Gestion de code

LIF7

http://licence-info.univ-lyon1.fr/LIF7

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

7

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 )
```

8

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 { …

9

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

10

Un .h type

```
// 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)

11

Règles de nommage

- Type (struct et champ de struct)
  - en minuscules avec le premier caractère et le début de chaque nouveau mot en majuscule → struct ListeEntier, TabDynMot, …
- Fonctions, Procédures
  - Le nom de la structure doit figurer dans le nom des fonctions, par exemple au début en minuscule, éventuellement en raccourci
  - 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

12

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 : Ligne, SystemeAudio, PointDeControle

Variables : ligne, application, compteur, cptDeLigne

Fonctions : fromageInit(), fromageSauver()
```

13

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<…
```

14

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
```

15

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
```

16

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**

17

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!!
        …
}
```

18

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

19

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

20

Système de contrôle de version

21

Organisation classique des fichiers d'un projet

```
Pacman/
   src/                          ← le répertoire avec les sources
      math/
         NbComplexe.h
         NbComplexe.c
      jeu/
         Terrrain.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
```

22

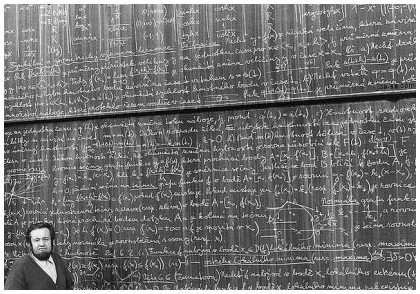
Les contraintes du développement

- 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)

23

Gestionnaire de version

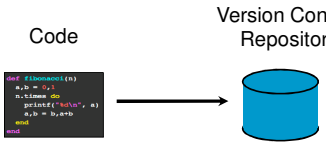
- Vous serez de simple utilisateur
- Fonctionnement interne ne nous intéresse pas ici



24

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



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?

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 !

Mercurial (HG)

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

HG De quoi a-t-on besoin ?

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

    apt-get install mercurial
  - TortoiseHG : extension pour l'explorateur de windows

    http://tortoisehg.bitbucket.org/download/index.html
  - HG ajoute un répertoire .hg pour stocker ses informations internes
- Serveur http://forge.univ-lyon1.fr/
  - Compte Lyon 1

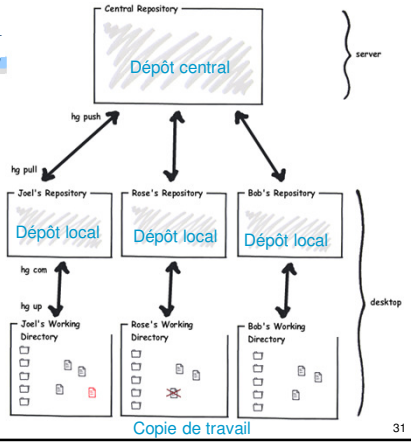
HG vocabulaires et principes

- **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/hg/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

Mercurial

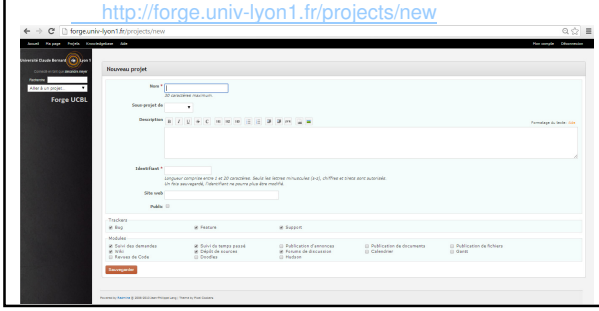
Le résumé de base à retenir



Mercurial hg : commandes de base

- Création d'un projet
  - Se fait sur le serveur via une page web

http://forge.univ-lyon1.fr/projects/new



Mercurial : hg clone

- Clône une base de code
  - **Appel** : hg 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 hg update/commit et hg pull/push
- Exemple

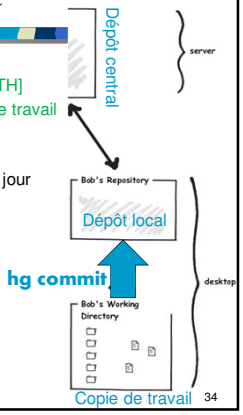
  $ hg clone http://hg.serveur.zz/.../Pacman

Mercurial : hg commit

- Envoie des modifications
  - **Appel** : hg 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 : hg ci OU hg com
- Exemple

  $ vi src/terrain.h && vi src/Terrain.c

  $ hg commit –m "Terrain.c.h: ajout de la fonction d'affichage"

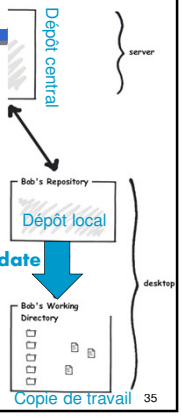


Mercurial : hg update

- Mise à jour des fichiers locaux
  - **Appel** : svn update [path]
  - Applique sur la copie de travail les modification du dépôt local
- Exemple

  $ hg update

  1 files updated, 0 files merged, 0 files removed, 0 files unresolved



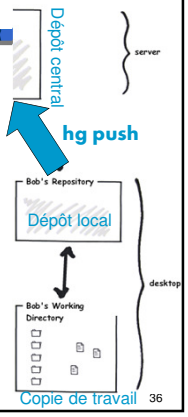
Mercurial : hg push

- Envoie des modifications
  - **Appel** : hg push
  - Envoie les modifications du dépôt local vers le dépôt central (web)
- Exemple d'utilisation

  $ vi src/terrain.h && vi src/Terrain.c

  $ hg commit –m "Terrain.c.h: ajout de la fonction d'affichage"

  $ hg push



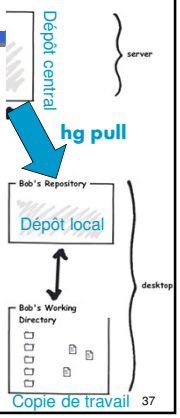
Mercurial : hg pull

- Mise à jour du dépôt local
  - **Appel** : hg pull
  - Applique sur le dépôt local les modification du dépôt central

- Exemple

  $ hg update

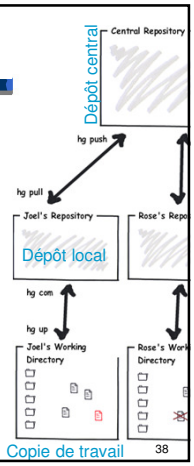
  1 files updated, 0 files merged, 0 files removed, 0 files unresolved



37

HG principes résumés

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



38

HG : utilisation simple

- Toujours grouper les commandes
  - commit/push et pull/update
- Et toujours faire pull/update avant de faire un commit/pull

- Récupérer des changements

  $ hg pull

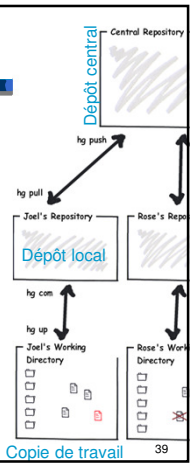
  $ hg update
- Entrer des changements

  $ hg pull

  $ hg update

  $ hg commit –m "descriptions"

  $ hg push



39

Mercurial : hg status

- Etat d'une copie locale
  - **Appel** : hg status [path]
  - Affiche l'état des fichiers du répertoire local
  - Utilise le .hg pour analyser l'état du répertoire local (modifié, ajouté, effacé, conflit, verrouillé, …)

- Exemple

  $ hg status

  M src/Terrain.c // signifie que le fichier est modifié par rapport au dépôt local

40

Mercurial : hg revert

- Remet un fichier ou toute la copie de travail dans l'état du dépôt local
  - **Appel** : hg revert [path]

- Exemple

  $ hg revert Terrain.c

renomme Terrain.c en Terrain.c.orig et restaure Terrain.c dans l'état du dépôt local (cad sans les cahngements non commités)

41

Mercurial hg : commandes de base

Ajout / suppression de fichier

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

42

Mercurial : à N développeurs

- Modifications sur des fichiers différents
  - Un autre développeur que vous modifie sa copie de travail : titi.c et fait hg commit/push
  - Vous modifiez votre copie de travail : le fichier toto.c

Vous faites

$ hg pull

$ hg update

Dans votre copie de travail, vous avez bien vos modifications sur toto.c et celles de l'autre développeur sur titi.c

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

Mercurial : hg merge (simple)

- Vous modifiez votre copie de travail : le fichier toto.c
- un autre développeur modifie sa copie de travail : toto.c également et fait hg commit/push
- Quand vous faites hg pull/update
  - hg créer une branche. Il n'applique pas les changements de l'autre développeur
  - Il faut lui dire explicitement avec la commande : hg merge

hg merge → 2 cas possible

a) hg sait fusionner les 2 modifications

quand elles ont eu lieu a des endroits différents de toto.c

hg fusionne donc les changements et rien à faire

Mercurial : hg merge (avec conflit)

- Vous modifiez votre copie de travail : le fichier toto.c
- un autre développeur modifie sa copie de travail : toto.c également et fait hg commit/push
- Quand vous faites hg pull/update
  - hg créer une branche. Il n'applique pas les changements de l'autre développeur
  - Il faut lui dire explicitement avec la commande : hg merge

hg merge → 2 cas possible

b) hg ne sait pas fusionner car même lignes

- fichier toto.c passe en mode conflit et contient les 2 versions
- à vous de faire la fusion à la main
- puis de lui dire que le conflit est résolu

$ hg resolve -m toto.c

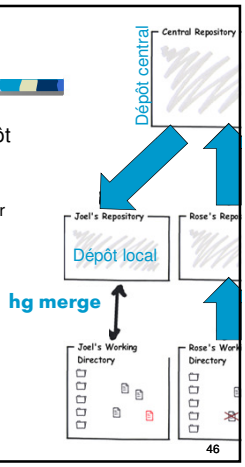
```
#include <stdio.h>
int main(int argc, char **argv)
{
  printf("hello, world!\n");
<<<<<<< local
  printf("I'm not using CVS");
=======
  printf("I'm using Mercurial!\n");
>>>>>>> other
  return 0;
}
```


Mercurial : hg merge

- Fusionne les changements de la copie de travail avec ceux du dépôt local
  - **Appel** : hg merge
  - Il faut faire un commit/push après pour passer ces changements au dépôt local/central
- Exemple

$ hg merge

merging hello.c 0 files updated, 1 files merged, 0 files removed, 0 files unresolved (branch merge, don't forget to commit)



Mercurial hg : commandes de base

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

$ hg help

$ hg help commit

Mercurial pour votre projet

Indispensable !!!

Un tutorial en français:

http://mercurial.selenic.com/wiki/FrenchTutorial

Documentation de code

49

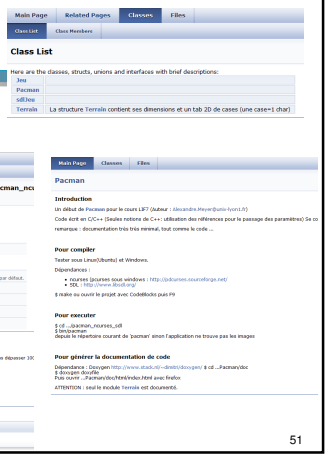
Introduction

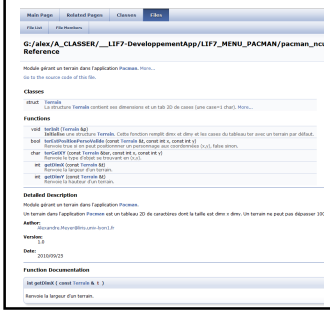
- Principe
  - La génération de documentation pour les codes sources est complexe si elle est faite à la main
  - On utilise des outils pour générer de manière semi-automatique cette documentation
- Doxygen
  - Pour documenter du code source C, C++, Java
  - Basé sur un ensemble de balises à ajouter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

50

Introduction

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





51

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

52

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 ...

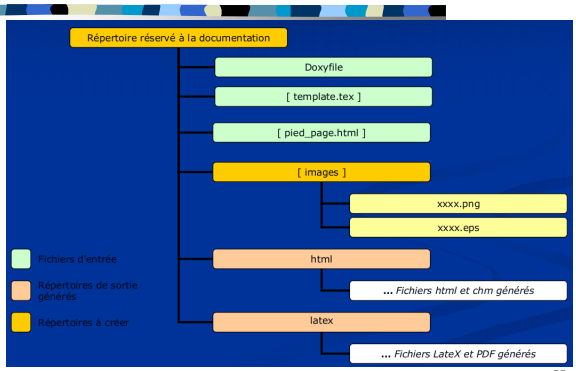
53

Organisation classique des fichiers d'un projet

```
Pacman/
   src/                          ← le répertoire avec les sources
      math/
         NbComplexe.h
         NbComplexe.c
      jeu/
         Terrrain.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
```

54

Structure des répertoires



55

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

56

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)

57

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)

58

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]
*/
```

59

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.

60

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**.

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

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

La page principale

- Cf. Pacman/src/documentation.h

```
/** \mainpage Pacman

\section Introduction
blahblah…

\section Compilation
blahblah …

\section Exécution

*/
```

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").

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. LIF13 et MIF31)*
  - ...
- Se reporter a la documentation de Doxygen pour connaître les autres balises utiles
  - Ou http://franckh.developpez.com/tutoriels/outils/doxygen/

Vérification de code, test de régression

Valgrind

- Valgrind
  - http://valgrind.org/
  - 6 outils
    - Outils de vérification mémoire
    - *Profiler* : graphe d'appel
    - *Profiler* : cache et saut
    - *Profiler* : tas
    - Thread : detection d'erreur

Valgrind

```
   // exemple.c : gcc –g exemple.c –o myprog
1  #include <stdlib.h>
2
3  void f(void)
4  {
5     int* x = malloc(10 * sizeof(int));
6     x[10] = 0;        // problem 1: heap block overrun
7  }                    // problem 2: memory leak -- x not freed
8
9  int main(void)
10 {
11    f();
12    return 0;
13 }
```


Valgrind

- Compiler avec option de debug : gcc **–g**
  gcc –g exemple.c –o myprog
- valgrind --leak-check=yes myprog arg1 arg2

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
```
➔ Écriture en mémoire non allouée, ligne 6

```
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```
➔ Mémoire non libérée, allouée en ligne 5

Valgrind

- Valgrind
  - Plus de chose à expérimenter en TD/TP
  - Prenez le temps de bien regarder, vous en gagnerez plus tard
  - Voir la doc : http://valgrind.org/docs/manual/manual.html
- Attention avec des lib externes
  - Souvent elles produisent des erreurs avec Valgrind
  - Les msg peuvent être long
  - Difficile d'extraire ses erreurs de celles des libs

  ➔ s'habituer au msg de valgrind sur des prog courts

Tests de (non) régression

- Tests de régression : à chaque fois que le logiciel est modifié, s'assurer que "les choses qui fonctionnaient avant fonctionnent toujours"
- Pourquoi modifier le code déjà testé ?
  - correction de défaut
  - ajout de fonctionnalités
- Quand ?
  - en phase de maintenance / évolution
  - ou durant le développement

TestRegression : Module

- Dans chaque module vous ajouterez une fonction de test *modTestRegression* qui vérifie
  - des éléments/constantes de base
    - par ex. var_taille>=0
  - des fonctionnements
    - Appel une fonction
    - Vérifie qu'elle a bien fait ce qu'elle prétend

  →Utilisez des assert
- Cette fonction
  - Sera longue car de nombreux tests sont toujours à faire
  - peut bien sûr être découpée en plusieurs sous-fonctions
- Chaque module viendra avec un exécutable qui appelle cette fonction

Exo : Ecrivez la fonction de TestRegression qui

- teste un module Nombre Complexe
- teste un module TabDyn
  - plusieurs Ajout à un TabDyn
  - plusieurs suppression à un TabDyn
  - la fonction qui trie un TabDyn
- teste un module ListeChainee
- teste un module ArbreBinaireDeRecherche

Test régression : module NbComplexe

```
struct NbComplexe
{
    float re,im;
};
void nbcInit(NbConplexe* n, const float re, const float im);
void nbcInitExpo(NbConplexe* n, const float radius, const float theta);
NbComplexe nbcAjouter(NbComplexe a, NbComplexe b);
NbComplexe nbcMultiplier(NbComplexe a, NbComplexe b);
NbComplexe nbcNegatif(NbComplexe a);
bool nbcEstReel(NbComplexe n);
bool nbcEstImaginaire(NbComplexe n);

void tdTestRegression();
```


Test régression : module TabDyn

```
struct TabDyn
{
        Element* tab;
        int taille, taille_alloc;
};
void tdInit(TabDyn* t);
void tdAjouter(TabDyn* t, const Element* e);
const Element* tdGetConst(TabDyn* t, int i);
Element* tdGet(TabDyn* t, int i);
int tdTaille();
void tdSupprimer(TabDyn* t, int i);
int tdTrouver(TabDyn* t,  const Element* );
Void tdTrier(TabDyn* t);          // suppose que on sait comparer Element
void tdLiberer(TabDyn* t);
void tdTestRegression();
```


void tdTestRegression();

```
{
    TabDyn* td;
    Element e;  eltInit(e, …);

    tdInit(td);
    assert( td.taille==0 );
    assert( td.taille_alloc*sizeof(Element) == malloc_usable_size(td.tab);

    tdAjouter( td, &e);
    assert( td.taille==1 );
    assert( td.tab[0] == e );
    assert(   *tdGetConst(td,0) == e );
    assert(   *tdGetConst(td,0) == td.tab[0] );
    … toutes les fonctions sont testées …
```


void tdTestRegression();

```
    … signifie que l'on test toutes les fonctions …
    tdLiberer( td );
    assert( td.taille==-1 );
    assert( td.tab == NULL );
}

-----------------------------------------
// TabDyn_TestRegression.c
#include <TabDyn.h>
int main()
{
    tdTestRegression();
    return 0;
}
```

Conclusion globale

- Pour votre projet de LIF7 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** (Mercurial pour nous)
    - Echange de .zip/.tgz par mail est à bannir
  - **Testé**
    - avec Valgrind (gestion mémoire)
    - par des tests de (non-)regression ➔ chaque module a sa longue fonction de test

79

Anciens transparents

80

Subversion (SVN)

- Présentation générale
  - Source Control basé sur CVS
  - Maintenant stable et très utilisé (Open Source)
  - Simple à installer (Linux, Windows)
  - Serveur standard via Apache2
  - Existe des Interface graphique (TortoiseSVN)
  - Gratuit
  - Permet de travailler 'déconnecté' du serveur

81

Subversion (SVN)

- Les différents programmes
  - svn : client en ligne de commande
  - svnadmin : outil d'administration du dépôt/repository
  - svnlook : inspection du dépôt/repository
  - svnserve : serveur SVN
  - TortoiseSVN : extension pour l'explorateur (windows)
- Fonctionnement interne
  - Dans chaque répertoire il y a un répertoire .svn
    - entries : liste des fichiers gérés (edit, add, del)
    - text-base : contient des copies des fichiers du repository
    - props : metadonnees associes aux fichiers du repertoire

82

SVN vocabulaires et principes

- **Dépôt/repository/projet sur le serveur** : version du projet stocké sur le serveur
  - Le dépôt est repéré par une adresse web (URL), par exemple http://svn.liris.cnrs.fr/ameyer/Pacman
  - on peut consulter le dépôt comme une page web classique
- **Copie de travail/répertoire de travail** : la version du projet qui se trouve en local sur votre machine
  - Chaque dév a une copie de travail
- **Commiter/pousser** : transferer les modifications effectuées en local vers le serveur
- **Updater** : mettre à jour la copie de travail avec la version du serveur

83

SVN De quoi a-t-on besoin ?

- Client en local
  - Windows

    http://subversion.tigris.org/servlets/ProjectDocumentList?expandFolder=91&folderID=91
  - Linux

    apt-get install subversion
- Serveur
  - En LIF7 durant les TD
    - vous récupérerez du code depuis le serveur 'svn.liris.cnrs.fr'. Vous ne pouvez pas commiter vos modifications sur svn.liris.cnrs.fr
    - Vous utiliserez le système de fichier en local comme serveur
  - En LIF7 durant le projet, utilisation de serveur gratuit sur le web
    - Googlecode
    - Berlios developper : https://developer.berlios.de/
    - …

84

Subversion : commandes de base

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

checkout (co) : Extrait une copie de travail a partir d'un dépôt.
usage : checkout URL[@REV]... [CHEMIN]

Subversion : commandes de base

- Création d'un projet
  - **Appel** : svn import [path] [-m description] URL
  - Ajoute une arborescence de fichier au repository depuis un répertoire non versionné
  - Attention : le répertoire local n'est pas versionné (il faut ensuite obtenir une copie de travail)
  - A ne faire qu'une fois par projet
- Exemple

$ cd …/Pacman
$ svn import -m 'Nouveau projet pacman' http ://svn.serveur.zz/…/Pacman
→ Ceci créer un répertoire ../Pacman sur le serveur en important tous les fichiers se trouvant dans le répertoire local

Subversion : commandes de base

- Récupération d'une copie de travail en local
  - **Appel** : svn checkout URL [PATH]
  - Récupère une copie de travail en local du projet dont est l'URL est passée en paramètre
  - En principe, à ne faire qu'une fois par machine (pour la création des répertoires de travail en local) !
  - Ensuite on utilisera que svn update et svn commit
- Exemple

$ svn checkout http ://svn.serveur.zz/.../Pacman
A Terrain.c
A Terrain.h
…

Subversion : commandes de base

- Envoie des modifications
  - **Appel** : svn commit [path] [-m description]
  - Envoie/pousse les modifications du répertoire de travail vers le dépôt
  - Incrémente le numéro de révision du repository
  - Echou si la copie de travail n'est pas à jour par rapport au repository (cad qu'il y a des conflits à résoudre d'abord)
  - Opération atomique : en cas de problème, le repository n'est pas modifié
- Exemple

$ vi src/terrain.h && vi src/Terrain.c
$ svn commit –m "Terrain.c.h: ajout de la fonction d'affichage"
sending content src/Terrain.h
sending content src/Terrain.c
Completed: At revision: 2142

Subversion : commandes de base

- Mise à jour des fichiers locaux
  - **Appel** : svn update [path]
  - Récupère en local la dernière révision du repository
  - Affiche les modifications apportés aux fichiers (Added, Deleted, Updated, Conflict, merGed)
- Exemple

$ svn update
U Terrain.c
→ signifie que le fichier local Terrain.c est mis à jour ('updater') avec la version du dépôt, éventuellement avec fusion des modifications

Subversion : commandes de base

- Etat d'une copie locale
  - **Appel** : svn status [path]
  - Affiche l'état des fichiers du répertoire local
  - Utilise le .svn pour analyser l'état du répertoire local (modifié,ajouté, deleté, conflit, locké)
- Exemple

$ svn status
M src/Terrain.c // signifie que le fichier est modifié par rapport au dépôt

Subversion : commandes de base

- Résolution de conflit
  - Fichier monfichier.c est modifié en local
  - un autre développeur a également modifié ce fichier et a déjà commité ses modifications sur le serveur
  - svn update : on souhaite récupérer la dernière version

→ 2 cas possible

a) svn sait fusionné les 2 modifications

quand elles ont eu lieu a un endroit différent du fichier

b) svn ne sait pas fusionné

car les modifications ont eu lieu au même endroit du fichier

→ monfichier.c est placé en status 'conflit'

- Correction du conflit avec un éditeur texte
- svn resolved monchier.cpp

→informe SVN que le conflit est résolu

91

Subversion : commandes de base

Ajout / suppression de fichier

- **Appel** : svn add [path]
  - le fichier est noté pour ajout lors du prochain commit
  - Fonctionne récursivement
- **Appel** : svn rm [path]
  - le fichier est noté pour delete lors du prochain commit.
  - Fonctionne récursivement
  - Ne delete pas les fichiers modifiés en local
- **Appel** : svn mv src dest
  - déplace un fichier/répertoire et note le renommage pour le prochain commit
  - Conserve l'historique

92

SVN : la base, un exemple de scénario

1. Création du répertoire Pacman avec les 1er fichiers source
   a) cd ~bob
   b) mkdir Pacman ; mkdir Pacman/src ; mkdir Pacman/doc …
   c) emacs Pacman.c Pacman.h Terrain.c Terrain.h Makefile …
2. Création du projet Pacman sur le serveur
   a) cd ~bob/Pacman
   b) svn import http://svn.serveur.zz/bob/Pacman
   →ceci créer le projet Pacman sur le serveur en copiant tous les fichiers et répertoire de ~bob/Pacman sur le serveur
3. Récupère en local une copie de travail de Pacman
   a) rm –r ~bob/Pacman // plus besoin du pacman local initial
   b) svn checkout http://svn.serveur.zz/bob/Pacman
   → copie le projet Pacman du serveur sur le disque local
   → ceci est à faire par tous les développeurs

93

4. dévA change la fonction *terAfficher* dans terrain.c et pousse ses modifs sur le serveur
   a) emacs terrain.c → modifie la fonction *terAfficher*
   b) svn commit –m "nouvelle fonctionnalité dans terrain"
   → La version sur le serveur comporte les modifs du dévA et incrémente le numéro de version

5. dévB change la fonction *terSauver* et veut récupérer les modifs de dévA
   a) emacs terrain.c → modifie la fonction *terSauver*
   b) svn update
   → récupère les modifications du dévA qui se fusionne/merge automatiquement avec ses modifs (en local). DévB vérifie que tout compile encore.
   b) svn commit –m "nv fonction de sauvegarde de terrain"
   → La version sur le serveur comporte les modifs du dévB et incrémente le numéro de version

94

VARIANTE AVEC CONFLIT

4. dévA change … idem au transparent précédent

5. dévB change la fonction *terAfficher* comme le dévA
   a) emacs terrain.c → modifie la fonction *terAfficher*
   b) svn update
   → récupère les modifications du dévA mais SVN ne sait pas comment fusionner les modifs de dévA et dévB. Le fichier terrain.c est placé en mode **conflit**
   c) emacs terrain.c → le dévB fusionne les modifs à la main
   d) svn resolved terrain.c
   → indique à SVN que le conflit est levé pour ce fichier
   d) svn commit –m "nv fonction de sauvegarde de terrain"
   → La version sur le serveur comporte les modifs du dévB et incrémente le numéro de version

95