Gestion de code

LIFAP4

http://licence-info.univ-lyon1.fr/LIFAP4

Alexandre Meyer

http://liris.cnrs.fr/~ameyer/

Gérer du code

- Objectifs
  - Travailler à plusieurs
  - Avoir du code pérenne dans le temps
  - Robuste au changement d'équipe de dév.
- Comment?
  - Définir des convention d'écriture de code
  - Utiliser un système de contrôle de version
  - Documenter le code (+ des exemples si possible)
  - Tester et retester le code

Convention d'écriture de code

La problématique

- Les contraintes du développement de projet
  - beaucoup de développeurs (qui changent d'emploi)
  - le code est conséquent
  - Il est souvent nécessaire de revenir en arrière
  - Chaque programmeur a sa propre façon de coder
  - Chaque éditeur (de texte/code) a sa propre façon de présenter le code source
- Exemple : Studio de dév de jeux vidéo
  - Plusieurs projets + outils communs
  - 100 développeurs et plusieurs millions de lignes de code

➔ Il faut se discipliner un minimum

Exemples

Que font ces programmes?

```
int truc(int* a, int b)
{
int c;
int d=a[0];
c=a[0]-a[0];
while(c<b)
{
d=(a[c]>d)?a[c]:d;
int z;
z=b-b;
while(z<c+(b/b))
{
z=z+(b/b);
}
c=z;
}
return d;
}
```

```
#include <stdio.h>#include <math.h>double l;main(_,o,O){
return putchar((_--+22&&_+44&&main(_,-43,_),_&&o)?
(main(-43,++o,O),((l=(o+21)/sqrt(3-O*22-O*O),l*l<4&&
(fabs(((time(0)-607728)%2551443)/405859.-4.7
+acos(l/2))<1.57))[" #"])):10);}
```

```
int truc2(int* a, int b)
{
int c;
int d=a[0];
for(c=0;c<b;c=c+1)
if(a[c]<d)
d=a[c];
return d;
}
```


Exemples

```
int tabTrouverMinimum(int* tab, int taille)
{
    int i;
    int mini=tab[0];
    for (i=0;i<taille;i=i+1)
        if (tab[i]<mini)
            mini=tab[i];
    return mini;
}
```

Que font ces programmes?

Conventions d'écriture

- Objectif : normaliser la rédaction du code pour
  - Augmenter la lisibilité et la compréhension du code source
  - Obtenir du code prédictible et facilement modifiable
  - Que tous les développeurs d'un même projet puissent comprendre et appréhender le code des autres
- Nous donnons ici des exemples de règles …
- Il faut s'en fixer lorsque l'on travaille à plusieurs sur un même projet mais elles peuvent différer de celle que l'on donne si il y a une bonne raison

Règles générales

- Le contenu d'un fichier ne doit pas dépasser 80 colonnes.
- Les noms doivent être tous en anglais ou tous en français
- Les bloques/fonctions doivent être indentés
- La césure des lignes trop longues doit être effectuée d'une manière lisible, logique et évidente

```
somme = a + b + c +
        d + e;
for ( int noTable = 0 ; noTable < nTables ;
    noTable += sautTable )
```


Les fichiers

- Extensions
  - Les fichiers d'entête : .h
  - Les fichiers source C : .c
  - Les fichiers source C++ : .c++, .cc ou .cpp
    (.C à bannir à cause des problèmes d'OS)
- Structure et fonctions en rapport
  - Déclarée dans un fichier d'entête : Camion.h
  - Fonctions définie dans un fichier source : Camion.c
  - Les noms des fichiers doivent correspondre au nom de la structure. struct Camion { …

Les fichiers d'entête (.h)

- Les fichiers d'entête doivent contenir une garde d'inclusion multiple

```
#ifndef __NOM_MODULE__H
#define __NOM_MODULE__H
…
#endif
```

- Les énoncés doivent pouvoir se faire indépendamment du système d'exploitation (en particulier l'usage des types de base)
- Les énoncés d'inclusion (#include <>) doivent se trouver seulement au début d'un fichier
- Les types locaux a un seul fichier doivent être déclarés à l'intérieur de ce fichier

Un .h type (en C)

```
// Camion: blahblah
#ifndef __CAMION_H__
#define __CAMION_H__

#include <stdio.h>

struct Camion
{
    int Poids;
    int AnneeImmatriculation;
};

void camInit(Camion* c);
int camAge(const Camion* c);

#endif
```

+documentation de code (cf. plus loin)

Un .h type (en C++)

```
// Camion: blahblah
#ifndef __CAMION_H__
#define __CAMION_H__

#include <stdio.h>

class Camion
{
public:
    void Camion (int _poids, int _annee);
    int age() const;
private:
    int Poids;
    int AnneeImmatriculation;
};

#endif
```

+documentation de code (cf. plus loin)

Règles de nommage

- Type (struct/class et champ de struct/class)
  - en minuscules avec le premier caractère et le début de chaque nouveau mot en majuscule → class ListeEntier, TabDynMot, …
- Fonctions, Procédures pour les structures
  - Le nom de la structure doit figurer dans le nom des fonctions
  - Pour les class le nom de la classe remplace ceci
  - Puis verbe à l'infinitif
    → par ex. Camion : camSauver(…) ou camDeplacer(…)
    → par ex. Liste chainée : lstAjouter(…), lstRechercher
    → par ex. Arbre B de R : abrAjouter(…), abrRechercher
- Constantes
  - en majuscules
  - caractère souligne entre chaque mot
  →par ex. M_PI ou TERRAIN_TAILLE_MAX

13

13

Règles de nommage

- Variables
  - minuscule sauf début de chaque mot
  - Les variables qui ont une longue portée doivent avoir des noms longs
  - Celles avec une portée réduite peuvent avoir des noms courts

Types/Struct/Class : Ligne, SystemeAudio, PointDeControle

Variables : ligne, application, compteur, cptDeLigne

Fonctions si hors class : fromageInit(), fromageSauver()

14

14

Règles de nommage

- Un peu plus loin
  - Les pluriels, pour une collection d'objets
    ListeCamions camions;
  - Premières lettres de la variable indique le type

```
int iAge;                 // un entier
float fSurface;           // un float
char txtPhrase[256];      // texte
int* piAge;               // pointeur d'entier
```

  - Les variables d'itération, un caractère minuscule

```
int i,j,k;
for(i=0;i<… for(j=0;j<… for(k=0;k<…
```

15

15

Les déclarations

- Les types
  - La conversion des types se font avec un cast de manière explicite. On ne doit jamais dépendre de la conversion implicite.

```
int a;
float  fPi=3.1415926535;
float fR = 12.5;
a = fPi*fR*fR;              // NON car conversion implicite
a = ((int)(fPi*fR*fR));     // OUI car conversion explicite
```

16

16

Les déclarations

- Les variables
  - Les variables devraient être initialisées lorsqu'elles sont déclarées (ou au plus tôt)
  - **L'utilisation des variables globales doit être bannie**
  - Une variable ne doit pas en cacher une autre

```
int poids=52, i;
char msg[12]="toto";
float fMoyenne;
for(i=0;i<12;++i)
{
        int poids=53;           // NON, poids cache poids
        …
}
while(…) { … }
fMoyenne = 15;                  // NON trop loin
```

17

17

Les déclarations

- Variables globales

  étant donné que la variable peut être modifiée depuis n'importe quelle fonction/procédure :
  - Compréhension et débogage difficile du programme
  - Recherche d'erreurs difficile
  - Modifications du programme difficile
    - il faut comprendre tout le programme pour savoir comment la variable est traitée
  - Vecteur de fuite de mémoire

→ L'utilisation des variables globales doit être bannie

18

18

Les structures de contrôle

- Les boucles
  - Seuls les énonces de boucle doivent être inclus dans la construction for()
  - Les variables de boucle doivent être initialisées juste avant la boucle
  - L'utilisation de break et continue dans les boucles doit être évitée
  - La forme while (true) ne doit être utilisée que rarement (pour les boucles infinies, cf. cours 4 IHM)

```
int i,j;
for(i=0, j=8; i<12; ++i)            // NON
{
        if (i==j) continue;         // NON!!
        …
}
```

19

19

Les structures de contrôle

- Les conditionnelles
  - Les expressions conditionnelles complexes doivent être évitées
    - If ((a && !b || c) || (b && c) || (d && f || a) { … // non
  - Le cas le plus fréquent d'une construction if doit être mis dans la partie if-then et l'exception dans la partie else
  - Les énoncés qui exécutent du traitement ne doivent pas se trouver à l'intérieur de conditions
    `if (i++==5) r=i;   // NON`

20

20

Pour aller plus loin …

- C++ Programming Style Guidelines
  http ://geosoft.no/development/cppstyle.html
- Code Complete, Steve McConnel - Microsoft Press
- Programming in C++, Rules and Recommendations
  M Henricson, e. Nyquist, Ellemtel (Swedish telecom)
  http ://www.doc.ic.ac.uk/lab/cplus/c++.rules/
- C++ Coding Standard, Todd Ho
  http ://www.possibility.com/Cpp/CppCodingStandard.htm
- C / C++ / Java Coding Standards from NASA
  http ://v2ma09.gsfc.nasa.gov/coding standards.html

21

21

Système de contrôle de version

22

22

Organisation classique des fichiers d'un projet

```
Pacman/
    src/                              ← le répertoire avec les sources
        math/
            NbComplexe.h
            NbComplexe.c
        jeu/
            Terrain.h
            Terrain.cpp
        …
    data/                    ← contenant les données (images, etc.)
        …
    bin/
        pacman.exe           ← l'exécutable
    doc/                     ← la documentation du code
        doxyfile             ← fichier de configuration de doxygen
    Makefile                 ← pour compiler
    README.txt           ← le minimum pour commencer à utiliser le projet
```

23

23

Les contraintes du développement

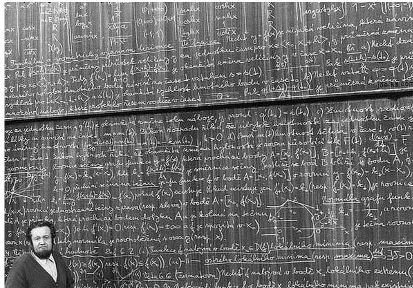
- Travail collaboratif :
  - Plusieurs développeurs (travail concurrent, mise à jour)
  - Conserver un historique d'évolution du projet
  - Conserver un historique de versions stables
  - Travailler en parallèle + reproduire les bugs
- Donc
  - Utilisation d'un outil de 'source control'
  - (Pas de .zip échangé par mail)

24

24

Gestionnaire de version

- Vous serez de simple utilisateur
- Fonctionnement interne ne nous intéresse pas ici

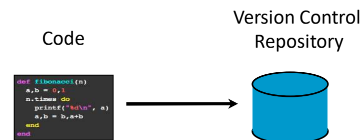


25

25

Architecture de la gestion de version

- Principes généraux
  - Le code est stocké sur un dépôt/repository (serveur)
  - les programmeurs (via un client)
    - y accèdent pour obtenir une copie de travail (clients)
    - mettent à jour le dépôt en fonctions de leurs modifications
    - récupère les modifications faites par les autres développeurs
  - le serveur conserve l'historique des modifications



26

26

Les outils de gestion de version

- Centralisés : 1 serveur
  - CVS
  - SourceSafe
  - Perforce
  - ClearCase
  - SubVersion (SVN)
- Distribués : n serveurs
  - Mercurial
  - Git
  - Bazaar

Lequel est le mieux?

27

27

Les outils de gestion de version

- Sécurisation
  - Le développeur peut 'pousser' son code régulièrement (plusieurs fois par jour)
  - On est moins tributaire des disques durs utilisateurs
  - Seul le serveur doit être archivé !
- Historique
  - Chaque version a un numéro de révision associé
  - On peut récupérer une version antérieure d'un fichier
  - Chaque révision peut avoir un descriptif associé
  - Permet d'avoir un suivi fin de l'évolution d'un projet
  - Le descriptif est très important !

28

28

Git

- Présentation générale
  - Stable et très utilisé (Open Source)
  - Simple à installer (Linux, Windows)
  - Serveur standard via Apache2
  - Existe des Interface graphique (TortoiseGIT)
  - Gratuit
  - Permet de travailler 'déconnecté' du serveur
  - Par rapport à SVN : permet de travailler sur la base de code 'déconnecté' du réseau

29

29

De quoi a-t-on besoin ?

- Le programme (client)
  - git : client en ligne de commande

    sudo apt-get install git (linux/ubuntu)
  - Google « windows install git » https://git-scm.com/download/win
  - TortoiseGIT : extension pour l'explorateur de windows

    https://tortoisegit.org/
  - GIT ajoute un répertoire .git pour stocker ses informations internes
- Serveur http://forge.univ-lyon1.fr/
  - Compte Lyon 1

30

30

GIT vocabulaires et principes

- **Dépôt central** : version du projet stocké sur le serveur
  - Le dépôt est repéré par une adresse web (URL), par exemple **https://forge.univ-lyon1.fr/git/gefo**
  - on peut consulter le dépôt comme une page web classique
- **Dépôt local** : version du dépôt stocké en local sur chaque machine des developpeurs
- **Copie de travail** : la version du projet qui se trouve en local sur votre machine sur laquelle le développeur travail

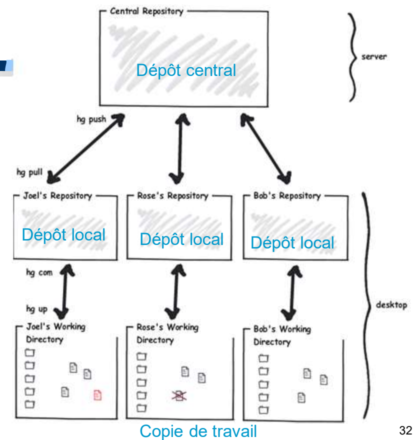
➔ Chaque développeur a un dépôt local et une copie de travail

31

31

Git

Le résumé de base à retenir



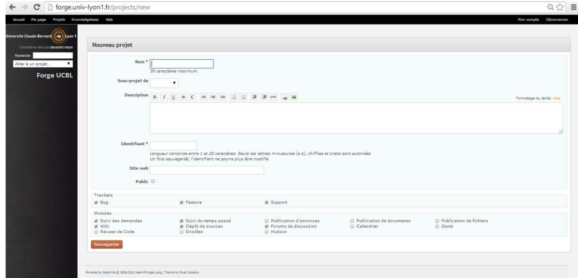
Copie de travail

32

32

git : commandes de base

- Création d'un projet
  - Se fait sur le serveur via une page web
    http://forge.univ-lyon1.fr/projects/new



33

git clone

- Clône une base de code
  - **Appel** : git clone URL [PATH]
  - En général, récupère en local une copie du dépôt central (web) repéré par l'URL
  - En utilisation simple, à ne faire qu'une fois par machine (pour la création des répertoires de travail en local) !
  - Ensuite on utilisera que git commit/push et git pull
- Exemple
  $ git clone http://git.serveur.zz/.../Pacman

34

34

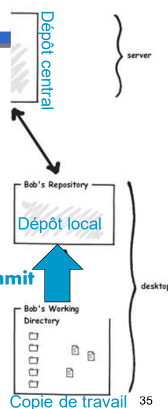
git commit

- Envoie des modifications
  - **Appel** : git commit -m "description" [PATH]
  - Envoie les modifications du répertoire de travail vers le dépôt local
  - Incrémente le numéro de révision
  - Echoue si la copie de travail n'est pas à jour
  - Alias : git ci OU git com
- Exemple
  $ vi src/terrain.h && vi src/Terrain.c
  $ git commit –m "ajout de f affichage" src/Terrain.c

Ou

  $ git commit –m "ajout de f affichage" .

Tout le répertoire



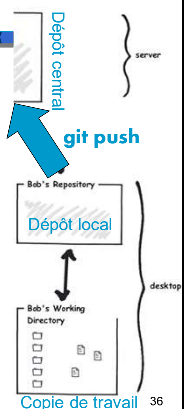
Copie de travail

35

35

git push

- Envoie des modifications
  - **Appel** : git push
  - Envoie les modifications du dépôt local vers le dépôt central (web)
- Exemple d'utilisation
  $ vi src/terrain.h && vi src/Terrain.c
  $ git commit –m "Terrain.c.h: ajout de la fonction d'affichage" src/Terrain.c
  $ git push



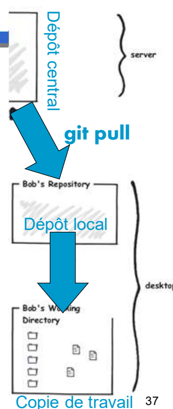
Copie de travail

36

36

git pull

- Mise à jour du dépôt local et des fichiers locaux
  - **Appel** : git pull
  - Applique sur le dépôt local les modification du dépôt central
  - Applique sur la copie de travail les modification du dépôt local
- Exemple
  $ git pull
  1 files updated, 0 files merged, 0 files removed, 0 files unresolved

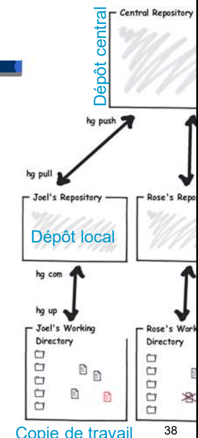


37

37

GIT principes résumés

- **Commit (s'engager)** : transférer les modifications effectuées sur la copie de travail vers le dépôt local
- **Push (pousser)** : transférer les modifications effectuées sur le dépôt local vers le dépôt central
- **Pull (tirer)**: mettre à jour le dépôt local et la copie de travail avec les modifications du dépôt central (web)

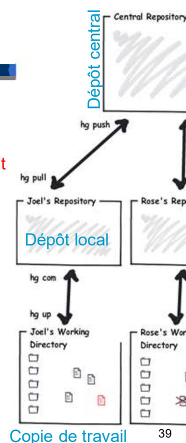


38

38

GIT : utilisation simple

- Grouper les commandes commit/push
  - SEULEMENT POUR UNE UTILISATION SIMPLE (donc en LIFAP4)
  - Après vous ferez moins de push car un commit peut-être annulé, pas un push
- Et toujours faire pull avant de faire un commit/push
- Récupérer des changements
  $ git pull
- Entrer des changements
  $ git pull # pour être sur d'être à jour
  $ git commit –m "descriptions"
  $ git push



39

39

git status

- Etat d'une copie locale
  - **Appel** : git status [path]
  - Affiche l'état des fichiers du répertoire path (tous les fichiers si vide)
  - Utilise le .git pour analyser l'état du répertoire local (modifié, ajouté, effacé, conflit, verrouillé, …)
- Exemple
  $ git status
  M src/Terrain.c // signifie que le fichier est modifié par rapport au dépôt local

40

40

git log

Appel : git log
- Donne tout l'historique des versions

- Exemple
  $ git log
  commit db27c56b469ef9f912867fd649ac712a583aca6d (HEAD -> master, origin/master, origin/HEAD)
  Author: Alex <alexandre.meyer@univ-lyon1.fr>
  Date: Fri Jan 19 09:39:37 2018 +0100
  ...

41

41

Git : commandes de base

Ajout / suppression de fichier

- **Appel** : git add [path]
  - le fichier est noté "à ajouter" lors du prochain commit
  - Fonctionne récursivement

  git add src/*
  !!! NE PAS METTRE TOUS LES FICHIERS DANS GIT
  Mettre seulement ceux qui ne sont pas régénérables :
  *.h *.cpp readme.txt data Makefile project.cbp
  Ne pas mettre :
  *.o les exécutables tmp log debug
  ➔ .gitignoe permet de lister des fichiers à ignorer

42

42

Git : commandes de base

Ajout / suppression de fichier

- **Appel** : git add [path]
  - le fichier est noté "à ajouter" lors du prochain commit
  - Fonctionne récursivement
- **Appel** : git rm [path]
  - le fichier est noté "à effacer" lors du prochain commit.
  - Fonctionne récursivement
- **Appel** : git mv src dest
  - déplace un fichier/répertoire et note le renommage pour le prochain commit
  - Conserve l'historique

43

43

git tag

Appel : git tag
- Liste toutes les versions qui ont un tag

Appel : git tag –a TAG –m "msg"
- Ajoute un TAG à la version courante

- Exemple

```
$ git tag
v0.1
v1.3
```

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

44

44

git checkout

Appel : git checkout [fichier]
- Remet la copie de travail dans l'état HEAD (la dernière version commit)

!!! Perte des changements

- Exemple

$ git pull
vi toto.cpp
➔ Finalement ne veut plus faire ces changements
$ git checkout

45

45

git reset (avancé pour LIFAP4)

Appel : git reset HEAD
- Reviens à la tête du dépôt

Appel : git reset HEAD^
- Annule le dernier commit et reviens à l'avant dernier

Appel : git reset HEAD^^
- Annule les 2 derniers commits et reviens à l'avant avant dernier

- Exemple

$ git pull
vi toto.cpp
git commit –m "changements" .
➔ Finalement veux annuler ce commit
$ git reset HEAD^

46

46

git stash (avancé pour LIFAP4)

Appel : git stash
- Remet un fichier ou toute la copie de travail dans l'état du dépôt local
- Sauvegarde tous les changements que vous avez fait

Appel : git stash pop
- Réapplique les changements sauvés sur les fichiers

Appel : git stash list
- Liste tous les changements sauvés

- Exemple

$ git pull
➔ Par exemple si conflit sur un fichier
$ git stash (sauve les changements et revient à HEAD)
$ git pull (applique les changements des autres developpeurs)
$ git stash pop (applique vos changements dessus en écrasant)

47

47

Git : à N développeurs

- **Modifications sur des fichiers différents**
  - Un développeur B modifie sa copie de travail : titi.cpp et fait git commit/push
  - Vous (dev. A) modifiez votre copie de travail : le fichier toto.cpp

Pour récupérer les changements de B, vous faites
$ git pull
Dans votre copie de travail, vous avez bien vos modifications sur toto.cpp et celles de l'autre développeur sur titi.cpp

➔ Sans passer par un mail avec un envoie de fichier ou de .zip !!!

48

48

Git : à N développeurs

- **Modifications de deux endroits distincts d'un même fichier**
  - Un développeur B modifie sa copie de travail : titi.cpp
    et fait git commit/push
  - Vous (dev. A) modifiez votre copie de travail : le fichier titi.cpp mais pas au même endroit que B

Pour récupérer les changements de B, vous faites
$ git pull
Dans votre copie de travail, vous avez bien vos modifications sur titi.cpp et celles de l'autre développeur sur titi.cpp

➔ Sans passer par un mail avec un envoie de fichier ou de .zip, ni de merge manuel !!!

49

49

Git : à N développeurs (avec conflit)

Modifications de deux endroits identiques d'un même fichier

- Vous modifiez votre copie de travail : le fichier toto.cpp
- un développeur B modifie sa copie de travail : toto.cpp également au même endroit que vous et fait git commit/push
- Quand vous faites git pull les deux changements apparaissent avec des <<<<<<<<< et des >>>>>>>>>>>>

- à vous de faire la fusion à la main
- puis de lui dire que le conflit est résolu

```
#include <stdio.h>
int main(int argc, char **argv)
{
  printf("hello, world!\n");
<<<<<<< local
  printf("I'm not using CVS");
=======
  printf("I'm using git!\n");
>>>>>>> other
  return 0;
}
```

$ git commit -m "resolution conflit"
$ git push

Sans chemin car on commit toute la branche

50

50

git : commandes de base

- Aide sur les commandes
  - **Appel** : git help[COMMAND]
  - Affiche la liste des commandes disponibles
  - Si COMMAND est spécifié affiche l'aide de cette commande
- Exemple
  $ git help
  $ git help commit

51

51

GIT pour votre projet

Indispensable !!!

https://git-scm.com/docs

- Tutoriaux simples existent sur internet mais
  - il faut pratiquer
  - Dès le début de votre projet
- De nombreuses choses à apprendre plus tard
  - git diff
  - Branches
  - …

52

52

Documentation de code

53

53

Introduction

- Principe
  - La génération de documentation pour les codes sources est complexe si elle est faite à la main
  - On utilise des outils pour générer de manière semi-automatique cette documentation
- Doxygen
  - Pour documenter du code source C, C++, Java
  - Basé sur un ensemble de balises à ajouter dans les sources
  - Différents formats de sorties : RTF (MS-Word), PostScript, PDF avec liens hypertexte, HTML (compresse ou pas), Unix, Man pages

54

54

Introduction

- Exemple de résultats
  → Pacman/doc/html/index.html

55

De quoi a-t-on besoin ?

- Doxygen :
  http ://www.stack.nl/dimitri/doxygen/download.html
- Optionnel
  - HTML Workshop (pour HTML compressé) :
  - intègre aux outils de Visual Studio
    help sur http ://msdn.microsoft.com/
  - Graphviz (pour les graphes) :
    http ://www.graphviz.org/
  - MikTex/Latex (génération de PDF) :
    http ://www.miktex.org/setup.html

Remarque : il existe de nombreux outils similaires à Doxygen

56

Procédure d'installation

- Windows
  - Doxygen : lancer doxygen-xxx-setup.exe
  - HTML Workshop (pour HTML compressé) :
    ouvrir le répertoire de HtmlHelp lancer htmlhelp.exe
  - Graphviz (pour les graphes) :
    lancer graphviz-xxx.exe
  - MikTex/Latex (generation de PDF) :
    lancer small-miktex-xxx.exe
- Linux
  - apt-get install …

57

Organisation classique des fichiers d'un projet

```
Pacman/
    src/                        ← le répertoire avec les sources
        math/
            NbComplexe.h
            NbComplexe.c
        jeu/
            Terrain.h
            Terrain.cpp
        …
    data/                       ← contenant les données (images, etc.)
        …
    bin/
        pacman.exe              ← l'éxecutable
    doc/                        ← la documentation du code
        doxyfile                ← fichier de configuration de doxygen
    Makefile                    ← pour compiler
    README.txt                  ← le minimum pour commencer à utiliser le projet
```

58

Structure des répertoires

Répertoire réservé à la documentation
- Doxyfile
- [template.tex]
- [pied_page.html]
- [images]
  - xxxx.png
  - xxxx.eps
- html
  - … Fichiers html et chm générés
- latex
  - … Fichiers LateX et PDF générés

Fichiers d'entrée
Répertoires de sortie générés
Répertoires à créer

59

Exécution

1) Générer le fichier de configuration
   ~bob$ cd ~bob/Pacman/doc
   ~bob/Pacman/doc$ doxygen -g doxyfile
   Ceci produit un fichier 'doxyfile' de configuration
   ~bob/Pacman/doc $ ls
   doxyfile
2) Editer le fichier doxyfile
   ~bob/Pacman/doc $ emacs doxyfile
3) Générer la documentation
   - A partir du fichier de configuration
   ~bob/Pacman/doc $ doxygen doxyfile
   → génération de la documentation

60

Les sections du fichier de configuration

- Fichier de configuration
  - Découpé en sections
  - très bien documenté
- Les sections
  - options du projet
  - options de compilation
  - options de traitement des fichiers source
  - options de sortie : HTML, Latex, pages de man, ...
  - options pour les graphes (dot)

61

61

Les options

- Formalisme
  NOM_OPTION = valeur option
- Les options de base
  - PROJECT_NAME, nom du projet
  - OUTPUT_DIRECTORY, répertoire de sortie
  - EXTRACT_ALL, extrait toute la documentation (YES/NO)
    - Important de le passer à YES si tout n'est pas documenté
  - INPUT, répertoire contenant les sources documentées
  - FILE_PATTERNS, motif des fichiers documentes
  - GENERATE_XXX, indique le format de sortie
    - GENERATE_HTML = YES (par défaut)

62

62

Généralités

- Les bases
  - Tout bloc de documentation devant être analyse par Doxygen commence par /** et se termine par */
  - Toute balise de génération doit être précédée de @ ou \ pour être reconnue par Doxygen
- Exemple

```
/**
@brief Description breve
Description detaillee
\<balise> [parametres]
*/
```

63

63

Généralités

- Les bases
  - Possibilités de rajouter des balises HTML
  - Un bloc de documentation précède la déclaration qui lui correspond
  - Dans tout bloc, il est recommande d'ajouter la balise @brief.
- Exemple

```
/**
@brief Method brief description
*/
Void myNewMethod(int myParam);
```

Functions

void	**myNewMethod** (int myParam) *Method brief description.*
void	**myNewMethod2** (int myParam) *Method brief description.*

64

64

Entête de fichiers

- Exemple

```
/**
    @brief File brief description
    Detailed description
    @author name
    @file file\_1.h
    @version 1.0
    @date yyyy/mm/dd
*/
```

Detailed Description

File brief description.

Detailed description

Author:
name

Version:
1.0

Date:
yyyy/mm/dd

Definition in file **file_1.h**.

65

65

Entête de fonctions

- Exemple

```
/**
    @brief brief description
    @param myParam ...
    @return none
    Example code block
    @code
    ...
    @endcode
    @warning This is a constraint
*/
void myNewMethod2 (int myParam);
```

void myNewMethod2 (int *myParam*)

Method brief description.

Method detailed description

Parameters:
myParam parameter decription

Returns:
none

Example code block

```
/* My function use example */
myNewMethod2( 10 );
```

Warning:
This is a constraint

Note:
This is a remark

66

66

Entête de fonctions

- Paramètres
  - On peut également préciser si le paramètre est entrant et/ou sortant : [in], [out] ou [in,out]
    - entrant=donnée
    - sortant=résultat
    - entrant/sortant = donnée-résultat
- Exemple

@param [in] myParam parameter decription

Parameters:

`[in]` *myParam* parameter decription

67

67

La page principale

- Cf. Pacman/src/documentation.h

```
/** \mainpage Pacman

\section Introduction
blahblah…

\section Compilation
blahblah …

\section Exécution

*/
```

Main Page | Classes | Files

Pacman

Introduction

Un début de Pacman pour le cours LIF7 (Auteur : Alexandre.Meyer@univ-lyon1.fr)

Code écrit en C/C++ (Seules notions de C++: utilisation des références pour le passage des paramètres) Se co

remarque : documentation très très minimal, tout comme le code ...

Pour compiler

Tester sous Linux(Ubuntu) et Windows.

Dépendances :

- ncurses (pcurses sous windows : http://pdcurses.sourceforge.net/
- SDL : http://www.libsdl.org/

$ make ou ouvrir le projet avec CodeBlocks puis F9

Pour executer

$ cd .../pacman_ncurses_sdl
$ bin/pacman
depuis le répertoire courant de 'pacman' sinon l'application ne trouve pas les images

Pour générer la documentation de code

Dépendance : Doxygen http://www.stack.nl/~dimitri/doxygen/ $ cd ...Pacman/doc
$ doxygen doxyfile
Puis ouvrir ...Pacman/doc/html/index.html avec firefox

ATTENTION : seul le module Terrain est documenté.

68

68

Quelques balises

- **\struct** pour documenter une structure C.
- **\union** pour documenter un union C.
- **\enum** pour documenter un type énuméré.
- **\fn** pour documenter une fonction.
- **\var** pour documenter une variable / un typedef / un énuméré.
- **\def** pour documenter un #define.
- **\typedef** pour documenter la définition d'un type.
- **\file** pour documenter un fichier.
- **\namespace** pour documenter un namespace.
- **\package** pour documenter un package Java.
- **\interface** pour documenter une interface IDL.
- **\brief** pour donner une description courte.
- **\class** pour documenter classe.
- **\param** pour documenter un paramètre de fonction/méthode.
- **\warning** pour attirer l'attention.
- **\author** pour donner le nom de l'auteur.
- **\return** pour documenter les valeurs de retour d'une méthode/fonction.
- **\see** pour renvoyer le lecteur vers quelque chose (une fonction, une classe, un fichier...).
- **\throws** pour documenter les exceptions possiblement levées.
- **\version** pour donner le numéro de version.
- **\since** pour faire une note de version (ex : Disponible depuis ...).
- **\exception** pour documenter une exception.
- **\deprecated** pour spécifier qu'une fonction/méthode/variable... n'est plus utilisée.
- **\li** pour faire une puce.
- **\todo** pour faire un To Do (= "à faire")   \bug pour déclarer un bug à corriger
- **\fixme** pour faire un Fix Me (= "Réparez-moi").

69

69

Conclusion Doc

- Doxygen permet de faire beaucoup de choses
  - documentation en ligne
  - documentation interne
  - graphe d'inclusions de fichier
  - *graphe de classe (prog objet, cf. LIFAP7 et MIF02)*
  - ...
- Se reporter a la documentation de Doxygen pour connaître les autres balises utiles
  - Ou http://franckh.developpez.com/tutoriels/outils/doxygen/

70

70

Conclusion pour LIFAP4

- Pour votre projet de LIFAP4 vous devrez écrire du code
  - '**propre**', lisible et maintenable dans le temps par toutes votre équipe de développement
  - **Documenté** (avec doxygen par exemple)
    - Faites le dès le début sinon ceci ne sert à rien
  - **Géré par un gestionnaire de version** (GIT pour nous)
    - Echange de .zip/.tgz par mail est à bannir
  - **Testé (voir CM suivant)**
    - avec Valgrind (gestion mémoire)
    - par des tests de (non-)regression ➔ chaque module a sa longue fonction de test

71

71