

Que fait ce code ?

#QDLE#Q#A*B#90#

```
int *f() {
    static int count=0;
    int n;
    int *t;
    count++;
    if ((scanf("%d",&n) != 1) || (n == -1)) {
        t = (int *)malloc(sizeof(int)
*count);
        t[count-1] = -1;
        count--;
        return t;
    }
    t = f();
    count--;
    t[count] = n;
    return t;
}
```

- f est une fonction récursive ?
 - A. oui
 - B. non

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int *t;  
    count++;  
    if ((scanf("%d",&n) != 1) || (n == -1)) {  
        t = (int *) malloc(sizeof(int)  
*count);  
        t[count-1] = -1;  
        count--;  
        return t;  
    }  
    t = f();  
    count--;  
    t[count] = n;  
    return t;  
}
```

count = 0

pile

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int *t;  
    count++;  
    if ((scanf("%d",&n) != 1) || (n == -1)) {  
        t = (int *)malloc(sizeof(int)  
*count);  
        t[count-1] = -1;  
        count--;  
        return t;  
    }  
    t = f();  
    count--;  
    t[count] = n;  
    return t;  
}
```

count = 1

pile

f (call 0)
n=10
t=?

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int *t;  
    count++;  
    if ((scanf("%d",&n) != 1) || (n == -1)) {  
        t = (int *)malloc(sizeof(int)  
*count);  
        t[count-1] = -1;  
        count--;  
        return t;  
    }  
    t = f();  
    count--;  
    t[count] = n;  
    return t;  
}
```

count = 2

pile

f (call 0)
n=10
t=?

f (call 1)
n=4
t=?

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int *t;  
    count++;  
    if ((scanf("%d",&n) != 1) || (n == -1)) {  
        t = (int *)malloc(sizeof(int)  
*count);  
        t[count-1] = -1;  
        count--;  
        return t;  
    }  
    t = f();  
    count--;  
    t[count] = n;  
    return t;  
}
```

count = 3

pile

f (call 0)
n=10
t=?

f (call 1)
n=4
t=?

f (call 2)
n=-1
t=?

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int *t;  
    count++;  
    if ((scanf("%d",&n) != 1) || (n == -1)) {  
        t = (int *)malloc(sizeof(int)  
*count);  
        t[count-1] = -1;  
        count--;  
        return t;  
    }  
    t = f();  
    count--;  
    t[count] = n;  
    return t;  
}
```

count = 3



pile

f (call 0)
n=10
t=?

f (call 1)
n=4
t=?

f (call 2)
n=-1
t=

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int *t;  
    count++;  
    if ((scanf("%d",&n) != 1) || (n == -1)) {  
        t = (int *)malloc(sizeof(int)  
*count);  
        t[count-1] = -1;  
        count--;  
        return t;  
    }  
    t = f();  
    count--;  
    t[count] = n;  
    return t;  
}
```

count = 1



pile

f (call 0)
n=10
t=?

f (call 1)
n=4
t=

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int *t;  
    count++;  
    if ((scanf("%d",&n) != 1) || (n == -1)) {  
        t = (int *) malloc(sizeof(int)  
*count);  
        t[count-1] = -1;  
        count--;  
        return t;  
    }  
    t = f();  
    count--;  
    t[count] = n;  
    return t;  
}
```

count = 0

10	4	-1
----	---	----

pile

f (call 0)
n=10
t=

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int  
    count  
    if (  
        t=  
    *count  
        t  
        co  
        re  
    }  
    t=f();  
    count--;  
    t[count]=n;  
    return t;  
}
```

Problèmes ?

count = 0

10	4	-1
----	---	----

pile

f (call 0)
n=10
t=

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int  
    count  
    if (  
        t=  
    *count  
        t  
        co  
        re  
    }  
    t=f();  
    count--;  
    t[count]=n;  
    return t;  
}
```

Problèmes ?
taille de la pile

count = 0

10 4 -1

pile

f (call 0)
n=10
t=

Que fait ce code ?

```
int *f() {  
    static int count=0;  
    int n;  
    int  
    coun  
    if (  
        t=  
    *count  
        t  
        co  
        re  
    }  
    t=f();  
    count--;  
    t[count]=n;  
    return t;  
}
```

Problèmes ?

taille de la pile

ré-entrance (variable statique)

count = 0

10	4	-1
----	---	----

pile

f (call 0)
n=10
t=

The compiling trivia

#QDLE#Q#ABCD*E#25#

- Quelle commande permet de compiler le fichier toto.c en objet ?
 - A. gcc toto.c
 - B. gcc -E toto.c
 - C. gcc -S toto.c
 - D. gcc -c toto.c
 - E. gcc -fPIC toto.c



The compiling trivia

#QDLE#Q#AB*CD#25#

- Quelle commande permet de récupérer le source après l'étape de pré-compilation ?
 - A. gcc toto.c
 - B. gcc -E toto.c
 - C. gcc -S toto.c
 - D. gcc -c toto.c



The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et main.o en un exécutable ?
 - A. gcc toto.o main.o
 - B. gcc -shared toto.o main.o
 - C. gcc -static toto.o main.o
 - D. gcc -exec toto.o main.o



The compiling trivia

#QDLE#Q#ABCD*#25#

- Quelle commande permet de lier toto.o et main.o en une bibliothèque statique?
 - A. gcc -shared -o libfoo.a toto.o main.o
 - B. gcc -static -o libfoo.a toto.o main.o
 - C. gcc toto.o main.o -lfoo
 - D. ar rcs libfoo.a toto.o main.o



The compiling trivia

#QDLE#Q#ABCDEF*G#30#

- Pour pouvoir regrouper des objets (.o) dans une bibliothèque dynamique, il faut les compiler avec la ou les options :
 - A. -c
 - B. -E
 - C. -s
 - D. -fPIC
 - E. réponse A&B
 - F. réponse A&D
 - G. réponse B&E



The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et main.o en une bibliothèque dynamique?
 - A. gcc -shared -o libfoo.so toto.o main.o
 - B. gcc -dynamic -o libfoo.a toto.o main.o
 - C. gcc toto.o main.o -lfoo
 - D. ar rcsD libfoo.so toto.o main.o



Automatisation

- La compilation manuelle n'est pas possible sur un grand projet :
 - homogénéisation des options de compilation
 - gestion des dépendances (que doit-on recompiler?)
 - commandes et options spécifiques à une plateforme.
 - erreurs manuelles
 - ...
- On doit se munir d'outil pour automatiser cette étape.

Makefile

- make est un outil qui permet d'automatiser la création et la mise à jour de fichiers.
- make repose sur un fichier de description : **Makefile**
- Un Makefile est un ensemble de règles formatées :

```
cible : source1 source2 source3 ...
```

```
    commande1
```

```
    commande2
```

```
... .
```

- Si « cible » n'existe pas ou est moins récente que l'une des sources, alors les commandes sont exécutées.

Makefile : exemple

```
image_thumbnail.jpg : image.jpg
    convert -size 80x80 image.jpg image_thumbnail.jpg
image.jpg : image.png
    convert image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

- make est récursif : si un fichier source n'existe pas, il cherche une règle pour le créer.
- initialement, make cherche à créer une seule cible : la première du Makefile ou celle(s) indiquée(s) sur la ligne d'appel : `make image.jpg`
- Si une source est manquante et qu'il n'y a pas de règle pour la créer : erreur
- Si une des commandes renvoie une valeur différente de 0, make s'arrête (tips : `commande || true`)

Makefile : variable

- make supporte la déclaration de variable :

NOM=VALEUR

- valeur peut comporter des espaces

NOM - =VALEUR

- Si NOM est dans l'environnement, alors c'est la valeur de l'environnement qui est utilisée, sinon c'est VALEUR.
- \$(NOM) est remplacé par VALEUR.

Makefile : variable, exemple

```
CONVERT=convert
THUMB_SIZE-=80x80
image_thumbail.jpg : image.jpg
    $(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
image.jpg : image.png
    $(CONVERT) image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

\$ THUMB_SIZE=60x60 make

Makefile : spéciales

- Quelques variables spéciales pour l'écriture des commandes :

`cible : source1 source2`

- `$@` : cible
- `$<` : source1
- `$$` : source1 source2

- Ainsi :

```
image_thumbail.jpg : image.jpg
$(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
```

- Devient :

```
image_thumbail.jpg : image.jpg
$(CONVERT) -size $(THUMB_SIZE) $< $@
```

Makefile : règles génériques

- Il est possible d'écrire des règles génériques :

```
%.jpg : %.png
```

```
    convert $< $@
```

- Permet de convertir tout fichier png en jpg à l'aide de convert

```
%_thumb.jpg : %.jpg
```

```
    convert -size 80x80 $< $@
```

- Il est enfin possible de séparer la liste des dépendances et la règle de création.

Makefile et compilation

```
CC=gcc
CFLAGS=-Wall

prog : prog.o hash.o
    $(CC) -o $@ $^
prog.o : prog.c hash.h
hash.o : hash.c hash.h
%.o :
    $(CC) $(CFLAGS) $< -o $@
```

- La dernière règle explique comment générer un fichier .o
- les deux règles au dessus donnent les dépendances

Makefile : PHONY

- On peut vouloir écrire des règles qui ne génèrent pas de fichier :

`all`

`install`

`clean`

`distclean`

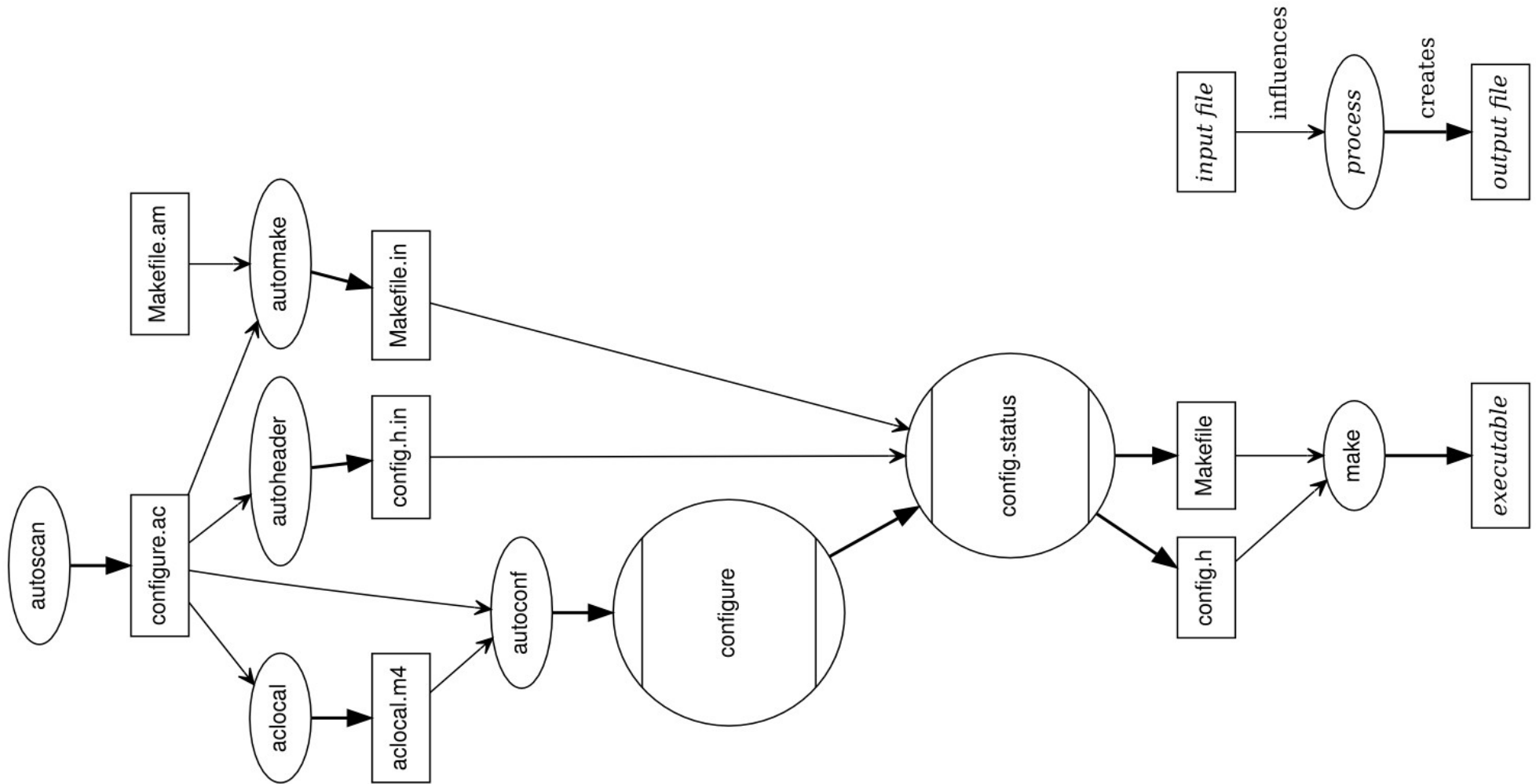
- On indique cela à make :

```
.PHONY=all install clean distclean
```

Makefile, automake...

- Makefile n'est pas spécifique à la compilation de programme
- Il ne gère pas, entre autre, la dépendance entre les fichiers sources (on peut utiliser `gcc -MM source1.c source2.c ...` pour cela)
- La prise en compte de l'environnement (compilateur, options, bibliothèque...) peut ce faire :
 - par l'édition des variables du Makefile (options de compilations, répertoire d'installation...)
 - par l'écriture de plusieurs Makefile (un par système)
 - par l'utilisation d'un générateur :
 - automake / autoconf
 - cmake

auto-tools : automake / autoconf



cmake

- Cmake est un outil simplifié permettant la compilation de sources C et C++.
- C'est un outil multiplateforme sous licence BSD
- Nécessite la présence d'un fichier CMakeLists.txt
- gestion automatique des dépendances
- peut générer des makefile
- Simple d'utilisation / facile à prendre en main
- Exemple: un projet <<HELLO>> avec une bibliothèque dans le répertoire Hello et un programme d'exemple dans le répertoire Demo

cmake

- `./CMakeLists.txt`:

```
cmake_minimum_required (VERSION 2.6)
project (HELLO)
```

```
add_subdirectory (Hello)
add_subdirectory (Demo)
```

- `./Hello/CMakeLists.txt`

```
add_library (Hello hello.c)
```

- `./Demo/CMakeLists.txt`

```
include_directories (${HELLO_SOURCE_DIR}/Hello)
link_directories (${HELLO_BINARY_DIR}/Hello)
add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo Hello)
```

cmake:cross platform make

- cmake est un système de compilation cross-platform. Il ne compile pas directement mais génère des fichiers dans différents formats :
 - Makefile
 - projet Visual Studio
 - Borland Makefile
 - projet Xcode
 - Kate
 - ...
- cmake utilise les fichiers CMakeLists.txt et génère des fichiers en fonction de la plate-forme de compilation (Makefile, visual, xcode....).

cmake : les bases

- La déclaration de variables :
 - `set(NAME VALUE)`
 - `${NAME}`
 - lors de l'appel à cmake : `cmake -DNAME=VALUE`
- Variables standards :
 - `CMAKE_INCLUDE_PATH` (pour les .h)
 - `CMAKE_LIBRARY_PATH` (pour la recherche de .so)
 - `DESTDIR` (pour l'installation)
 - `CMAKE_BUILD_TYPE` (Debug, Release)
- Dans le CMakeLists.txt :

– <code>CMAKE_C_FLAGS</code>	– <code>CMAKE_CURRENT_SOURCE_DIR</code>
– <code>CMAKE_C_FLAGS_DEBUG</code>	– <code>CMAKE_CURRENT_BINARY_DIR</code>
– <code>CMAKE_C_FLAGS_RELEASE</code>	– <code>CMAKE_SOURCE_DIR</code>

cmake : les bases (2)

- `add_executable(name sources)`
- `add_library(name STATIC sources)`
- `add_library(name SHARED sources)`
- `target_link_libraries(name libs)`
- `include_directories(dir1 dir2...)`
- `add_custom_command`

cmake : utilisation

- cmake supporte l'out-source building : c'est à dire la compilation en dehors du répertoire des sources :
- On suppose : projet/CMakeLists.txt
- alors on peut faire :

```
mkdir projet-build
```

```
cmake ../projet
```

```
make
```

- et

```
mkdir projet-debug
```

```
cmake -DCMAKE_BUILD_TYPE=Debug ../projet
```

```
make
```

Question

#QDLE#Q#AB*#20#

- cmake analyse les dépendances :
 - A. en énumérant les fonctions appelées
 - B. en traçant les inclusions

Question

#QDLE#Q#A*BCD#35#

- J'ai une bibliothèque dynamique libtoto.so compilée à partir de toto.c et toto.h. J'ai un programme de démo demo.c qui utilise cette bibliothèque. L'ensemble est compilé. Si je modifie toto.c je dois :
 - A. re-compiler la bibliothèque
 - B. re-compiler l'exécutable
 - C. réponse A & B
 - D. ne rien faire.

Question

#QDLE#Q#ABC*D#25#

- J'ai une bibliothèque dynamique libtoto.so compilée à partir de toto.c et toto.h. J'ai un programme de démo demo.c qui utilise cette bibliothèque. L'ensemble est compilé. Si je modifie toto.h je dois :
 - A. re-compiler la bibliothèque
 - B. re-compiler l'exécutable
 - C. réponse A & B
 - D. ne rien faire.

Convention de codage

- Une convention de codage est un document qui liste les règles d'écriture de code source pour un projet / une entreprise :
 - nommage des fonctions, variables, macro...
 - nommage et organisation de fichiers
 - langue pour le code et les commentaires
 - formatage spécifique (boucle, tests...)
 - techniques de programmation
- Une convention est un document vivant qui doit être mis à jour si nécessaire.

convention de codage : exemple

- <https://www.kernel.org/doc/Documentation/CodingStyle>
 - indentation
 - taille max de lignes
 - positionnement des accolades et parenthèses
 - espaces
 - nommage : C is a Spartan language, and so should your naming be....
 - typedef
 - fonctions
 - sortie de fonctions
 - commentaires

Documentation

- La documentation est primordiale pour un projet sur le long terme
- Plusieurs niveaux de doc :
 - très haut niveau : présentation large, vue d'ensemble, éléments d'architecture
 - modules : aspects fonctionnels, périmètre
 - fonction : description des paramètres, valeurs de retour, spécification
 - code : astuces mises en œuvre, point d'algorithmique non trivial, justification de choix.

Documentation

- Le maintien d'une documentation peut être un travail long et il arrive souvent qu'il y a divergence entre la documentation et le code.
- Pour cela, on rapproche la documentation du code en l'incluant dans celui-ci
- Utilisation des commentaires et de générateurs de documentation
- Ne permet de faire toute la documentation (en particulier, la doc de haut niveau, manuel d'utilisation...).

doxygen

- **doxygen** permet de générer la documentation au format html/pdf/latex... à partir des commentaires dans le code source.
- doxygen peut également intégrer de la documentation au format **Markdown**
- Le résultat est une documentation séparée des sources mais synchronisée avec celles-ci.

doxygen

- doxygen utilise des commentaires suivant certaines règles d'écriture :

```
///  
//! power function
```

```
/*!  
The pow() function returns the value of x raised to the power  
of y.
```

```
* \param x a real in double format
```

```
* \param y a real in double format
```

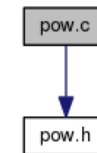
```
* \return x raised to power of y or NaN if wrong arguments
```

```
*/
```

pow.c File Reference

#include "pow.h"

Include dependency graph for pow.c:



[Go to the source code of this file.](#)

Functions

double [pow](#) (double x, double y)

Function Documentation

```
double pow ( double x,  
             double y  
             )
```

power function The [pow\(\)](#) function returns the value of x raised to the power of y.

Parameters

x a real in double format

y a real in double format

Returns

x raised to power of y or NaN if wrong arguments

Definition at line [5](#) of file [pow.c](#).

doxygen et README.md

- Il est souhaitable pour un projet d'avoir un fichier qui donne des informations générales de haut niveau :
 - auteurs
 - objectif de l'application
 - pré-requis
 - installation
 - utilisation

```
My great project      {#mainpage}  
=====
```

● About

Une technique consiste à intégrer ces éléments dans un fichier README.md à la racine du

My Project

Main Page

Files

▼ My Project



My great project

► Files

My great project

Author

Julien Allali

About

example for PG106 course.

Commentaires : qq règles

- Les commentaires doivent être **utiles**
- Pour une fonction :
 - ce que fait la fonction (spécification)
 - éventuellement, comment elle le fait (algo, complexité, coût mémoire...)
 - domaine de valeur des paramètres
 - cas d'erreurs
- Les commentaires dans le code doivent servir à suivre la logique de celui-ci, par ex :
 - `// set default value into the matrix`
 - `///`

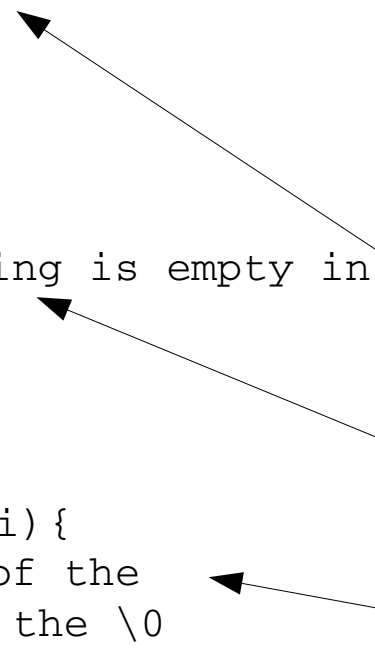
Commentaires : qq règles

- Pour les modules, penser à ajouter une description générale de ce que fait le module, avec un code d'exemple d'utilisation.
- Au début des fichiers d'implémentation, un entête spécifie :
 - les auteurs,
 - la licence, (copyright si rien)
 - une liste datée des modifications

commentaires

#QDLE#Q#A*BC#30#

```
int my_function(int arg){
    // set a counter to 0
    int counter=0 ;
    ... ..
    ... ..
    if (x==0){
        // because the string is empty in
        //this case
    }
    ...
    ...
    for(i=0;i<counter-1;++i){
        // parse the char of the
        // string avoiding the \0
        ... .
    }
    ... .
}
```



The diagram shows three arrows originating from labels A, B, and C on the right side of the slide. Arrow A points to the comment '// because the string is empty in //this case'. Arrow B points to the comment '// set a counter to 0'. Arrow C points to the comment '// parse the char of the // string avoiding the \0'.

- sur cet exemple, quel commentaire est sans intérêt ?

– A

– B

– C

gestion de sources

- Lorsque l'on interagit avec d'autres développeurs, il est indispensable de pouvoir communiquer des propositions de modifications (ajout de fonctionnalité, correctif de bug, amélioration des perfs...) :

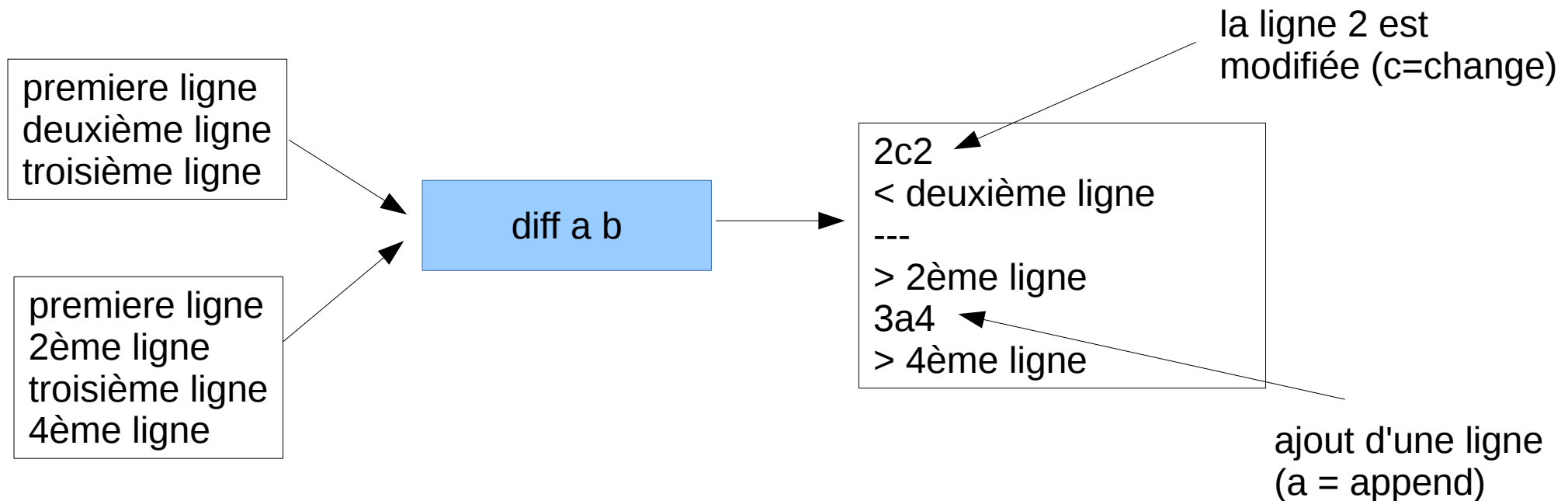
gestion de sources

#QDLE#S#ABC#25#

- Lorsque l'on interagit avec d'autres développeurs, il est indispensable de pouvoir communiquer des propositions de modifications (ajout de fonctionnalité, correctif de bug, amélioration des perfs...) :
 - A) j'envoie tout le code en indiquant que c'est une nouvelle version.
 - B) j'écris un email détaillé des modifications à effectuer.
 - C) autre approche...

diff

- **diff** est un outil d'analyse de texte qui compare deux fichiers entre eux et produit le nombre minimum d'édition à faire sur le premier fichier pour obtenir le second :



diff : side by side

- on peut afficher les deux fichiers cote à cote :

```
diff -y a b
```

```
premiere ligne  
deuxième ligne  
troisième ligne
```

```
premiere ligne  
| 2ème ligne  
troisième ligne  
> 4ème ligne
```

- diff permet la comparaison récursive de deux arborescences.

diff : recursif

- Je souhaite modifier le code source d'un projet :
 1. je fais une copie de sauvegarde des sources d'origine
 2. j'effectue mes modifications
 3. à tout moment, je visualise mes modifications avec
`diff : diff -r projet projet_new`

diff : recursif

- Je souhaite modifier le code source d'un projet :
 1. je fais une copie de sauvegarde des sources d'origine
 2. j'effectue mes modifications
 3. à tout moment, je visualise mes modifications avec
`diff : diff -r projet projet_new`

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLib_new/TaskParser.cpp
15c15
<     throw gtl::InvalidArgumentException("json parser error: expected a dictionnary at each task");
---
>     throw gtl::InvalidArgumentException("json parser error: expected a dictionnary for each task");
diff -r GenTaskLib/TaskParser.hpp GenTaskLib_new/TaskParser.hpp
36a37,38
>  /*! TaskParser: build a task from a json description.
>  */
```

Ajout d'un commentaire

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

MOTELS ARE NOT HELL!

MIROIR TU ES LA!

- Sur cet exemple « MO T » est une sous séquence commune.

diff : algorithme

#QDLE#Q#ABC*D#45#

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

MOTELS ARE NOT HELL!

MIROIR TU ES LA!

- Taille de **la plus longue** sous séquence commune ? (les espaces comptent).
A. 6 B. 8 C. 9 D. 10

diff : algorithme

- Le programme diff repose sur un problème classique d'algorithmique du texte : La plus longue sous-séquence commune

MOTELS_ARE_NOT_HELL!

MIROIR_TU_ES_LA!

MORTEL!

diff : algorithm

- Chaque fichier est découpé en ligne
- Les lignes sont comparées entre elles (LCS entre chaque ligne)
- Puis l'ensemble des lignes sont comparées entre elles (LCS où chaque symbole représente une ligne).

diff et communication

- Ainsi, si l'on souhaite communiquer une modification, il suffit d'envoyer le résultat d'un diff récursif : `diff -rupN original new > patch`
- On appelle ce fichier un patch.
- Le destinataire peut lire ce fichier et comprendre vos modifications
- Il peut également appliquer ces modifications en local grâce au programme `patch`
- *les options upN sont nécessaires au fonctionnement de patch, elles*

patch

- le programme patch permet d'appliquer les modifications identifiées par diff.
- Ainsi sur l'exemple précédent je peux faire :

```
$ cp -R GenTaskLib GenTaskLibMod
```

```
$ cd GenTaskLibMod
```

```
$ patch < ../patch
```

```
patching file TaskParser.cpp
```

```
diff -r GenTaskLib/TaskParser.cpp GenTaskLibMod/TaskParser.cpp
```

```
15c15
```

```
<     throw gtl::InvalidArgumentException("json parser error: expected a dictionary at each task");
```

```
---
```

```
>     throw gtl::InvalidArgumentException("json parser error: expected a dictionary for each task");
```

```
diff -r GenTaskLib/TaskParser.hpp GenTaskLibMod/TaskParser.hpp
```

```
36a37,38
```

```
>  /*! TaskParser: build a task from a json description.
```

```
>  */
```

diff & patch

- Dans le monde open source, diff et patch sont extrêmement utilisés

Bugzilla@Mozilla

New Account | Log In | Forgot Password

Home New Browse Search [help] Reports Product Dashboard

Bug List: (1 of 307) First Last Prev Next Show last search results

Bug 328174 - ISP files: can't preselect server type choice [Last Comment](#)

Status: REOPENED
Whiteboard:
Keywords: fixed1.8.1
Product: Thunderbird ([show info](#))
Component: Account Manager ([show other bugs](#)) ([show info](#))
Version: unspecified
Platform: All All
Importance: -- normal (vote)
Target Milestone: ---
Assigned To: Yann Rouillard
QA Contact:
Mentors:
URL:
Depends on:
Blocks: [Show dependency tree / graph](#)

Reported: 2006-02-22 02:29 PST by Yann Rouillard
Modified: 2009-11-11 19:21 PST ([History](#))
CC List: 4 users ([show](#))
See Also:
Crash Signature:
Project Flags:
Tracking Flags:

Attachments

Attachment	Flags	Details
ISP example file to test the bug. (2.25 KB, application/rdf+xml) 2006-02-22 02:31 PST, Yann Rouillard	no flags	Details
Patch to preselect imap in server page if server type was set to imap in isp rdf file (11 KB, patch) 2006-02-22 02:32 PST, Yann Rouillard	mozilla: review-	Details Diff Review
cumulative patch (2.73 KB, patch) 2006-02-22 10:57 PST, David Bienvenu	mozilla: review+ mscott: superreview+ mscott: approval-branch-1.8.1+	Details Diff Review
Allow isp rdf to force use of incoming username for outgoing server (4.65 KB, patch) 2006-04-10 13:00 PDT, Yann Rouillard	mozilla: review+ mkmelin+mozilla: superreview-	Details Diff Review
rdf file to test smtpUseIncomingUsername (2.32 KB, application/rdf+xml) 2006-04-11 01:07 PDT, Yann Rouillard	no flags	Details
Add an attachment (proposed patch, testcase, etc.)		Show Obsolete (1) View All

liste de 3 patchs correctifs

gestion de source

- Un gestionnaire de source permet de conserver l'historique des modifications apportées à un ensemble de fichier.
- Il permet à un ensemble d'utilisateurs d'interagir sur un même code source.
- Tous les gestionnaires de codes sources sont basés sur les principes de diff et patch
- Il existe deux grandes catégories de gestionnaires :
 - les centralisés
 - les décentralisés

Les gestionnaires centralisés

- Historiquement, cvs (concurrent versioning system) et son successeur svn (subversion)
- Les gestionnaires centralisés reposent sur un serveur central qui archive toutes les modifications apportées au code.
- Chaque modification incrémente un numéro de révision
- Les commandes de base de svn :
 - checkout, update, infos, status, commit, diff, revert

svn

- Chaque utilisateur interagit avec le serveur, il n'y a pas d'échange direct entre deux utilisateurs.
- Une méthodologie est associée à l'utilisation de svn, vous retrouverez cette méthodologie dans tout projet de développement (open source ou entreprise).

svn : méthodologie

- Il est possible d'utiliser SVN juste pour le répertoire de développement principal.
- Seulement, comment faire si on a un « gros » développement à produire: si on transmet les modifications intermédiaires, le programme devient instable (ne compile plus par exemple...)
- Comment faire également pour gérer les versions:
 - On sort une version 1.0 qui évolue en 1.1 puis 1.2
 - On sort la version 2.0 (« casse » la compatibilité avec 1.x)

svn : méthodologie

- Aussi, il est nécessaire de maintenir plusieurs « répertoires » de développement parallèle.
- Une méthodologie classique organise les sources ainsi:
 - / « racine »
 - /trunk : contient la version en cours des sources (dev.)
 - /branches/: des copies de trunk (« svn copy »)
 - /branches/1.0 : copie de trunk pour release (tests), retour de modif dans le trunk avec « svn merge » si compatible
 - /tags/1.0.0: version **figée** d'une branche, sert de référence, est diffusée. La branche correspondante est étiquetée (tag). **pas de commit/modifs dans ce répertoire**

svn : la création de dépôt

- Le création d'un dépôt se fait à l'aide de la commande « `svnadmin create nom_de_depot` »
- possibilité d'ajouter l'exécution de script avant/pendant et après les « commits »
- possibilité d'envoi de mails lors des commits
- facile à mettre en place sur son compte:
- `cd ~/.depots/ ; svnadmin create SVN`
- `cd ~/; svn co file:/// $HOME/.depots/SVN`
- via ssh:
- `svn co`

Les gestionnaires dé-centralisés

- Dans ce cas, il n'y a pas de dépôt centrale.
- Chaque utilisateur gère son propre dépôt.
- Un protocole permet l'échange de modification (commit) entre deux utilisateurs.
- Exemple : git
- Commandes de base :
 - clone, add, commit, push, pull, checkout

git : les commits, pull et push

- Dans git, les commits sont locaux (il n'y a pas de dépôt centrale).
- On peut transmettre un ensemble de commit à un autre utilisateur avec la commande push
- On peut réceptionner un ensemble de commit depuis un autre utilisateur avec la commande pull.

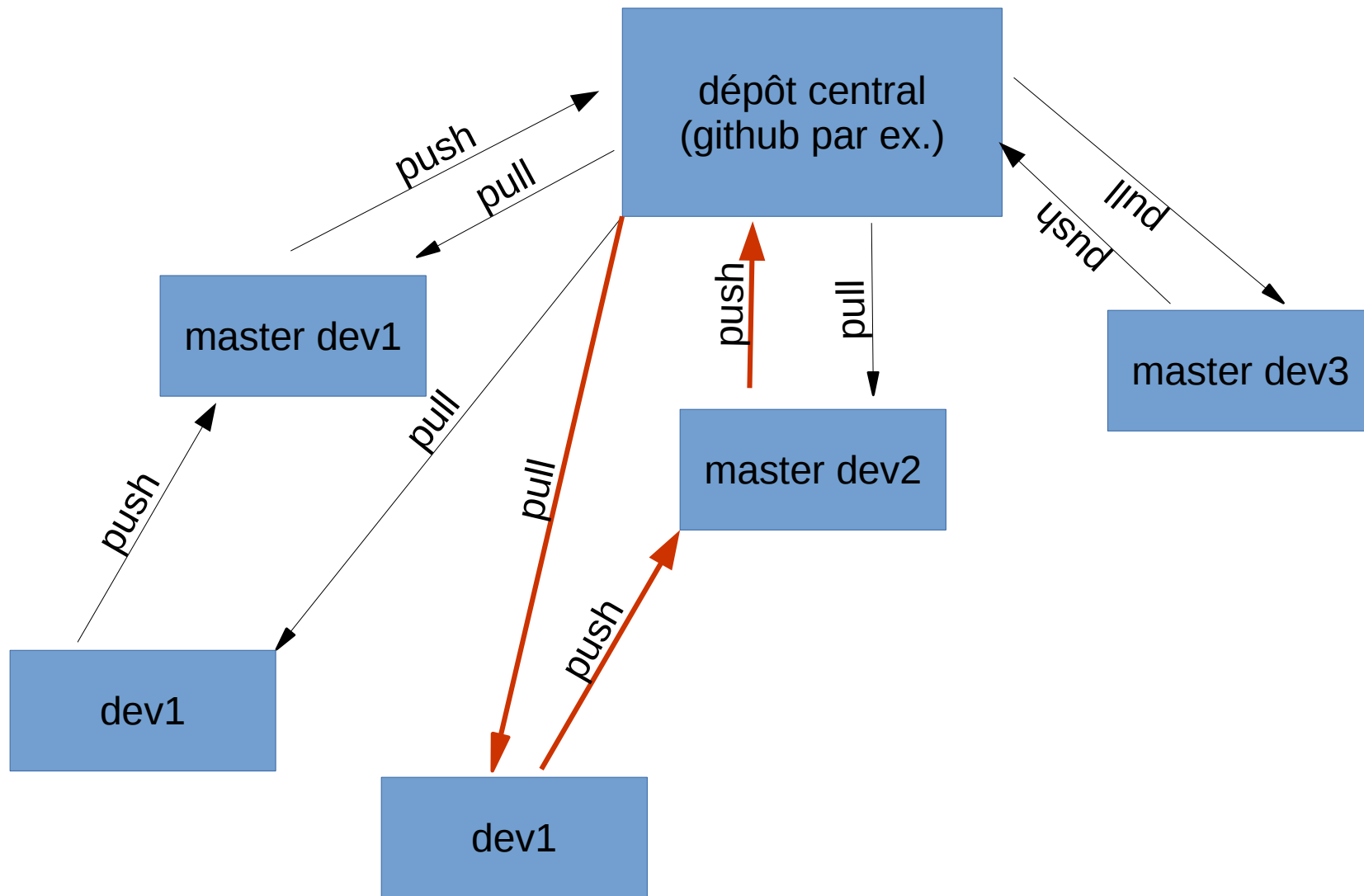
git

- git intègre une gestion de branches :
 - création :
 - git branch b2
 - git checkout b2
 - ou bien git checkout -b b2
 - la fusion :
 - on bascule dans la branche qui doit recevoir les modifs
 - git checkout master ; git merge b2
 - Les modifications doivent avoir été commité dans b2
 - la déléition : git branch -d b2
- La branche par défaut s'appelle **master**

re-centralisation

- L'avantage d'être en décentraliser et de pouvoir faire des « commit » sans connexion à un serveur.
- Pour la plus part de projet, il est cependant nécessaire d'avoir une référence : on utilise alors un dépôt git comme tel (github par exemple).
- On peut ensuite mettre en place un système de propagation hiérarchique des « commits ».

git



contrôle dans svn

- Il est aussi possible d'avoir cette approche hiérarchique dans svn en subdivisant le répertoire « branches » en répertoire et en ajustant les droits :
 - seuls les masters peuvent commiter dans « trunk »
 - les dev travaillent dans des branches
- Différences majeures entre svn et git est :
 - centralisé / dé-centralisé
 - support natif du système de branches dans git
 - commit locaux dans git

Les autres gestionnaires

	Open Source	Centralisé	Décentralisé
CVS	•	•	
SVN	•	•	
GIT	•		•
SourceSafe		•	
Mercurial	•		•
Bazaar	•		•
BitKeeper			•
Team Foundation Server		•	

Intégration Continue



L'Intégration Continue

- Lors que l'on développe, on est sur un système particulier :
 - type de système (unix, linux, windows, macosx, etc.)
 - version du compilateur
 - version des bibliothèques
 - environnement général (ressources...)
- Avant de transmettre une modification (commit), le développeur doit s'assurer que ses modifications fonctionnent pour l'ensemble des systèmes/configurations cibles.

L'Intégration Continue

- Pour cela, on dispose d'un ensemble de machines.
 - Solution 1 : avant de transmettre mes modifications, je me connecte sur chacune des machines et je testes.
 - Solution 2 : j'utilise un système qui fait cela automatiquement pour moi !
⇒ C'est ce que l'on appelle l'Intégration Continue.
- l'IC *garantie* une stabilité des développements au fur et à mesure. Cela permet de contrôler certaines dettes techniques.

L'Intégration Continue

- Il existe plusieurs plateformes d'intégration continue :
 - Jenkins (successeur de Hudson, java-open source)
 - TeamCity (JetBrain, commercial)
 - CruiseControl (java-open source)
 - Team Foundation Server (Microsoft, commercial)
 - Travis IC (online IC for github projects).
 - ...

L'Intégration Continue

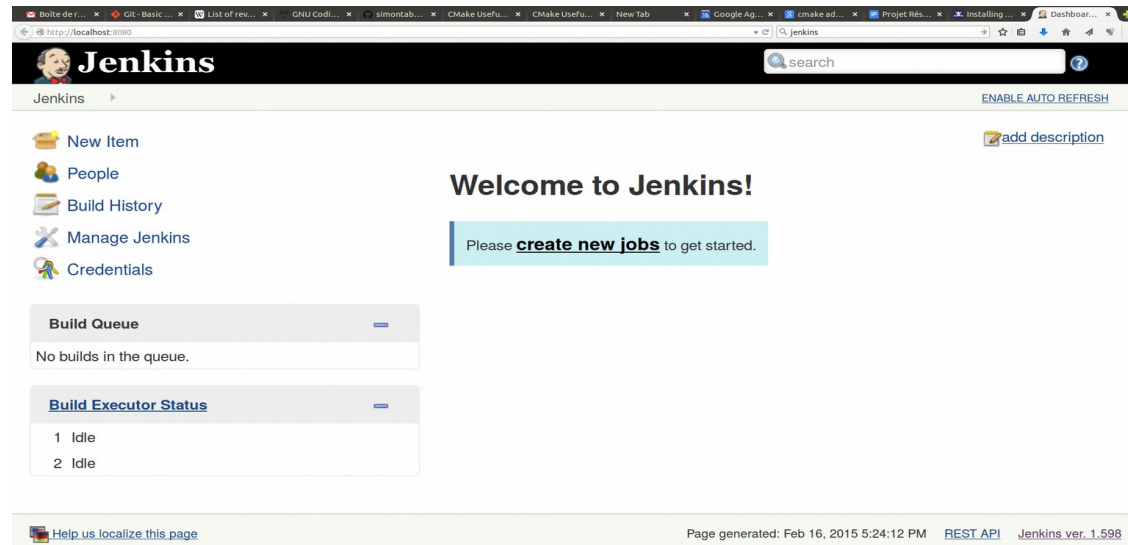
- Un serveur d'intégration continue va se synchroniser avec un dépôt
- A intervalle régulier, il va vérifier que le dépôt est à jour. Si une mise à jour est intervenue, il va effectuer une série de tâches (compilation par exemple).
- En fonction du résultat des tâches, le serveur va indiquer l'état du projet et possiblement transmettre des alertes.
- Afin de gérer plusieurs environnements, le serveur d'IC va piloter des clients sur lesquels il lancera les tâches (via ssh par exemple)

L'Intégration Continue

- La qualité qu'offre l'IC va dépendre principalement de deux facteurs :
 - la nature et la diversité des clients (environnement de validation)
 - la complexité des tâches à réaliser :
 - de la compilation
 - à l'exécution de tâches complexes de validation
- Le serveur d'IC peut également rendre compte de facteurs comme les ressources utilisées (cpu, temps, mémoire).

Exemple avec Jenkins

- répertoire projet :
 - projet/ :
 - makefile
 - main.c
- installation de jenkins et connexion au port 8080 sur localhost :



Jenkins : exemple

- création d'un dépôt local avec les sources :
svnadmin create /tmp/projet
checkout + ajout des sources + commit
- Ajout du dépôt dans Jenkins en utilisant comme url *svn+ssh://localhost/tmp/projet/*
- Ajout comme commande de build : make
- Lancement d'un build dans jenkins

Jenkins : statut du projet

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery-... x CMake Useful Variable... x CMake Useful Variable... x Projet [Jenkins] x Jenkins users - SVN ... x

http://localhost:8080/job/Projet/ jenkins subversion "file:///"

Jenkins

Jenkins ▶ Projet ▶ [ENABLE AUTO REFRESH](#)

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

Project Projet

[add description](#)





[Disable Project](#)



[Workspace](#)

[Recent Changes](#)

Build History

[trend](#)

 #4	Feb 16, 2015 5:39 PM
 #3	Feb 16, 2015 5:38 PM
 #2	Feb 16, 2015 5:32 PM
 #1	Feb 16, 2015 5:31 PM

 [RSS for all](#)  [RSS for failures](#)

Permalinks

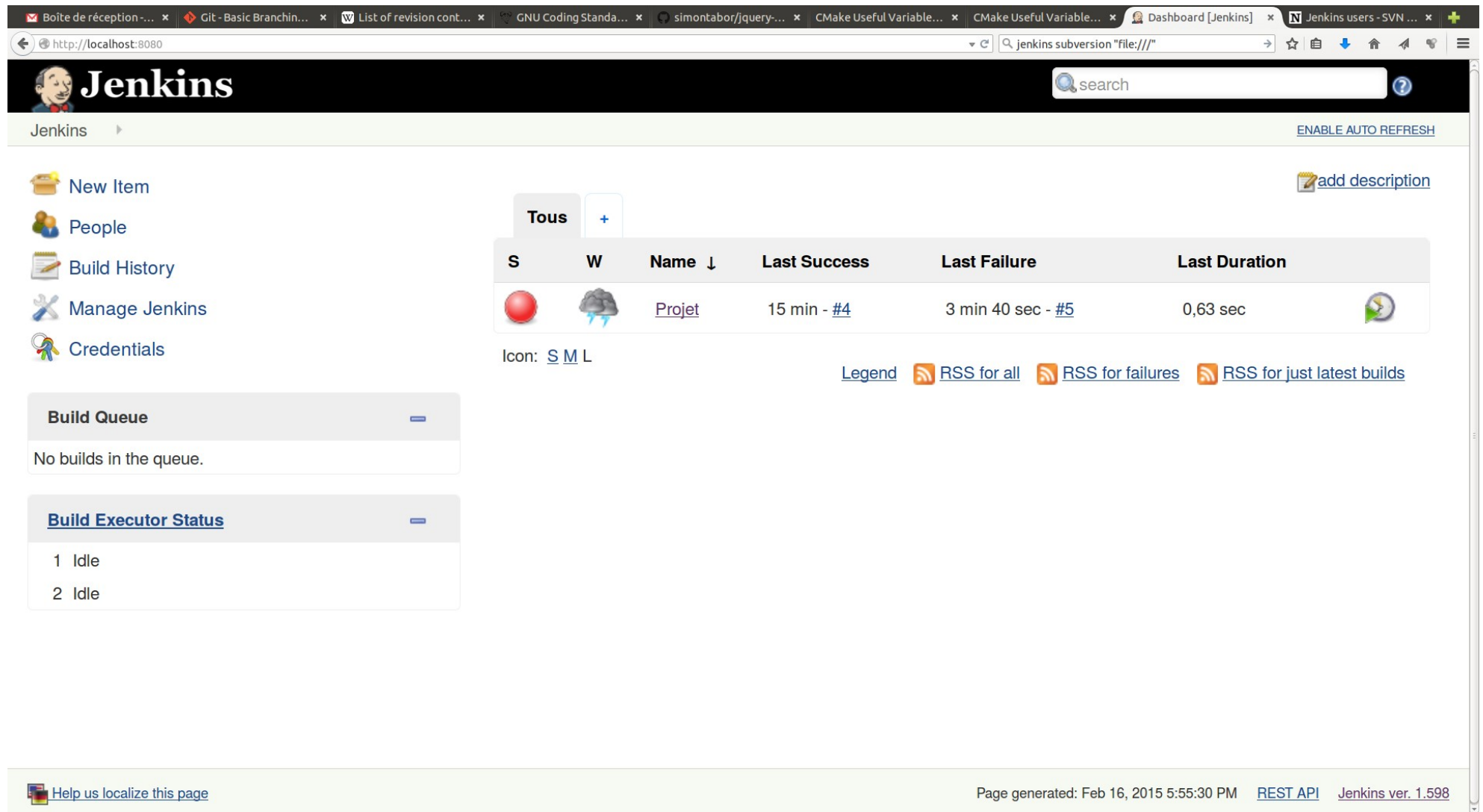
- [Last build \(#4\), 1 min 53 sec ago](#)
- [Last stable build \(#4\), 1 min 53 sec ago](#)
- [Last successful build \(#4\), 1 min 53 sec ago](#)
- [Last failed build \(#3\), 3 min 43 sec ago](#)
- [Last unsuccessful build \(#3\), 3 min 43 sec ago](#)

[Help us localize this page](#)

Page generated: Feb 16, 2015 5:41:46 PM [REST API](#) [Jenkins ver. 1.598](#)

Jenkins : commit

- Ajout d'un bug, commit dans le dépôt



The screenshot shows the Jenkins web interface. The top navigation bar includes the Jenkins logo, a search bar, and a link to 'ENABLE AUTO REFRESH'. The left sidebar contains links for 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. The main content area displays a table of builds. The table has columns for 'S' (Status), 'W' (Weather icon), 'Name', 'Last Success', 'Last Failure', and 'Last Duration'. A single build is listed with the name 'Projet', which is in a failed state (red circle icon). The 'Last Success' column shows '15 min - #4' and the 'Last Failure' column shows '3 min 40 sec - #5'. The 'Last Duration' is '0,63 sec'. Below the table, there are links for 'Icon: S M L' and 'Legend' with RSS feeds for 'all', 'failures', and 'latest builds'. On the left, there are sections for 'Build Queue' (showing 'No builds in the queue.') and 'Build Executor Status' (showing two idle executors).

S	W	Name ↓	Last Success	Last Failure	Last Duration
		Projet	15 min - #4	3 min 40 sec - #5	0,63 sec

Icon: [S](#) [M](#) [L](#)

Legend [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

Build Queue
No builds in the queue.

Build Executor Status

- 1 Idle
- 2 Idle

Page generated: Feb 16, 2015 5:55:30 PM [REST API](#) Jenkins ver. 1.598

Jenkins : bug

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery... x CMake Useful Variable... x CMake Useful Variable... x Projet #5 [Jenkins] x Jenkins users - SVN ... x

http://localhost:8080/job/Projet/lastFailedBuild/ jenkins subversion "file:///"


Jenkins

Jenkins ▸ Projet ▸ #5 [ENABLE AUTO REFRESH](#)


[Back to Project](#) [Status](#) [Changes](#) [Console Output](#) [Edit Build Information](#) [Delete Build](#) [Tag this build](#) [Previous Build](#)

Build #5 (Feb 16, 2015 5:51:49 PM)

Started 5 min 18 sec ago
Took [1 sec](#) [add description](#)

 Revision: 2
Changes

1. add a bug ([detail](#))

 Started by anonymous user

[Help us localize this page](#) Page generated: Feb 16, 2015 5:57:07 PM [REST API](#) Jenkins ver. 1.598

Jenkins : bug

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery-... x CMake Useful Variable... x CMake Useful Variable... x Projet #5 Console [... x Jenkins users - SVN ... x

http://localhost:8080/job/Projet/lastFailedBuild/console jenkins subversion "file:///"

Jenkins

Jenkins ▶ Projet ▶ #5

- Back to Project
- Status
- Changes
- Console Output**
- View as plain text
- Edit Build Information
- Delete Build
- Tag this build
- Previous Build

Console Output

```
Démarré par l'utilisateur anonymous
Building in workspace /var/lib/jenkins/workspace/Projet
Updating svn+ssh://localhost/tmp/projet at revision '2015-02-16T17:51:49.226 +0100'
U      main.c
At revision 2
[Projet] $ /bin/sh -xe /tmp/hudson9096975002011345535.sh
+ make
gcc -c main.c
main.c: In function 'main':
main.c:3:3: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'return'
    return 0;
    ^
Makefile:4: recipe for target 'main.o' failed
make: *** [main.o] Error 1
Build step 'Exécuter un script shell' marked build as failure
Finished: FAILURE
```

[Help us localize this page](#)

Page generated: Feb 16, 2015 5:57:38 PM [REST API](#) [Jenkins ver. 1.598](#)

Intégration Continue

- Tester que le programme compile sous plusieurs environnements est bien mais cela n'offre que peu de garantie quand à l'état fonctionnel du projet.
- Pour *garantir* une qualité tout au long du développement, il faut ajouter des **tests**

Les Tests



Les Tests

- Il existe de nombreux types de tests
- Les tests ont pour objectifs de valider votre code :
 - au niveau d'une fonction : tests unitaires
 - au niveau d'un module : tests fonctionnels
 - entre plusieurs modules : tests d'intégration
 - au niveau général, applicatif : tests de recette

TDD : test driven development

- La méthodologie TDD repose sur l'écriture d'abord de tests puis du code validant les tests.
- La méthode XP (extreme programming) repose en partie sur partie sur TDD.
- TDD repose sur des cycles courts consistant :
 - à écrire un test fonctionnel
 - vérifier que le test plante
 - à écrire un test unitaire
 - vérifier que le test plante
 - écrire le code minimal pour que le test fonctionne
 - vérifier que le test passe

TDD

- A la fin de l'écriture d'un code et lorsque tout les tests passent, on peut vouloir refactoriser votre code (copier/coller, simplification, unification, ...)
- Dans ce cas, on ne touche surtout pas aux tests et on remanie le code jusqu'à ce que à nouveau il valide l'ensemble des tests.

TDD par l'exemple : bowling

- Supposons que l'on souhaite écrire un module de calcul de feuille de score de bowling :



exemple grandement inspirée de la présentation « TDD in C » par Olve Maudal (dispo. slideshare)

bowling

- Le joueur a 10 sets
- Pour chaque set, le joueur lance la boule une ou deux fois:
 - Si le joueur élimine les 10 quilles du premier coup, il fait strike et il ne joue pas de 2ème boule
 - Si le joueur élimine les 10 quilles au deuxième coup, il fait spare
- Le score d'un set fait :
 - le score précédent + la somme des deux lancés si pas de strike ni de spare
 - le score précédent + 10 + le score du premier lancé du prochain set si spare

bowling

- La spécification client est d'avoir un module BowlingGame avoir les méthodes suivantes
 - void roll(BowlingGame *,int nbPinsDown) : enregistre un nouveau lancé
 - int score(BowlingGame *) : renvoie le score actuel

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
 - struct Game
- puis une structure pour modéliser un set
 - struct Set
- Une partie est composée de 10 sets :

```
struct Game {  
    struct Set sets[10] ;  
}
```
- Il faut connaître le set en cours (ajout de currentSet dans Game)

bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :

- struct Game

- puis une structure pour modéliser un set

- struct Set

- Une partie est composée de 10 sets :

```
struct Game {  
    struct Set sets[10] ;  
}
```

- Il faut connaître le set en cours (ajout de currentSet dans Game)

bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

```
#include<assert.h>
#include<stdbool.h>

int main(){
    assert(false && « c'est parti ») ;
}
```

ok, le système de test fonctionne !

```
$ make bow
gcc -Wall bow.c -o bow
$ ./bow
bow: bow.c:5: main: Assertion `0 && "c'est parti"' failed.
```



bowling : cas vide

- Commençons par le cas vide

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game) ;
}

int main(){
    test_empty();
}
```

```
$ gcc -Wall bow.c
bow.c: In function 'test_empty':
bow.c:6:10: warning: implicit declaration of function 'bg_init' [-Wimplicit-function-declaration]
    struct BowlingGame *game=bg_init();
    ^
bow.c:6:28: warning: initialization makes pointer from integer without a cast
    struct BowlingGame *game=bg_init();
```



Ajout de bowling.h

```
#ifndef BOWLING_H
#define BOWLING_H

struct BowlingGame;

struct BowlingGame *bg_init();
void bg_roll(struct BowlingGame *,int );
int bg_score(struct BowlingGame *);
void bg_free(struct BowlingGame *) ;

#endif
```

bowling : cas vide

```
$ gcc -Wall bow.c  
/tmp/ccTstSlJ.o: In function `test_empty':  
bow.c:(.text+0xe): undefined reference to `bg_init'  
bow.c:(.text+0x2c): undefined reference to `bg_roll'  
bow.c:(.text+0x42): undefined reference to `bg_score'  
bow.c:(.text+0x6b): undefined reference to `bg_free'  
collect2: error: ld returned 1 exit status
```



Ajout de bowling.c

bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```

↓
Ajout de bowling.c

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{};

struct BowlingGame *bg_init(){
    return NULL;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
    return -1;
}
void bg_free(struct BowlingGame *g){}
```

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

bowling : cas vide

```
$ gcc -Wall bow.c bowling.c  
allali@hebus:/tmp/bowling$ ./a.out  
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

Ajout du score

```
$ gcc bowling.c bow.c  
-Wall  
$ ./a.out  
$ valgrind ./a.out  
ok
```



```
#include "bowling.h"  
#include <stdlib.h>  
  
struct BowlingGame{  
    int score ;  
};  
  
struct BowlingGame *bg_init(){  
    struct BowlingGame * g=malloc(sizeof(* g)) ;  
    g->score=0 ;  
    return g ;  
}  
void bg_roll(struct BowlingGame *g,int s){  
}  
int bg_score(struct BowlingGame *g){  
    return g->score;  
}  
void bg_free(struct BowlingGame *g){  
    free(g) ;  
}
```

bowling : test tout à 1

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){ ... }

void test_all_ones(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game) ;
}

int main(){
    test_empty();
    test_all_ones() ;
}
```

```
$ ./a.out
```

```
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```


bowling : test tout à 1

```
$ ./a.out
```

```
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

```
$ gcc bowling.c bow.c -Wall  
$ ./a.out  
$
```



```
#include "bowling.h"  
#include <stdlib.h>  
  
struct BowlingGame{  
    int score ;  
};  
  
struct BowlingGame *bg_init(){  
    struct BowlingGame * g=malloc(sizeof(* g)) ;  
    g->score=0 ;  
    return g ;  
}  
void bg_roll(struct BowlingGame *g,int s){  
    g->score+=s ;  
}  
int bg_score(struct BowlingGame *g){  
    return g->score;  
}  
void bg_free(struct BowlingGame *g){  
    free(g) ;  
}
```

bowling : code smell...

```
void test_empty(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game);
}

void test_all_ones(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game);
}

int main(){
    test_empty();
    test_all_ones();
    return 0;
}
```

code dupliqué



refactoring !

bowling : code smell...

```
void test_empty(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game);
}

void test_all_ones(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game);
}

int main(){
    test_empty();
    test_all_ones();
    return 0;
}
```



```
void rolls(struct BowlingGame *game,int n, int v){
    int i;
    for(i=0;i<n;++i)
        bg_roll(game,v);
}

void test_empty(){
    struct BowlingGame *game=bg_init();
    rolls(game,20,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game);
}

void test_all_ones(){
    struct BowlingGame *game=bg_init();
    rolls(game,20,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game);
}

int main(){
    test_empty();
    test_all_ones();
    return 0;
}
```

bowling : un spare

```
void test_one_spare(){
    struct BowlingGame *game=bg_init();
    bg_roll(game,5);
    bg_roll(game,5);
    bg_roll(game,3);
    rolls(game,17,0);
    assert(bg_score(game)==16 && "test one spare");
}
```



```
$ gcc bow.c bowling.c -Wall
allali@hebus:~/SVN_LaBRI/ENSEIRB/PG106/Cours$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16 && "test one spare"' failed.
```

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

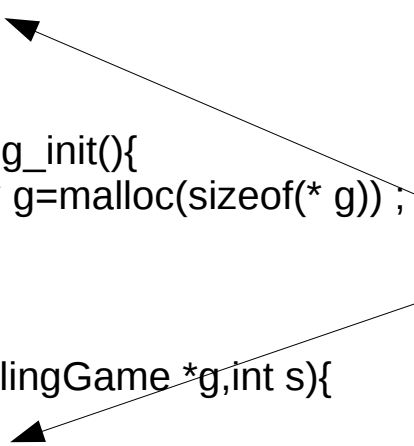
struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```



- Pour gérer un spare, il faut connaitre le coup d'avant.
- On pourrait ajouter un temporaire pour cela

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

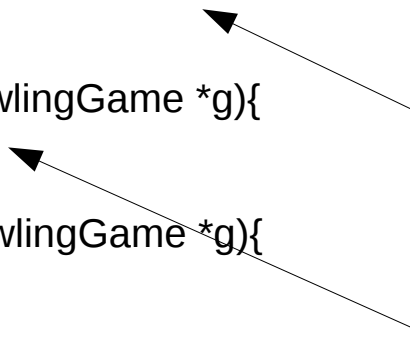
struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```



- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
 - roll : calcule le score mais ne devrait pas
 - score : doit calculer le score mais ne le calcul pas

bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
 - roll : calcule le score mais ne devrait pas
 - score : doit calculer le score mais ne le calcul pas

➔ **Refactoring !**

bowling : refactoring

- On revient en arrière :

```
int main(){  
    test_empty();  
    test_all_ones();  
    // test_one_spare() ;  
    return 0;  
}
```

```
$ gcc bow.c bowling.c -Wall  
$ ./a.out  
$
```



- On modifie le code : ajout d'un tableau de score, du coup en cours et mise à jour de la fonction de calcul

bowling : refactoring

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int rolls[21] ;
    int current;
    int score;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->current=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s;
    g->rolls[g->current++]=s ;
}

int bg_score(struct BowlingGame *g){
    int score=0,i ;
    for(i=0;i<g->current;++i) score+=g->rolls[i] ;
    return score ;
return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

\$ gcc bow.c bowling.c -Wall
\$./a.out
\$



```
int main(){
    test_empty();
    test_all_ones();
    test_one_spare() ;
    return 0;
}
```

\$ gcc bow.c bowling.c -Wall
\$./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.

bowling : one spare (back)

```
int bg_score(struct BowlingGame *g){  
    int score=0,i ;  
    for(i=0;i<g->current;++i) score+=g->rolls[i] ;  
    return score ;  
}
```



```
int bg_score(struct BowlingGame *g){  
    int score=0,i ;  
    for(i=0;i<g->current;++i) {  
        if (g->rolls[i]+g->rolls[i+1]==10){  
            // this is a spare...  
            score= ... ; // ?  
        }  
        score+=g->rolls[i] ;  
    }  
    return score ;  
}
```

ca ne marchera pas car il faut compter par set. On doit encore faire un refactoring !

```
int main(){  
    test_empty();  
    test_all_ones();  
    //test_one_spare() ;  
    return 0;  
}
```

```
$ gcc bow.c bowling.c -Wall  
$ ./a.out  
$
```



bowling : refactoring (again)

```
int bg_score(struct BowlingGame *g){  
    int score=0,i ;  
    for(i=0;i<g->current;++i) score+=g->rolls[i] ;  
    return score ;  
}
```



```
struct BowlingGame *bg_init(){  
    int i ;  
    struct BowlingGame * g=malloc(sizeof(* g)) ;  
    g->current=0 ;  
    for(i=0;i<21;++i) g->rolls[i]=0 ;  
    return g ;  
}
```

```
int bg_score(struct BowlingGame *g){  
    int score=0, frame;  
    for(frame=0;frame<10;++frame) {  
        score+=g->rolls[2*frame]+g->rolls[2*frame+1] ;  
    }  
    return score ;  
}
```

```
$ gcc bow.c bowling.c -Wall  
$ ./a.out  
$
```



bowling : one spare (again)

```
$ gcc bow.c bowling.c -Wall  
$ ./a.out  
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16  
&& "test one spare" failed.
```

```
int bg_score(struct BowlingGame *g){  
    int score=0, frame, hits;  
    for(frame=0;frame<10;++frame) {  
        hits=g->rolls[2*frame]+g->rolls[2*frame+1] ;  
        score+=hits ;  
        if (hits==10) // spare  
            score+=+g->rolls[2*frame+2] ;  
    }  
    return score ;  
}
```

```
$ gcc bow.c bowling.c -Wall  
$ ./a.out  
$
```



bowling : TDD

- Et ainsi de suite :
 - ajout d'un test avec un strike
 - cas pour la fin de partie
 - ...
- Le cycle à suivre en TDD est :
 - écriture d'un test
 - le test ne passe pas : ROUGE
 - écriture du code
 - le test passe : VERT
- Lorsqu'on ré-écrit des tests, on ne touche pas au code jusqu'à ce que ça repasse au vert

Tests

- Il existe plusieurs types de tests.
- Les plus importants sont :
 - Les tests unitaires
 - Les tests fonctionnels
 - Les tests d'intégration
 - Les tests de recette

Les tests unitaires

- Les tests unitaires ont pour objet de valider le fonctionnement d'une fonction.
- Pour qu'un test unitaire soit correct, il faut tester le fonctionnement « normal » ainsi qu'aux limites (cas NULL, domaine de valeur).
- Un test unitaire doit tester une fonction le plus indépendamment possible du reste du code : comment faire si la fonction utilise d'autres fonctions ?