fichier, marquez le fichier comme n'ayant plus de conflit :

```
git add
```

4. Une fois qu'il n'y a plus de conflit dans aucun fichier, terminez le commit de fusion :

```
git commit
```

`git status` vous indique toutes les commandes à saisir !

Dépôts distants

- Un dépôt distant (celui que vous avez cloné, par exemple) est un dépôt Git (presque) comme un autre (contient des commits, des branches, les fichiers versionnés, ...)
- Il sert de point de synchronisation entre plusieurs personnes
- Le dépôt par défaut (celui que vous avez cloné) est nommé `origin`
- Un dépôt local peut avoir plusieurs dépôts distants

Lister les dépôts distants connus (avec leurs adresse grâce à `-v`) :

```
git remote -v
```

Ajouter un dépôt distant :

```
git remote add $nom $adresse
```

Informations des dépôts distants

Récupérer les informations (nouveaux commits, nouvelles branches, ...) d'un dépôt distant :

```
git fetch [$depot]
```

Par défaut, c'est le dépôt `origin` qui est interrogé

Dépôt distants et branches

Les branches locales peuvent être configurées pour *suivre* des branches de dépôts distants :

- Indique que la branche locale et la branche distante ont vocation à être identiques
- Par défaut, la branche obtenue lors du `clone` suit la branche correspondante sur le dépôt distant

Lister toutes les branches, locales et des dépôts distants :

```
git branch --all
```

On désigne les branches distantes avec le format `$depot/$branche`. Par exemple :

```
git log origin/main
```

Mettre à jour sa branche locale

Dans la branche locale courante, récupérer les commits de la branche `$branche` du dépôt distant `$depot` :

```
git pull $depot $branche
```

Sans paramètre, `git pull` récupère les commits de la branche suivie

`git pull` est un raccourci pour :

```
git fetch $depot
git merge $depot/$branch
```

Envoyer ses commits à un dépôt distant

Pour envoyer les nouveaux commits de sa branche locale courante à la branche distante correspondante :

```
git push
```

Pour une branche créée localement, lors du premier `push`, il faut indiquer dans quelle dépôt doit être envoyée la branche, pour désigner la branche distante à suivre :

```
git push --set-upstream $depot
```

Génération, envoi et application de modifications dans le code : ✓
Gestion de l'historique des modifications : ✓

Récupérer une branche d'un dépôt distant

1. Mettre à jour les informations locales du dépôt distant, pour avoir connaissance de la branche distante :

```
git fetch $depot
```

2. Basculer sur la branche :

```
git switch $branche
```

où `$branche` est le nom de la branche distante. Ce sera aussi le nom de branche locale, qui suivra automatiquement la branche distante.

Exercice

1. Étudiant A : poussez votre commit vers le dépôt distant
2. Étudiant B : récupérez le commit poussé par l'étudiant A
3. Étudiant B : (sans doute un conflit à régler)
4. Étudiant B : poussez votre commit vers le dépôt distant
5. Étudiant A : récupérez le commit poussé par l'étudiant B

Chaque étudiant doit maintenant avoir le même contenu de `README.md` dans son dépôt local

À vous de jouer !

Se balader dans l'historique

Se positionner sur un ancien commit :

```
git checkout 1cbe41d
```

Les fichiers seront restaurés à l'état après avoir créé le commit `1cbe41d`.
Les commits enfants existent toujours !

Revenir au dernière commit, la tête de la branche :

```
git checkout $branche
```

Annuler des modifications pas encore committées

Modifications qui seront validées — Changes to be committed

```
git restore --staged $fichier
```

Modifications qui ne seront pas validées — Changes not staged for commit

```
git restore $fichier
```

Fichiers non suivis — Untracked files

Impossible, fichiers non suivis par Git !

Créer ses propres raccourcis

Avec la commande `git config` ou directement en éditant le fichier `$HOME/.gitconfig` :

```
[alias]
  logg = log --oneline --graph --decorate=full
  st = status
  grepi = grep -n
  rv = remote -v
```

Permet ainsi de faire :

```
git logg
git st
git grepi
git rv
```

Encore beaucoup d'autres choses...

- Réécrire l'historique avec `rebase` (dangereux si on ne sait pas ce qu'on fait)
- Générer une archive avec `archive` (prend en compte le `.gitignore` !)
- Recupérer un commit d'une branche dans une autre branche avec `cherry-pick`
- Chercher un commit par dichotomie avec `bisect`
- Chercher dans les fichiers du dépôt avec `grep`
- *Commit hooks* pour exécuter des scripts avant/après un commit, un push, ...
- Sous-modules Git pour inclure des dépôts Git dans un dépôt Git
- Faire un commit qui en annule un autre avec `revert`
- Identifier des commits spécifiques par des étiquettes avec `tag`
- Voir qui a modifié les lignes d'un fichier avec `blame`
- Sauvegarder temporairement des modifications sans faire un commit avec `stash`
- ...

En résumé : tentative de schéma

En résumé : votre utilisation de Git en PG110

- `git clone`
- `git status`
- `git diff`
- `git status`
- `git add`
- `git status`
- `git commit`
- `git status`
- `git log`
- `git pull`
- `git status`
- `git push`
- `git status`

Forges logicielles

- `github.com`, `gitlab.com` ou autres instances GitLab, BitBucket, instances Gogs, instances Gitea et même `thor.enseirb-matmeca.fr`
- Interfaces web pour interagir avec des dépôts distants
- Permet de visualiser toutes les informations des dépôts Git (fichiers, commits, branches, ...)
- Suivant la forge utilisées, peut avoir plus ou moins de fonctionnalités :
  - Système de tickets (*issues*) : pour rapporter des problèmes, des demandes de fonctionnalités, ...
  - *Forks* (traduction affreuse : *divergeances*) : copie d'un dépôt Git dans son espace personnel de la forge. On peut alors travailler sur notre dépôt comme on veut.
  - *Pull* ou *Merge Request* : on demande au dépôt original d'un fork d'intégrer nos commits venant d'une branche de notre fork

Quelques liens

- https://thor.enseirb-matmeca.fr/ruby/docs/repository/git
- https://sed-bso.gitlabpages.inria.fr/formations/git-basics-Pau-02-05-2023/git-basics-Pau-02-05-2023.html
- https://blog.stephane-robert.info/docs/developper/version/git/introduction/

Autres outils et pratiques

Tests

La compilation du code ne suffit pas à s'assurer que le comportement du programme obtenu est correct.
$\Rightarrow$ écriture de codes qui vont tester le comportement du programme :

- Permet d'augmenter la robustesse du code
- Permet de détecter qu'un bug corrigé ne réapparaîtra pas suite à d'autres modifications (*tests de non-régression*)
- Mais : la présence de tests ne garantit pas l'absence de bug !

Par convention, les tests peuvent être compilés et exécutés avec `make check` ou `make test`

Types de tests

Différents types de tests :

- Tests **unitaires** pour tester le bon comportement d'une fonction
- Tests **fonctionnels** pour tester la bonne interaction entre différentes fonctions d'un module
- Tests **d'intégration** pour tester la bonne interaction entre différents modules
- Tests **de recette** pour tester le bon fonctionnement général de l'application

Test Driven Development (TDD)

Méthodologie de développement qui consiste à commencer par écrire les tests, puis le code testé :

1. Écriture d'un test
2. Le test ne passe pas
3. Écrire le code testé de façon minimale
4. Le test passe

Peut aussi s'appliquer à la correction d'un bug :

1. Écriture d'un test qui échoue à cause du bug
2. Correction du bug
3. Le test passe

Avantages du TDD :

- Force à écrire le test
- Force à réfléchir à ce qui doit être testé et, par extension, à la façon d'écrire le code testé

Couverture de code

Permet de savoir combien de fois chaque ligne de code a été exécutée

```
gcc -coverage prog.c -o prog
./prog
gcov prog.c
less prog.c.gcov
```

Les lignes exécutées 0 fois sont appelées *code mort* :

- si le code n'est pas exécuté car inutile, il doit être supprimé
- si l'exécution des programme des tests n'est pas passée par des lignes de code, c'est qu'il manque peut-être des tests
  - Dans le cas des tests, on cherche à maximiser le *taux de couverture*
  - Même si ce n'est parfois pas la métrique la plus pertinente
  - Exécuter du code ne signifie pas qu'il est bien testé !

Intégration continue

Dans l'idéal, on voudrait s'assurer régulièrement (à chaque commit, à chaque push, ...) que :

- le code compile (`make`)
- les tests passent (`make test`)

avec potentiellement les contraintes suivantes :

- le code compile et les tests passent sur des environnements différents (différents sytèmes d'exploitation, différents compilateurs, différentes versions d'OS, de compilateurs, de bibliothèques, ...)
- la compilation et l'exécution des tests prend du temps

Deux solutions :

- Exécution de tous ces essais manuellement
- Demander (gentiment) à un logiciel de le faire automatiquement pour nous
  $\Rightarrow$ c'est **l'intégration continue**

Systèmes d'intégration continue

- Jenkins
- Travis CI
- GitLab CI
- GitHub Actions
- ...

Tous ces sytèmes se configurent avec un fichier qui indique ce qu'il faut faire (compiler, lancer les tests, générer la documentation, ...) à quel moment (généralement lors d'un `push`).

Pair programming

Développement à deux :

- Chacun de son côté
- ou **Pair Programming** :
  - Méthodologie de programmation
  - Deux personnes utilisent le même ordinateur, regardent le même écran
  - Une personne a le clavier pendant un temps défini, puis le passe à l'autre
  - Permet de réflechir à deux sur le code écrit et donc d'être plus efficace

Rubber duck debugging

Vous le faites souvent sans le savoir avec vos enseignants...

- Ayez une peluche ou un canard en plastique sur votre bureau.
- En cas de bug ou de code pas clair : lisez et expliquez le code à votre peluche
- Vous allez trouver votre bug !