Cours outils de développement

David SALLÉ (david.salle@ensemblescolaire-niort.com)

Version du document : v0.5
Date : 15/11/2021

Table des matières

1 – Introduction

Ce document présente les principaux outils de développement informatique et notamment en langage C/C++ :

Outil	***Description***	***Outils***
Compilateur	Permet de transformer un code source en langage C/C++ par exemple en langage machine directement exécutable	`gcc/g++`
Compilation projet	Permet de gérer la compilation d'un projet contenant plusieurs fichiers sources	`make`
Débogueur	Permet de mettre au point un programme, de le déboguer avec une exécution en mode pas à pas par exemple ou en inspectant directement le contenu des variables en mémoire	`gdb`
Profileur Testeur performances	Permet d'analyser l'exécution d'un programme (temps CPU par fonction, consommation mémoire, ...) afin de pourvoir par la suite l'optimiser	`gprof`
Tests unitaires	Permet de tester unitairement chaque méthode de chaque classe et ainsi valider le bon fonctionnement des briques de base	`Catch2`
Tests bugs	Permet de repérer des bugs potentiels via une analyse statique du code source	`cppchecker`
Convention de codage	Permet de vérifier que le code source est conforme à une convention de codage (charte qualité du code source)	`cpplint`
Documentation automatique	Permet de générer une documentation technique à partir des commentaires du code source	`Doxygen`
Métriques & statistiques	Permet de quantifier de nombreux aspects d'un code source (nombre de lignes, % commentaires, nombres d'instructions, complexité, nombre de ligne par fonctions...)	`SourceMonitor`
Gestion de versions	Permet de gérer les différentes versions des fichiers d'un projet informatique et notamment en équipe	`git`

Si les outils utilisés pour illustrer le cours font partie de la suite GNU, les notions et concepts associés restent globalement transposables à tous les autres outils du marché.

A noter que les principaux IDE du marché intègrent plus ou moins l'ensemble de ces outils dans une IHM graphique ergonomique.

2 – Compilateur (gcc/g++)

2.1 – Chaîne de compilation (native)

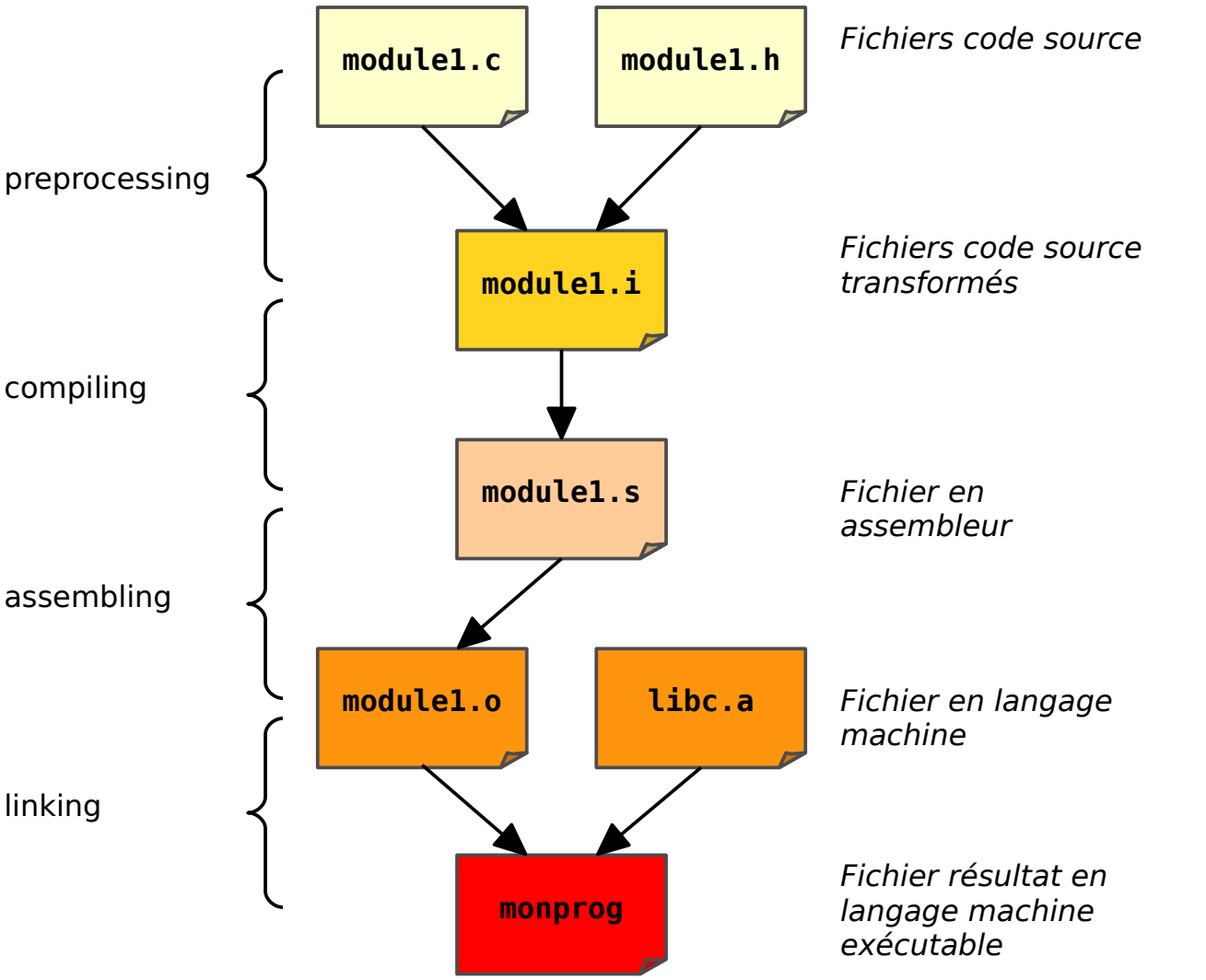
La compilation d'un code source pour obtenir un exécutable directement exploitable sur une machine se décompose en plusieurs sous-étapes :

1. **preprocessing** : le pré-processeur analyse et transforme le code source
2. **compiling** : compilation du code transformé en langage assembleur
3. **assembling** : conversion du langage assembleur en langage machine
4. **linking** : édition des liens (liaison entre tous les binaires et les librairies)

Toutes ces étapes sont transparentes pour le développeur. Ainsi la commande

```
$ gcc module1.c -o monprog
```

... produit la chaîne de compilation suivante



2.1.1 – Preprocessing

Lors de cette étape le pré-processeur effectue plusieurs transformations sur le code source original :

- suppression des commentaires (// … et /* … */)
- inclusion des fichiers *.h dans le fichier intermédiaire *.i (#include)
- traitement de toutes les directives de précompilations commençant par # (#define, #ifdef, #ifndef, #endif, #if, …)

Le pré-processeur procède ensuite à une première vérification de la syntaxe du code. Si une erreur intervient le processus de compilation s'arrête. Un avertissement n'est lui pas bloquant.

2.1.2 – Compiling

Lors de cette étape le fichier code source transformé va être compilé en langage assembleur. Un seconde vérification de la syntaxe plus poussée est également effectuée avec les mêmes conséquences que précédemment.

2.1.3 – Assembling

Le code assembleur généré lors de l'étape précédente va être transformé en langage machine directement compréhensible par le processeur de l'ordinateur.

2.1.4 – Edition des liens

Lors de cette dernière étape, l'éditeur de lien aura deux missions :

- assembler les différents modules binaires générés par la compilation
- « copier/coller » le code binaire des fonctions stockées dans des librairies externes

Une fois ces derniers assemblages binaires effectués, le programme exécutable final est enfin prêt.

A noter qu'un programme exécutable pour Linux ne l'est pas du tout pour Windows, ou pour MacOS et inversement. Chaque système d'exploitation dispose de son prore format exécutable, et il faudra recompiler un même code source avec différentes versions de compilateur pour obtenir un exécutable pour chaque plateforme.

Linux	***Windows***	***Mac OS***
ELF	PE	Mach-O

2.2 – Utilisation et principales options

La commande gcc peut s'utiliser directement en console, ou bien via un Makefile ou bien encore via un IDE (exemple : Qt, Code::Blocks, Eclipse CDT, ...)

2.2.1 - Choix de l'étape

Option	***Description***
`-c`	Etape de compilation simple générant un objet binaire à partir d'un code source. Ici on générera un fichier binaire module1.o **`gcc -c module1.c`**
`-o <output>`	Etape d'édition des liens consistant à regrouper tous les fichiers binaires en un exécutable dont le nom est à fournir (<output>). Si <output> n'est pas fourni, l'exécutable se nommera a.out par défaut **`gcc -o monprog module1.o module2.o module3.o`**

2.2.2 – Options pour l'étape de compilation

Option	***Description***
`-Wall`	Le compilateur est très exigeant et relèvera beaucoup de d'avertissements
`-O<level>`	Précise le niveau d'optimisation du code machine généré. Valeur entre 1 et 6, généralement 2 **`gcc -c -O2 module1.c`**
`-g`	Compilation en mode « debug » permettant d'ajouter de nombreuses informations pour le débogage avec gdb. Le binaire est plus gros en taille et plus lent. Par défaut si l'option n'est pas présente, la compilation est en mode « release » **`gcc -c -g module1.c`**
`-pg`	Compilation en mode « debug » comme précédemment mais permettant d'analyser l'exécutable avec gprof
`-I<path>`	Permet d'ajouter un répertoire (<path>) où trouver les fichiers *.h en plus du répertoire système (/usr/include) utilisé par défaut **`gcc -c -I/home/toto/test module1.c`**
`-D<macro>`	Permet d'activer une macro qui sera utilisée par le pré-processeur
`-std=<standard>`	Permet de choisir le standard C/C++ : • c89 : langage C ANSI (1ière version normalisée) • c99 : langage C (2nde version) • c++0x : nouvelle norme C++

Option	Description
	• c++11 : nouvelle norme C++ **gcc -c -std=c99 module1.c** **g++ -c -std=c++11 module1.cpp**
-v	Affiche beaucoup d'informations lors de la compilation il devient « versatile » (bavard)

2.2.3 - Option pour l'étape d'édition des liens

Option	Description
-l<libname>	Indique qu'il faut lier une librairie externe au(x) binaire(s) lors de l'édition des liens. Le répertoire par défaut où sont stockées les librairies est /usr/lib. Ici la librairie se nomme unelib **gcc -o monprog module1.o module2.o -lunelib**
-L<path>	Si la librairie externe à lier n'est pas dans le répertoire par défaut /usr/lib, alors il est possible de préciser un autre répertoire où la trouver **gcc -o monprog module1.o module2.o -L/home/toto/ici -lmalib**
-shared	Indique qu'on souhaite créer une librairie dynamique plutôt qu'un exécutable

2.3 – Création de librairie

Parfois on ne souhaite par forcément générer un programme exécutable, mais simplement créer une librairie pour être utilisée dans un autre programme.

2.3.1 – Statique VS dynamique

Il existe 2 types de librairies :

- **statique**
- **dynamique** (ou partagée)

Selon les systèmes d'exploitation elles portent une extension particulière et sont stockées dans différents répertoires.

	Linux	Windows	Mac OS X
Statique	libXXX.a	XXX.lib	XXX.a
Dynamique	libXXX.so	XXX.dll	XXX.dylib
Répertoire(s)	/lib /usr/lib	Windows\System	/System/Library

Quelles différences entre une librairie statique/dynamique ?

	Statique	***Dynamique***
Avantages	=> L'exécutable contient tout ce qu'il faut pour fonctionner (si distribution)	=> Programme plus léger => Pas obligé de tout recompiler si changement dans la librairie (si prototypes identiques) => Librairies chargées une seule fois en mémoire pour tous les programmes => Possibilité de charger une librairie que quand le besoin est là
Inconvénients	=> Programme plus lourd => Obligation de tout recompiler si changement dans la librairie	=> L'exécutable nécessite impérativement que toutes les librairies soient présentes pour s'exécuter (si distribution) => Temps de lancement plus long car liaison à l'exécution

2.3.2 – Création/utilisation d'une librairie statique

Ma librairie sera constituée d'un ensemble de fonctions (sans main) réparties dans les fichiers codes sources suivants :

- **module1.c**
- **module1.h**
- **module2.c**
- **module2.h**

J'intégrerai ensuite cette librairie dans un fichier **main.c**

Etape #1 : compilation des sources en binaires

```
$ gcc -c module1.c -o module1.o
$ gcc -c module2.c -o module2.o
```

Etape #2 : création de la librairie statique à partir des binaires

```
$ ar -rcsv libmalibtest.a module1.o module2.o
```

Etape #3 : utilisation de la librairie statique

```
$ gcc main.c -o maintestlib -L. -lmalibtest
```

A noter que si la librairie malibtest est liée statiquement au programme, les autres fonctions des autres librairies (printf, ...) le sont dynamiquement.

Pour lier statiquement toutes les librairies utilisées par un programme il faut alors utiliser l'option -static de gcc

```
$ gcc -static monfichier.c -o monprog
```

2.3.3 – Création/utilisation d'une librairie dynamique

Ma librairie sera constituée d'un ensemble de fonctions (sans main) réparties dans les fichiers codes sources suivants :

- **module1.c**
- **module1.h**
- **module2.c**
- **module2.h**

J'intégrerai ensuite cette librairie dans un fichier **main.c**

Etape #1 : compilation des sources en binaires

```
$ gcc -c module1.c -o module1.o
$ gcc -c module2.c -o module2.o
```

Etape #2 : création de la librairie dynamique à partir des binaires

```
$ gcc -o libmalibtest.so -shared module1.o module2.o
```

Etape #3 : utilisation de la librairie dynamique

```
$ gcc main.c -o maintestlib -L. -lmalibtest
```

Etape #4 : modification de la variable LD_LIBRARY_PATH

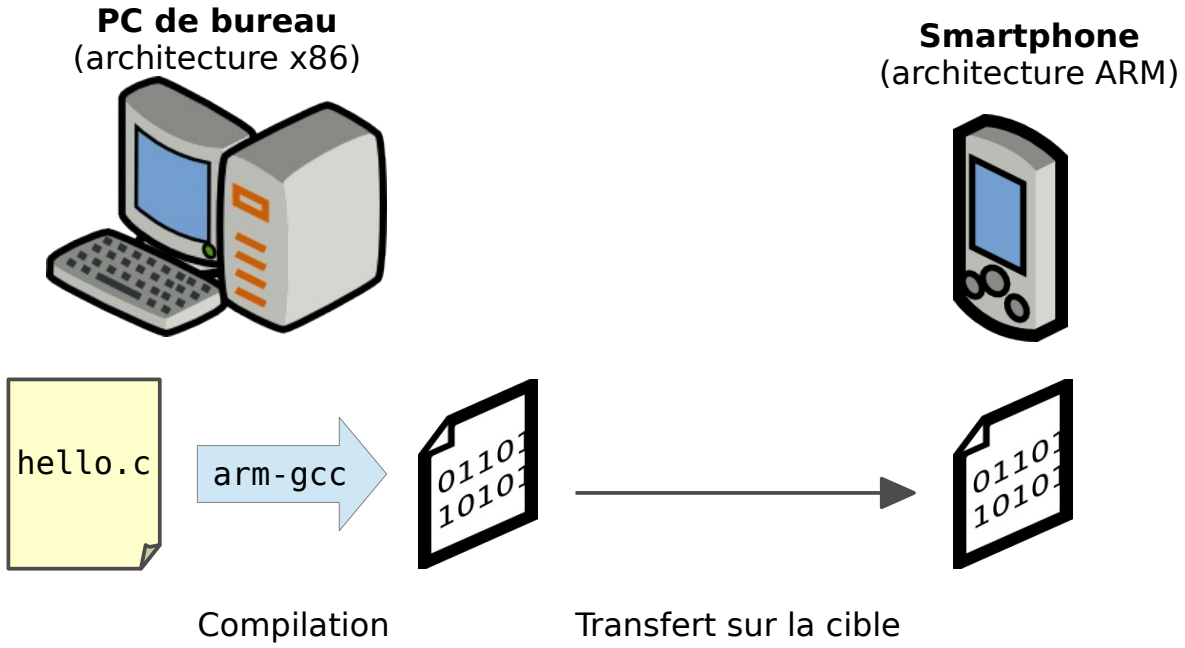
```
$ export LD_LIBRARY_PATH=/home/jbegood/testlib:$LD_LIBRARY_PATH
```

Sans cette modification, au lancement du programme testlib une erreur surviendrait indiquant que la librairie dynamique n'a pas pu être trouvée.

2.4 – Chaîne de compilation croisée

La compilation croisée consiste à compiler un programme pour une plateforme cible différente de celle utilisée pour la compilation, la plateforme hôte.

Cette technique est très utilisée dans le monde du développement embarqué et mobile où les systèmes cibles ne permettent pas toujours d'installer un environnement de développement.



Il est également parfois possible d'installer un débogueur sur la plateforme cible pour assurer un débogage à distance (réseau, RS232, ...)

3 – Compilation projet (make)

3.1 - Généralités

Un Makefile est un fichier utilisé par le programme make qui regroupe toute une séries d'instructions pour assurer la compilation d'un projet par exemple.

Ce fichier Makefile peut être écrit à la main ou bien généré automatiquement par un programme comme qmake, cmake, automake, ...

Il est consitué de règles de la forme :

```
cible:    dépendance(s)
          commande(s)
```

Pour l'exécuter, il suffit dans le répertoire de travail où se situe le fichier Makefile de taper la commande suivante :

```
$ make
```

3.2 – Exemples

Dans les exemples qui vont suivre on cherchera à compiler un projet constitué des fichiers codes sources suivants :

- module1.c
- module1.h
- module2.c
- module2.h
- main.c

On cherchera à produire un exécutable nommé « monprog »

3.2.1 – Makefile simple

Voici un premier Makefile très basique mais suffisant pour compiler le projet :

```
# Règles de construction
monprog:        module1.o module2.o main.o
                gcc -o monprog module1.o module2.o main.o

module1.o:      module1.c
                gcc -c module1.c -o module1.o

module2.o:      module2.c
                gcc -c module2.c -o module2.o

main.o:         main.c
                gcc -c main.c -o main.o
```

A noter que la cible principale générant l'exécutable doit être en premier.

3.2.2 – Ajout des cibles standards

Voici un second Makefile dans lequel on a ajouté les cibles standards suivantes :

- **all** : permet de tout compiler et doit donc contenir comme dépendances les exécutables à générer
- **clean** : permet de supprimer tous les fichiers intermédiaire *.o
- **mrproper** : permet via une dépendance à clean de supprimer tous les fichiers générés lors d'une compilation (*.o + exécutable)

```
# Règles de construction
all:            monprog

monprog:        module1.o module2.o main.o
                gcc -o monprog module1.o module2.o main.o

module1.o:      module1.c
                gcc -c module1.c -o module1.o

module2.o:      module2.c
                gcc -c module2.c -o module2.o

main.o:         main.c
                gcc -c main.c -o main.o

# Règles de nettoyage
clean:
                rm -f *.o

mrproper:       clean
                rm -f monprog
```

3.2.3 – Ajout de variables

Dans cette troisième version, nous ajoutons quelques variables qui vont permettre de rendre le Makefile plus adaptable et plus facile à travailler.

Elles se déclarent sous la forme MAVAR=UNEVALEUR et s'utilise $(MAVAR)

- **CC** : désigne le compilateur utilisé
- **CFLAGS** : désigne les options du compilateur lors de l'étape de compilation
- **LDFLAGS** : désigne les options du compilateur lors de l'étape d'édition des liens
- **EXEC** : désigne l'exécutable à produire

```
# Variables de compilation
CC=gcc
CFLAGS=-Wall
LDFLAGS=-L. -lm
EXEC=monprog

# Règles de construction
all:            $(EXEC)

$(EXEC):        module1.o module2.o main.o
          $(CC) -o $(EXEC) module1.o module2.o main.o $(LDFLAGS)

module1.o:      module1.c
                $(CC) -c module1.c -o module1.o $(CFLAGS)

module2.o:      module2.c
                $(CC) -c module2.c -o module2.o $(CFLAGS)

main.o:         main.c
                $(CC) -c main.c -o main.o $(CFLAGS)

# Règles de nettoyage
clean:
                rm -f *.o

mrproper:       clean
                rm -f $(EXEC)
```

3.2.4 – Variables internes et règles d'inférence

Il existe d'autres raffinements possibles pour les fichiers Makefile dont nous n'aborderons pas la mise en pratique car elles ont tendance à rendre le fichier Makefile moins lisible.

Les variables internes :

- $@ : nom de la cible
- $< : nom de la première dépendance
- $^ : liste des dépendances
- $? : liste des dépendances plus récentes que la cible
- $* : nom d'un fichier sans son suffixe

Exemple :

```
$(EXEC):        module1.o module2.o main.o
                $(CC) -o $@ $^ $(LDFLAGS)

module1.o:      module1.c
                $(CC) -c $< -o $@ $(CFLAGS)
```

Les règles d'inférence :

Elles permettent de créer des règles génériques en utilisant le caractère %

Exemple :

La règle suivante ...

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

... permet de remplacer les 3 règles :

```
module1.o:      module1.c
                $(CC) -c module1.c -o module1.o $(CFLAGS)

module2.o:      module2.c
                $(CC) -c module2.c -o module2.o $(CFLAGS)

main.o:         main.c
                $(CC) -c main.c -o main.o $(CFLAGS)
```

4 – Débogueur (gdb)

Le débogueur est un outil très important pour assurer la mise au point et la correction bugs. Il permet d'exécuter un programme un peu comme on pourrait regarder un film sur une console de montage ...

- en mode pas à pas
- en lecture normale avec un point d'arrêt prédéfini (breakpoint)
- en inspectant chaque variable
- en inspectant la totalité de la mémoire
- ...

Ce document présentera gdb en console, mais les fonctionnalités sont identiques dans les débogueurs des IDE.

4.1 – Préparation du programme

Avant d'être déboguer, il faut compiler le programme en mode « debug » avec l'option -g dans gcc :

```
$ gcc -g module1.c -o monprog
```

On peut ensuite lancer l'exécution du programme avec le débogueur ainsi :

```
$ gdb ./monprog
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
...
Reading symbols from ./test_boucle...done.
(gdb)
```

On retrouve alors le promt de gdb permettant de saisir des commandes

4.2 – Commandes gdb

4.2.1 - Commandes générales

Commande	***Description***	***Exemple***
`run`	Démarrer l'exécution du programme	`run`
`kill`	Arrêter définitivement le programme	`kill`
`quit`	Quitter le débogueur	`quit`

`help`	Afficher de l'aide sur les commandes gdb	`help`
		`help list`
		`help print`

4.2.2 – Commandes pour gérer les points d'arrêt

Commande	***Description***	***Exemple***
`breakpoint`	Créer un point d'arrêt sur une ligne ou sur une fonction	`break 10` `break mafonction`
`continue`	Continuer l'exécution d'un programme après un breakpoint	`continue`
`disable`	Désactiver un breakpoint	`disable 1`
`enable`	Activer un breakpoint	`enable 1`
`delete`	Supprimer un breakpoint	`delete 1`
`info`	Afficher des informations sur les breakpoints	`info breakpoint`
`watch`	Surveiller une variable. L'exécution stoppe à chaque changement d'état	`watch mavariable`

4.2.3 – Commandes pour le mode pas à pas

Commande	***Description***	***Exemple***
`step`	Exécuter l'instruction suivante (ou plusieurs) en entrant dans les fonctions, sous-fonctions, ...	`step` `step 5`
`next`	Exécuter l'instruction suivante (ou plusieurs) sans entrer dans les fonctions, sous-fonctions, ...	`next` `next 3`

4.2.4 – Commandes pour l'affichage et les variables

Commande	***Description***	***Exemple***

`print`	Examiner ponctuellement l'état d'une variable (x=hexa, t=binaire, c=caractères, a=adresse, f=float)	`print mavariable`
		`print /x mavariable`
		`print /t mavariable`
		`print /c mavariable`
		`print /f mavariable`
		`print /a &mavariable`

`display`	Afficher automatiquement l'état d'une variable	`display mavariable`
`x`	Afficher le contenu de la mémoire. Formatage avec /NFU : => N : nombre d'octets à afficher => F : x, t, c ou a (voir print) => U : groupement (b=1, h=2, w=4)	`x &mavariable` `x /4xb mavariable`
`list`	Afficher le code source	`list`

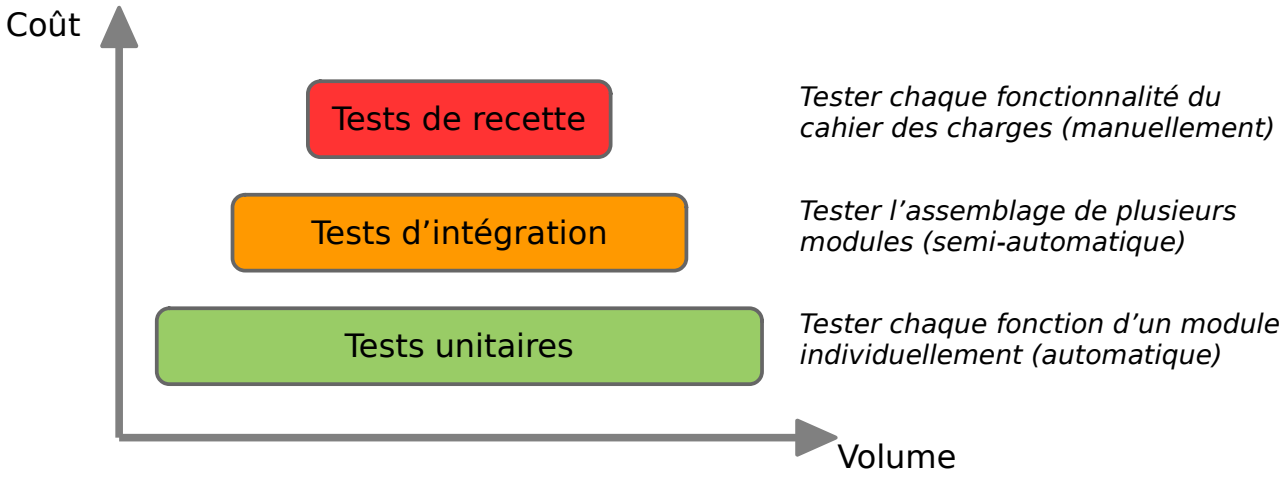
Commande	***Description***	***Exemple***
		`list 15` `list mafonction`
`disas`	Afficher le langage assembleur	`disas`
`info`	Afficher des informations diverses	`info registers`
`set`	Changer la valeur d'une variable	`set variable x=5`

5 – Tests

Les tests occupent une place très importante dans le monde du développement car c'est au travers eux qu'on peut valider la conformité d'un logiciel par rapport à un cahier des charges.

5.1 – Pyramide des tests

Les tests peuvent se classer en 3 grandes catégories selon le volume et l'impact sur un projet (complexité, coût de résolution d'un bug).



Les bonnes pratiques :

- effectuer des tests unitaires au plus tôt pour déceler un corriger un maximum de bugs potentiels
- intégration continue du code pour éviter les mauvaises surprises en fin de projet

5.2 – Tests unitaires (Catch2)

Les tests unitaires consiste donc à tester chaque fonction d'un module individuellement en recréant un environnement d'exécution. Comme ces tests peuvent vite s'avérer très nombreux, on utilise fréquemment des outils pour automatiser ces tests. Globalement, cela permet :

- déceler des bugs potentiels
- de multiplier les scenarii d'utilisation
- de réduire l'impact d'une erreur humaine
- de tester rapidement toute une application

A noter que les tests unitaires peuvent également concerner la mise en œuvre de matériels. On peut utiliser alors une fiche pour consigner les résultats.

De nombreux outils permettent d'effectuer des tests unitaires dans la plupart des langages. Nous aborderons ici **Catch2** qui est à la fois simple d'installation et d'utilisation.

Après avoir téléchargé le fichier catch.hpp dans le dossier contenant le code source à tester, il suffit de créer un fichier de tests unitaires, le compiler et l'exécuter. Exemple de fichier de tests unitaires :

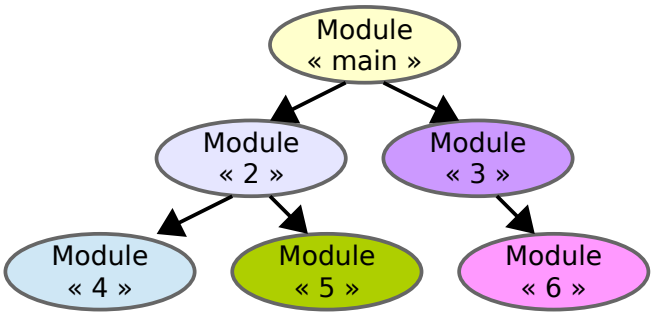
```
// Précise à la librairie de se charger de créer le main()
#define CATCH_CONFIG_MAIN

// La librairie Catch2 permettant de faire des tests unitaires
#include "catch.hpp"

// La librairie Chat avec la classe à tester
#include "chat.h"

// Tests unitaires des méthodes de la classe
TEST_CASE("Méthodes de la classe Chat", "[chat]") {

    // Chats de tests
    Chat c1();
    chat c2(2, 4.1);
    Chat c3(1, 2.8);

    SECTION("Tests des constructeurs") {
        REQUIRE( c1.getPoids() == 0.0 );
        REQUIRE( c1.getAge() == 0 );
        REQUIRE( c2.getPoids() == Approx(4.1).epsilon(0.01) );
        REQUIRE( c2.getAge() == 2);
    }

    SECTION("Tests des valeurs négatives") {
        REQUIRE_THROWS( c3.setPoids(-2.8) );
        REQUIRE_THROWS( c3.setAge(-1) );
    }
}
```

Exemple de résultat :

```
main_test.cpp:21: passed: c1.getPoids() == 0 for: 0 == 0
main_test.cpp:22: failed: c1.getAge() == -9 for: -9 == 0
main_test.cpp:27: passed: c2.getPoids()Age() == 2 for: 4 == 2
main_test.cpp:33: failed: c2.getAge() == -89 for: -89 == 0
...
Failed all 1 test cases, failed 7 assertions.
```

5.3 – Tests d'intégration

Les tests d'intégration consiste à assembler plusieurs modules contenant chacun plusieurs fonctions afin de voir si ils travaillent bien ensemble.



Plusieurs stratégies envisageables :

Stratégie	***Description***
Top-down	On part du main et on ajoute progressivement les autres modules
Bottom-Up	On part des modules de bas-niveau qu'on assemble progressivement jusqu'au module « main »
Sandwich	Mélange entre les 2 stratégies précédentes
Big-bang	On assemble tous les modules d'un coup

Quelle que soit la stratégie, il est souvent nécessaire de créer des **modules de simulation** (stub) qui ont les mêmes fonctions que les vrais mais qui produisent un résultat contrôlé. Cela permet ainsi de tester l'intégration avec un module qui n'est pas encore terminé par exemple.

Bonne pratique : intégration continue

5.4 – Tests de recette

Les recettes ou tests de validation fonctionnelle consiste à tester les fonctionnalités du cahier des charges. Ces tests sont souvent manuels.

On les consigne dans un cahier de recette accompagné d'un procès verbal.

Il est également possible d'utiliser des outils comme **Cucumber** et la syntaxe **Gherkin** qui accompagne les tests :

```
GIVEN Je navigue sur la page d'authentification
WHEN Je saisis mon identifiant dans le champ « Login »
AND Je saisis mon mot de passe dans le champ « Password »
AND Je clique sur le bouton « Connect »
THEN J'accède à la page d'accueil avec mon nom en haut à droite
```

5.5 – Tests de performance (gprof)

5.5.1 – Présentation de gprof

Le profilage des programmes consiste à étudier en direct ou a posteriori l'exécution d'un programme. Divers paramètres sont étudiés, en vue ensuite de localiser les points à améliorer dans le programme (algorithme, fonctions coûteuse, ...) :

- **temps d'exécution** (global, de chaque fonction, sous-fonction, ...)
- **nombre d'appel** de chaque fonction, sous-fonction

5.5.2 - Utilisation

La première étape consiste à compiler son programme avec l'option -pg comme suit :

```
$ gcc -pg module1.c -o monprog
```

Ensuite il faut exécuter le programme pendant un certains temps et surtout sortir proprement. Un fichier gmon.out sera alors généré.

```
$ ./monprog
```

Enfin on peut générer un rapport de profilage avec la commande suivante :

```
$ gprof ./monprog > rapport.txt
```

Il suffit ensuite d'afficher le rapport généré avec cat ou more

```
$ cat ./rapport.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 76.42      4.18     4.18 455179558    0.01     0.01  fct1
 23.40      5.46     1.28   100000    12.80    54.60  fct2
  0.18      5.47     0.01                             main
  0.00      5.47     0.00     9594     0.00     0.00  fct3
...
```

6 – Gestion du code source

6.1 – Convention de codage

Pourquoi une convention de codage ? Les raisons sont multiples et nous allons essayer de les appréhender. Les conventions de codage sont un des outils utilisés par les entreprises pour améliorer la qualité de leurs produits et l'efficacité de leurs équipes de travail.

Quelques avantages, liste non exhaustive :

- faciliter le travail en équipe en uniformisant les pratiques de codage de chacun
- faciliter la maintenance du projet informatique, donc du code source (relecture, organisation, correction de bugs...)
- faciliter la génération automatique de la documentation du code

Quelques exemples concerts issus de grands noms de l'industrie informatique

- Google C++ Style Guide
- Mozilla Coding Style
- GCC C++ Coding Convention
- ...

6.2 – Vérification (cpplint)

Des outils de vérification de la bonne application d'une convention de codage existe, un peu sur le principe des « validators » du W3C (HTML, CSS)

- **KWStyle**
- **cxxchecker**
- **cpplint** (vérifie la convention de codage C++ chez Google)

Exemple avec **cpplint** qui est un simple script Python à copier/coller dans le dossier de code source à analyser :

```
$ ./cpplint.py --filter=-whitespace –extensions=h,cpp *
mon_fichier.cpp:10: Include directory when naming .h file
mon_fichier.h:9: No copyright message found
...
Total errors found: 6
```

6.3 – Documentation automatique (doxygen)

Des outils comme Doxygen, permettent de générer automatiquement une documentation à partir d'un code source, à condition que ce dernier respecte quelques règles comme dans l'exemple ci-dessous.

Exemple d'en-tête d'un fichier source, ici le fichier exemple.cpp :

```
/**
 * @file    exemple.c
 * @brief   Une description brève d'une ligne maximum
 * @author  Johnny BEGOOD
 * @date    1er janvier 1970
 * @version 0.1
 *
 * Le reste des commentaires, sur plusieurs lignes
 * par exemple une ici
 */
// ... Le reste du code ici ...
```

Exemple d'en-tête de classe :

```
/**
 * @brief   Class de démonstration
 * @class   Demonstration
 */

class Demonstration
{
    // Intérieur de la classe
};
```

Exemple d'en-tête de fonction/méthode :

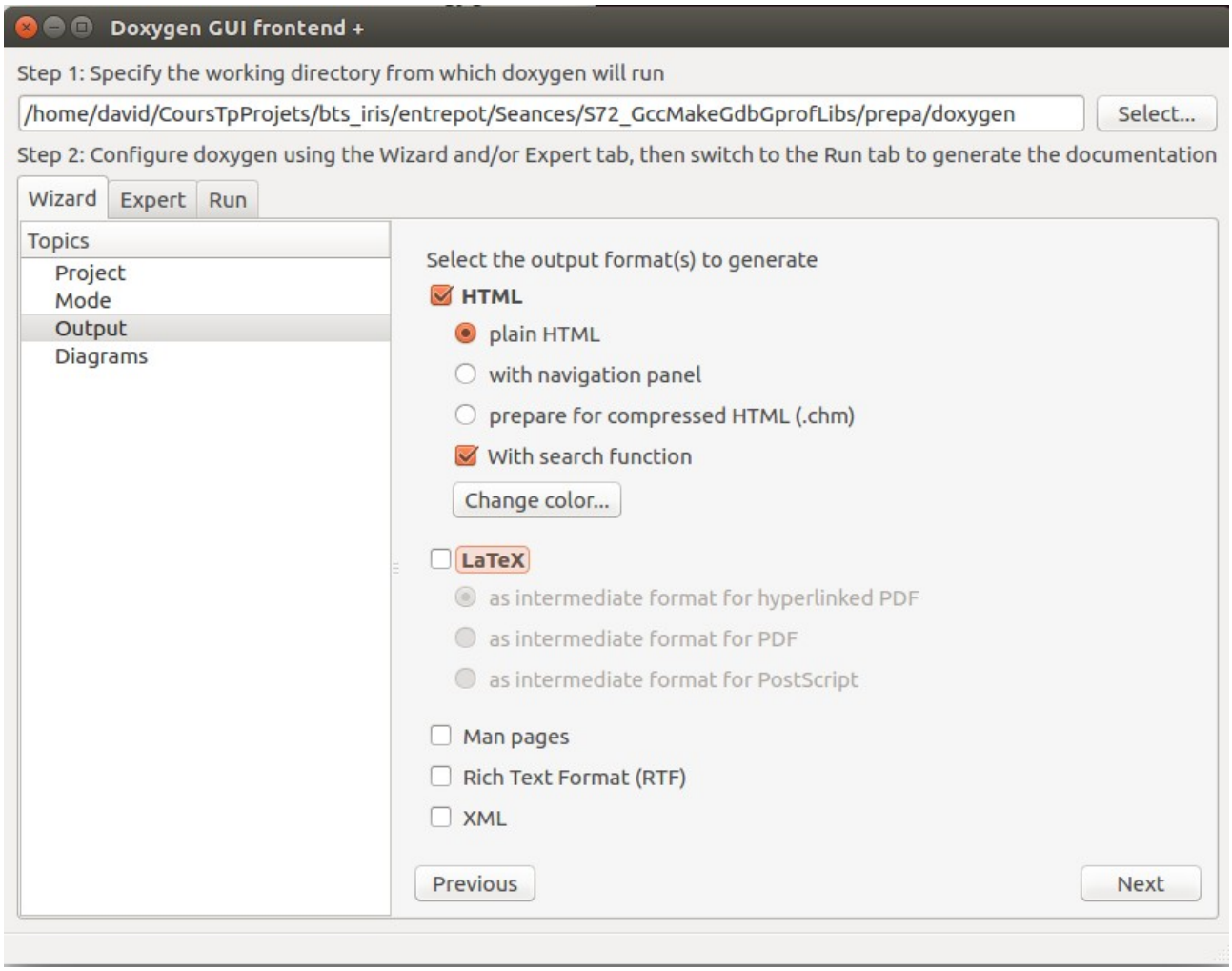
```
/**
 * @brief   Methode permettant de calculer la somme de 2 entiers
 * @param a Un argument entier
 * @param b Un autre argument entier
 * @return  La somme des 2 arguments a et b passés en paramètres
 */
int Demonstration::somme(int a, int b)
{
    // Commentaire "classique" dans la fonction
    return (a + b);
}
```

Le logiciel **Doxygen** permet à partir des blocs de commentaires de générer automatiquement toute la documentation technique du code source.

```
$ doxywizard
```

L'IHM permet alors de configurer la génération de cette documentation au travers de différents paramètres :

- dossier de travail
- dossier contenant le code source à analyser
- format de sortie (HTML, Latex/PDF, man pages, chm…)
- diagrammes
- moteur de recherche intégré



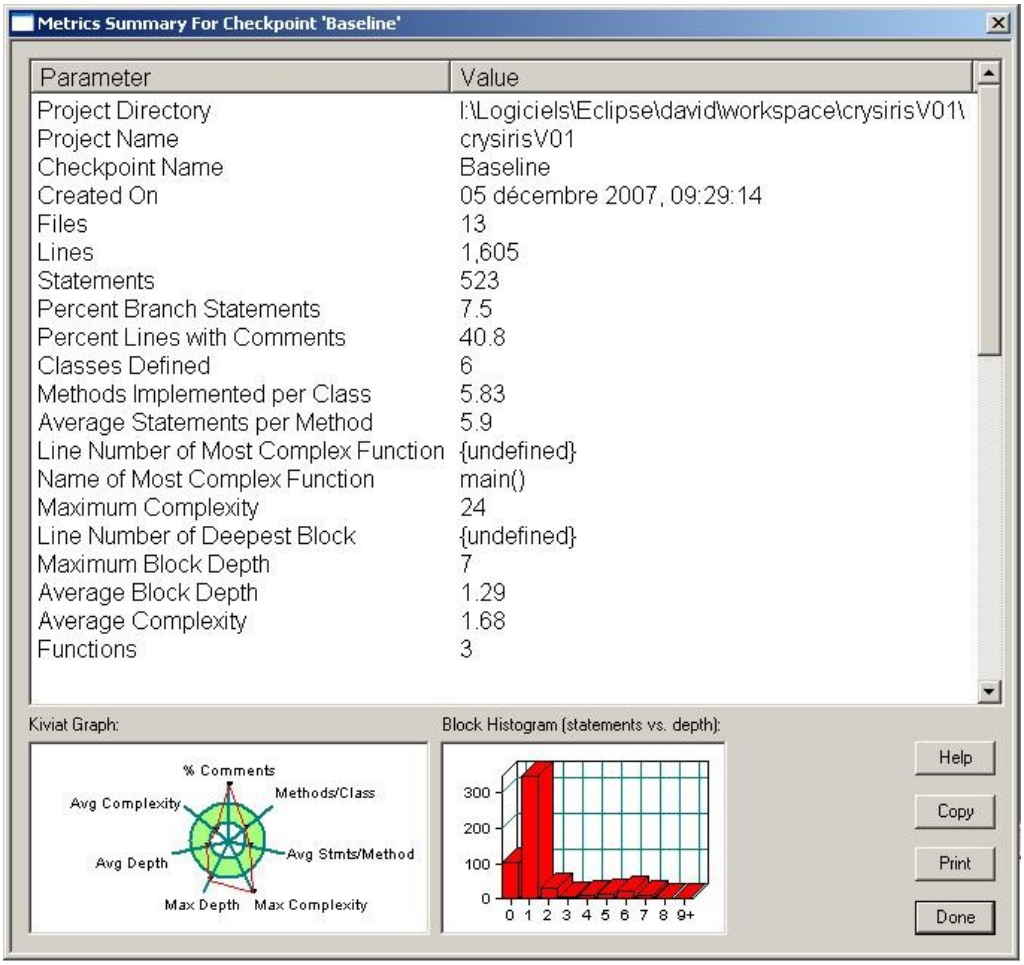
On ajoute souvent une page d'accueil **mainpage.md** pour présenter la documentation générée. Le format utilisé est Markdown (comme sur Github)

6.4 – Métriques & statistiques (Source Monitor)

Les métriques d'un projet, sont des indices dont l'évolution permet de caractériser l'avancement et la qualité d'un projet informatique. Quelques exemples de métriques :

- Nombre de lignes de code source (sloc)
- Nombre d'instructions
- Nombre de classes
- Nombre de méthodes/classes
- Nombre d'instructions/méthodes
- Nombre de bogues
- Couverture de code (est ce que chaque ligne du code a été exécuté et testé ?)
- Complexité cyclique (chemin au travers des appels de fonctions)
- FPA (estimation de la complexité fonctionnelle)

Exemple de rapport généré par SourceMonitor :

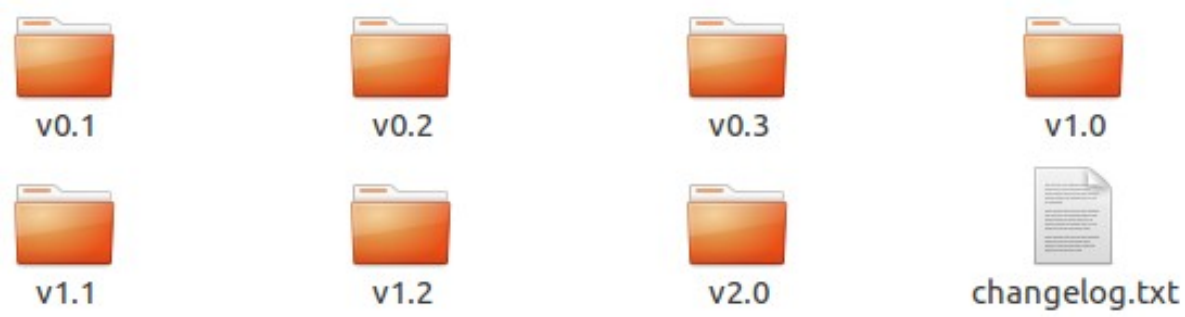


7 – Gestion des versions (git)

La gestion de versions d'un projet informatique est cruciale et complexe car elle doit permettre de gérer les différentes versions de tous les fichiers, de tous les développeurs sur toute la durée d'un projet.

7.1 – Solution minimaliste

Une première solution consiste à créer des répertoires contenant les différentes versions du logiciel développé. Il est impératif d'associer un fichier **changelog.txt** qui détaille le contenu de chaque version. On utilise ici une numérotation de type **vM.m** avec chiffre pour les versions majeures (M) et un autre pour les versions mineures (m)



Cette solution reste acceptable tant que le projet reste réduit en nombre de fichiers, de développeurs et de jours de travail. Mais il risque de devenir très vite ingérable.

7.2 – Solution avec un logiciel (git)

Comme la gestion de versions est complexe, la solution consiste alors à sous-traiter ce travail à un logiciel dédié. Il en existe plusieurs sur le marché avec chacun leurs caractéristiques :

Logiciel	***Type***	***Licence***
RCS	Gestion locale	Libre
CVS	Client/serveur	Libre
SVN	Client/serveur	Libre
Git	Décentralisé	Libre
ClearCase	Client/serveur	Propriétaire
BitKeeper	Décentralisé	Propriétaire

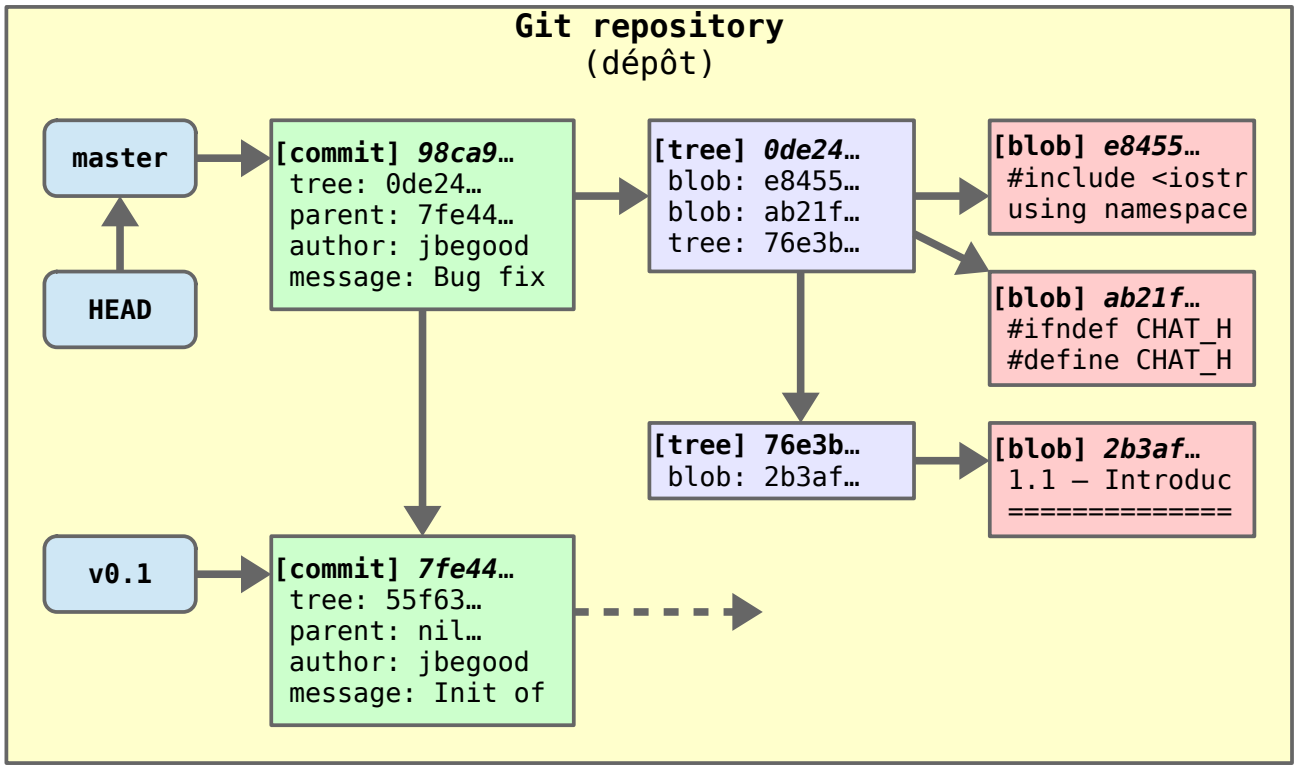
Parce que libre, décentralisé, versatile et très puissant (système de branches), **git** est aujourd'hui le logiciel de gestion de version le plus utilisé. Cependant il peut être relativement complexe à appréhender.

7.2.1 – Présentation du dépôt git

Un **dépôt** git est une collection d'objets liés entre eux et identifiés par une chaîne unique de 40 caractères hexadécimaux (SHA1). On y distingue :

- les objets **blob** (données) contenant les données binaires des fichiers
- les objets **tree** (arborescence) pointant vers les blobs et/ou vers d'autres objets tree si il y a des sous-répertoires
- les objets **commit** (instantané) pointant vers un objet tree et vers un objet commit parent

En plus de ces objets git travaille avec des **références** qui sont des pointeurs vers les objets commit. Elles servent à gérer les branches, les tags, le pointeur HEAD...



La référence **master** représente la branche de travail principale utilisée par défaut à la création d'un dépôt.

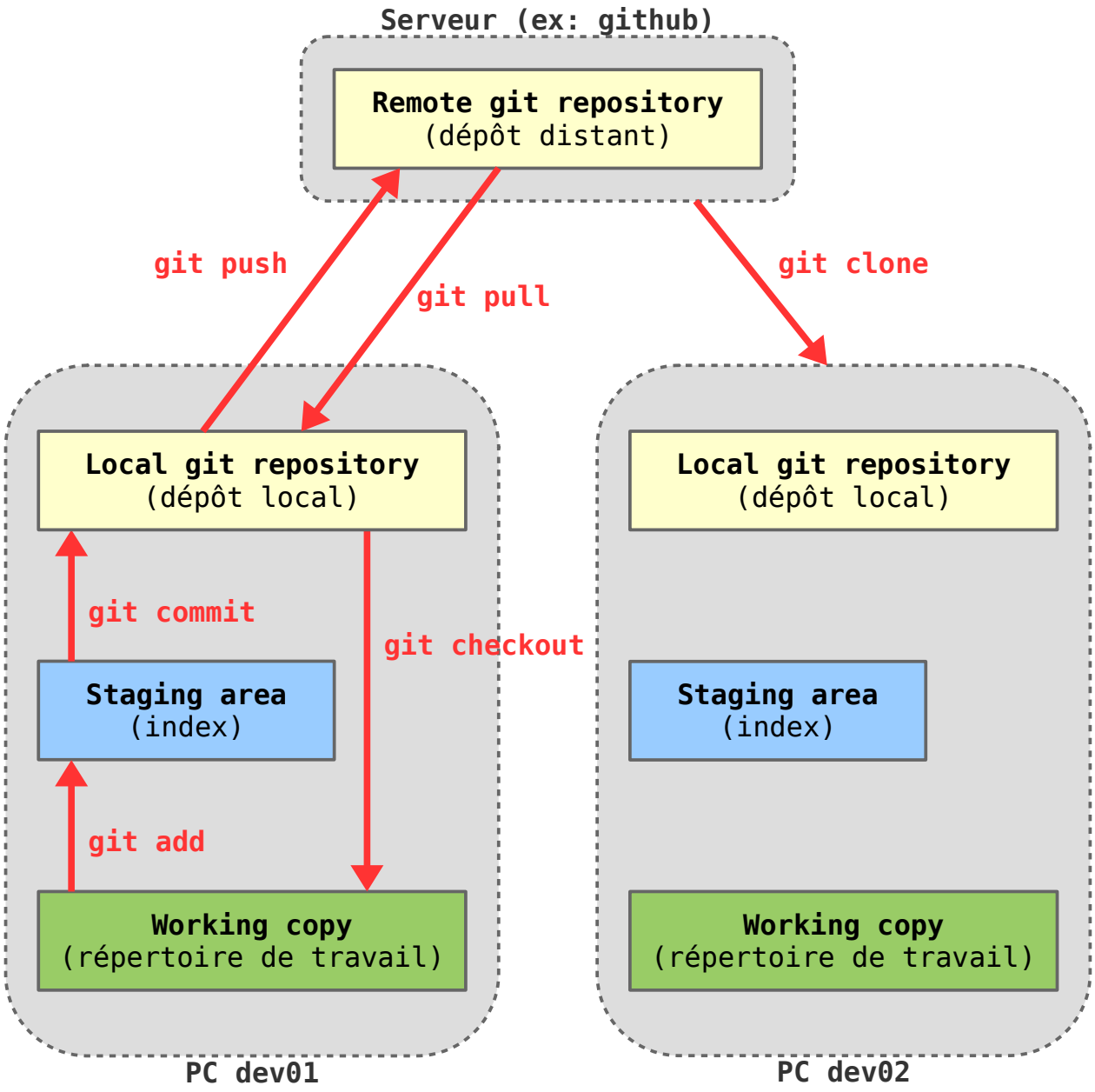
Le pointeur **HEAD** représente l'état courant et pointe sur une branche de travail, ici la branche master.

Le tag **v0.1** sert à identifier une version particulière de l'historique du dépôt.

7.2.2 – Les 3 arbres de travail

Git travaille avec 3 arbres :

- le **dépôt** (présenté précédemment)
- l'**index** qui est une zone tampon utilisée pour préparer les commits
- le r**épertoire de travail** qui contient physiquement les fichiers du projet



C'est ensuite via des **commandes** que le développeur peut interagir avec git pour sauvegarder son travail et gérer ses versions de travail y compris à distance avec d'autres développeurs.

Les IDE permettent de faire la même chose graphiquement.

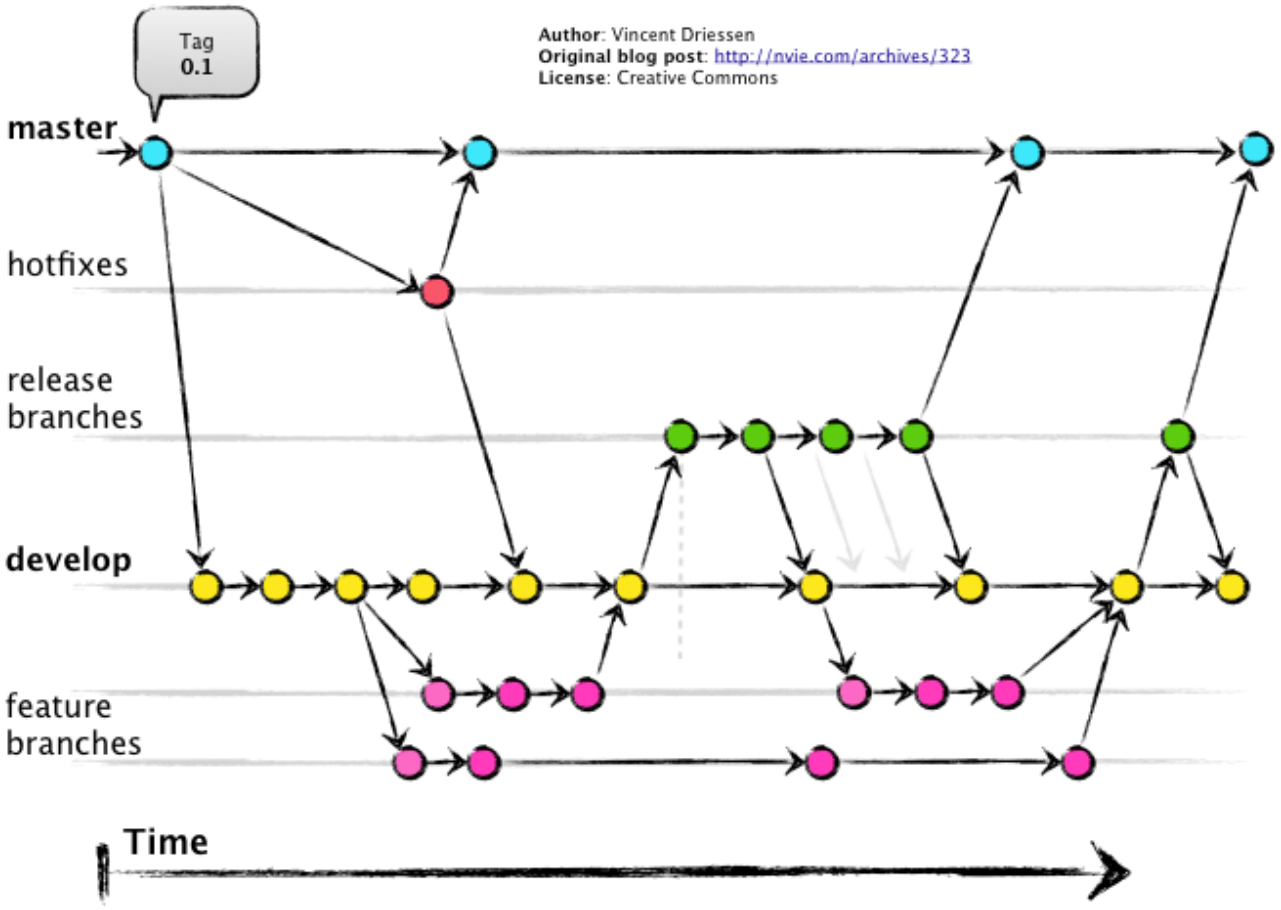
7.2.3 – Les branches

Les branches sont une des principales fonctionnalités de git et à l'origine de son succès car elles permettent une grande souplesse de travail.

On les utilise par exemple pour...

- développer une nouvelle fonctionnalité
- corriger un bug
- distinguer branche « production » et branche « développement »

... et ce sans impacter la branche principale (master)



7.2.4 – Les principales commandes

Création et configuration :

- **git init** : pour créer un nouveau dépôt vide localement
- **git clone [url]** : télécharge un dépôt distant (il sera nommé origin)
- **git config --global [option] [valeur**] : pour configurer le dépôt git

Préparation d’un commit (répertoire de travail → index) :

- **git add [fichier]** : ajoute un instantané d’un fichier à l’index
- **git reset [fichier]** : désindexe un fichier (tout en gardant son contenu)
- **git rm [fichier]** : supprime un fichier (l’index + répertoire de travail)
- **git mv [ancien] [nouveau]** : change le nom d’un fichier dans l’index

Gestion des commits (index → dépôt local) :

- **git commit -m "[message]"** : enregistre les instantanés de l’index dans le dépôt
- **git reset [commit]** : annule tous les commits après commit en conservant les modifications
- **git reset --hard [commit]** : annule tous les commits et les modifications sur les fichiers après commit (/!\ attention irréversible)

Synchronisation avec un dépôt distant (dépôt local → dépôt distant) :

- **git push** : envoie les changements du dépôt local vers le dépôt distant
- **git fetch** : récupère les changements du dépôt distant et les intègre au dépôt local
- **git merge** : fusionne les différentes versions
- **git pull** : télécharge les changements du dépôt distant et les intègre localement au dépôt local, à l’index et au répertoire courant

Affichage :

- **git log** : affiche l’historique de la branche courante
- **git status** : affiche l’état du dépôt
- **git diff** : affiche les modifications des fichiers qui ne sont pas indexées

Gestion des branches :

- **git branch [nom]** : créé une nouvelle branche
- **git checkout [nom]** : se déplace dans une branche (via pointeur HEAD) et met à jour les fichiers du répertoire de travail
- **git merge [nom]** : fusionne une branche avec la branche courante
- **git branch -d [nom]** : supprimer une branche
- **git push origin [nom]** : envoie les changements de la branche vers le dépôt distant

Gestion des tags :

- **git tag [nom] [commit]** : ajoute une étiquette à un certain commit

7.2.5 – Conseils

Quelques conseils :

- git est complexe, mais puissant. Il faut l'utiliser pour l'apprivoiser
- utiliser le même « workflow » de travail dans une équipe
- un bon commit ? Quelque chose qui marche avec un message explicite et le plus atomique possible

8 – Webographie

Quelques liens pour aller plus loin :

- http://en.wikipedia.org/wiki/Comparison_of_executable_file_formats
- https://github.com/catchorg/Catch2
- https://github.com/google/styleguide/tree/gh-pages/cpplint
- http://www.stack.nl/~dimitri/doxygen/
- https://git-scm.com/
- http://www.campwoodsw.com/sourcemonitor.html
- ...