R2.03 : Qualité de développement

Configuration et construction de projet, Git, Doxygen et tests unitaires

BUT Informatique 2e semestre

Eric REMY
eric.remy@univ-amu.fr

IUT d'Aix-Marseille, site d'Arles

Version du 20 octobre 2025

Introduction

Ce cours s'intéresse aux logiciels ou moyens techniques utiles aux développeurs lors de leur travail sur des projets de taille industrielle. Il ne se concentre pas sur ce qui fera l'objet d'autres enseignements :

- la gestion de projets informatiques (organisation humaine) ;
- la conception (UML...) ;
- la programmation elle-même (C++, SQL, JAVA, etc.).

Ce support de cours est divisé en plusieurs parties :

- l'automatisation de la construction ;
- la documentation du code ;
- le contrôle qualité.

Erwan Lenhard vous parlera de développement collaboratif (Git) (2h CM et 6h TP notées).

Première partie

Automatisation de la construction

Automatisation de la construction

Le temps passé à faire de la programmation ne se limite pas à écrire du code. Un développeur aura intérêt à se faciliter la vie avec :

- l'automatisation de la construction (*build*) ;
- la compilation croisée ;
- la configuration du code source :
  - en fonction de la plateforme de construction ;
  - en fonction de la plateforme d'exécution ;
- des outils qui font la configuration et la construction en même temps.

- Origine UNIX ; créé en 1977 ; GNU Make est un clone pour le projet GNU
- La commande make utilise un ensemble de règles constituées de :
  - un but (*target*) qui indique le fichier à produire (avant « : »)
  - les fichiers dont le but dépend (dependencies) pour sa construction (après le « : ») ;
  - une action à faire pour construire le fichier (ligne(s) du dessous, chacune précédée du caractère « tabulation »).
- Les actions seront effectuées automatiquement en cascade pour ne refaire que ce qui est nécessaire.
- make utilise la date de dernière modification de chaque fichier pour savoir qui est plus récent que qui.
- La première règle est celle qui est faite si on n'en précise aucune sur la ligne de commande.

```
1 prog.exe : main.o classe.o
2   g++ -o prog.exe main.o classe.o -lgdi32
3
4 main.o : main.cpp classe.hpp
5   g++ -c main.cpp
6
7 classe.o : classe.cpp classe.hpp
8   g++ -c classe.cpp
```

- On peut créer des variables pour généraliser les écritures
  - changer facilement de compilateur
  - changer facilement les options de compilation
- Variables prédéfinies : $@, $<, $^, etc.
- Des règles par défaut, etc.
- ⇒ lisez la documentation !

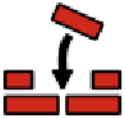
```
1  # Activez l'une ou l'autre des deux lignes ci-dessous
2  CXX=g++
3  #CXX=clang++
4
5  CXXFLAGS=-g -Wall -O0 # En cours de développement :
6  #CXXFLAGS=-O3 # Pour la livraison :
7
8  # Les biliothèques :
9  LIBS=-lgdi32          # Pour Windows
10 #LIBS=-lX11 -lXext # Pour Linux
11
12 prog.exe : main.o classe.o
13   $(CXX) -o $@ $^ $(LIBS)
14
15 main.o : main.cpp classe.hpp
16   $(CXX) $(CXXFLAGS) -c $<
17
18 classe.o : classe.cpp classe.hpp
19   $(CXX) $(CXXFLAGS) -c $<
```

Ant

https://ant.apache.org/

- Comparable à make
- Fait partie des outils classiques autour de JAVA
- Apparu vers 2000 pour automatiser la construction de Tomcat
- Utilise un fichier de configuration en XML
- Pour des projets plus complexes et pour l'intégration continue, l'outil Maven sera probablement à privilégier

SCons



https://scons.org/

- Comparable à make (mais pas seulement...)
- Moderne (apparu vers 1999)
- Multi-plateforme
- Fichiers de configuration en Python
- Utilisé par quelques projets connus (Quake3, MongoDB,...)

Ninja

`https://ninja-build.org/`

- Comparable à make
- Moderne (2010)
- Multi-plateforme
- Utilisé par des gros projets connus : Google Chrome, certaines parties d'Android, LLVM, etc.
- Ninja est un outil de bas niveau
- Destiné à être utilisé avec un générateur de configuration
- On va y revenir...

imake

- Lié au système X-Window qui sert à avoir des fenêtres sous UNIX
- imake a été créé (1987) pour permettre de compiler X11 sur plusieurs UNIX différents
- Crée un fichier Makefile différent pour chaque plateforme cible
- Ce programme est obsolète mais pas l'idée qui est derrière !...

Configurer le code source

- On peut écrire un code source pour plusieurs plateformes (#if...#endif)
- Tester les capacités du compilateur (version C++14, C++17, C++20, #pragma, etc.)
- Tester la présence et la version de bibliothèques (Qt5, Qt6, Boost, etc.)
- Adapter à un système d'exploitation particulier (Windows, MacOSX, Linux, autre)
- Laisser l'utilisateur choisir ce qu'il fabrique ou non (quand tout n'est pas utile)
- Laisser l'utilisateur choisir où et comment il installe le programme (globalement pour tous les utilisateurs ou seulement pour un utilisateur ; dans /usr/bin ou /usr/local/bin ou /opt/bin, etc.)

https://www.qt.io/

- Fait partie du système de Qt
- Ne se cantonne pas aux applications graphiques
- Assez facile à utiliser en particulier car la documentation est claire !
- C'est probablement mon système préféré actuellement pour les étudiants

cmake

`https://cmake.org/`

- Créé vers 2000 ; multi-plateforme (Windows, Linux, MacOSX, etc.)
- Grande popularité ; très général
- Utilisé par beaucoup de logiciels libres ou non : Insight Toolkit pour le Visible Human Project, VTK, MySQL (sous Windows), KDE, LLVM/Clang, Netflix, OpenCV, SFML, Swift (Apple), Android Native Development Kit (Google), Blender, etc.
- Utilise des fichiers CMakeLists.txt et xxxx.cmake
- Peut générer :
  - un fichier Makefile pour Make ;
  - un fichier ninja.build pour Ninja ;
  - une configuration pour Meson ;
  - un fichier projet pour Microsoft VisualStudio ;
  - un fichier projet pour Apple XCode ;
  - ...

- Projet GNU
- Formé de plusieurs sous-systèmes complémentaires : autoconf, automake et libtool...
- Système ancien... voire archaïque (1991, 1994 et 1997 respectivement) !
- Très puissant et fin
- Assez complexe à utiliser ; très complexe à maîtriser !
- Il va falloir du temps pour le remplacer !
- autoconf génère un script bash nommé configure que l'utilisateur lance pour faire des tests et générer un Makefile avant de pouvoir compiler
- automake est apparu ensuite pour générer le Makefile.in
- Pour générer des bibliothèques dynamiques, utiliser en plus libtool

https://mesonbuild.com/

- Logiciel écrit en Python (2013)
- Multi-plateforme (UNIX, Windows, MacOSX,...)
- Très activement développé
- De nombreux projets s'y convertissent : systemd, X.org, Glib, Gtk+, (GNOME), GStreamer, Mesa, etc.

Environnement de développement intégrés (*IDE*)

- Il n'est pas questions ici des éditeurs avancés (Atom, Sublime, Notepad++, etc.) ou des « petits » IDE (VisualStudioCode, QtCreator) qui sont finalement assez simples à utiliser à votre niveau.
- Il est question des « gros » IDE qui peuvent traiter des projets très complexes.
- VisualStudio, XCode, Eclipse, AndroidStudio, (Netbeans)
- C'est super... quand vous savez bien l'utiliser !
- ... sinon c'est un calvaire !
- Pour survivre, il est donc impératif de lire la documentation (« RTFM ! »).

Deuxième partie

Documentation du code

Documentation d'un logiciel

Un projet logiciel industriel est inévitablement accompagné de plusieurs types de documentations :

- une expression du besoin (ce qu'à l'IUT vous appellerez « cahier des charges ») ;
- une description de l'architecture globale du projet (à l'IUT, le « cahier de conception ») ;
- une documentation technique qui décrit le code lui-même, les algorithmes, les interfaces de programmation (API), etc. ;
- une documentation destinée à l'utilisateur final, à l'administrateur d'un système, à une hotline.

Nous allons nous intéresser maintenant à la documentation technique...

Documentation du code

Dans un projet logiciel d'une ampleur professionnelle, il est impératif d'assurer la documentation du code produit :

- pour améliorer le travail de groupe (« ça fait quoi cette fonction ? ») ;
- pour assurer la maintenance du programme (« si on change ça, quel impact cela aura ? ») ;
- etc.

Il est également apparu à l'usage que la meilleure façon d'avoir une documentation qui ne se sépare pas (retard, incohérence, voire absence) du code qu'elle accompagne consiste à réclamer aux développeurs eux-mêmes d'assurer sa rédaction, et ce directement dans le code au sein des commentaires.

Quelques outils

Différents outils permettant de documenter du code ont vu le jour :

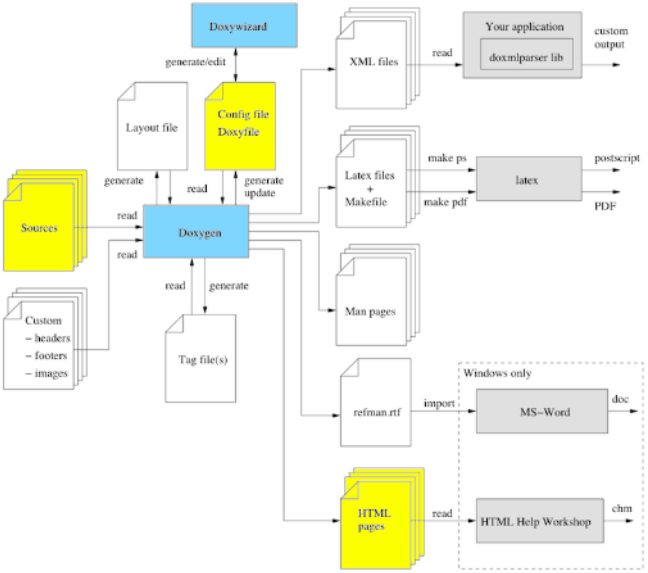
- pour JAVA, javadoc (système créé par Sun Microsystems pour la documentation du langage lui-même) ;
- pour Python, Sphinx ;
- pour C/C++ (et pleins d'autres), Doxygen ;
- etc.

La syntaxe des commentaires du langage est améliorée de balises spéciales reconnues par l'outil de documentation et permettant au programmeur d'indiquer des informations sur les classes, fonctions, paramètres, etc.
Ces informations sont ensuite compilées pour produire une documentation dans un ensemble de formats possibles : HTML, PDF, DocBook, UNIX man page, aide Windows (`.chm`), RTF, XML, etc.
Nous allons utiliser Doxygen en TP, et donc voir maintenant son usage...

Doxygen

- Doxygen est un logiciel libre
- Accepte : C, C++, Objective-C, C#, PHP, Java, Python, etc.
- Site Web de Doxygen
- Documentation sur le site Web : onglet « Documentation »
- Avant le TP, vous devrez lire les rubriques « Getting started » et « Documenting the code »

- La commande doxygen lit le fichier de configuration Doxyfile et en fonction de ce qui s'y trouve, parcourt les fichiers sources et génère la documentation.
- La commande doxywizard permet de générer le fichier de configuration Doxyfile assez facilement... mais il est parfois nécessaire de peaufiner à la main derrière.



- Souvent deux syntaxes possibles :
  - JavaDoc : « /** Blah blah. */ » ou bien « /// Blah blah. »
  - Qt : « /*! Blah blah. */ » ou bien « //! Blah blah. »
- Le commentaire est à mettre immédiatement avant ce qu'il décrit.
- Si vous activez « JAVADOC_AUTOBRIEF » (respectivement « QT_AUTOBRIEF ») dans la configuration, la première phrase (jusqu'au premier « . » rencontré) servira de descripion courte (*brief*). Sinon, il faut l'indiquer explicitement, n'importe où, avec « \brief Blah blah. ».
- On peut décrire un paramètre ou une donnée membre immédiatement après en indiquant :
  - JavaDoc : « /**< Blah blah. */ » ou bien « ///< Blah blah. »
  - Qt : « /*!< Blah blah. */ » ou bien « //!< Blah blah. »
- Pour une fonction, on peut aussi commenter les paramètres depuis l'intérieur de la description :
  - JavaDoc : « @param a est un entier qui sert à ça. »
  - Qt : « \param a est un entier qui sert à ça. »

- Pour le type de retour depuis l'intérieur de la description :
  - JavaDoc : « `@return la valeur X+1.` »
  - Qt : « `\return la valeur X+1.` »
- Plein d'autres mots-clés : `\author`, `\warning`, `\image`, etc.
- Possible de commenter aussi bien dans le « `.hpp` » que dans le « `.cpp` ».
- Pour la mise en forme, le langage Markdown est utilisable (sections, listes, gras, souligné, liens, images, etc.).
- Grâce à LaTeX, il est également possible de mettre des formules mathématiques.
- Bien d'autres possibilités... mais qui ne seront pas nécessaires pour le TP prévu : à creuser ultérieurement pendant les projets de SAÉ, le stage, ...

Seule la syntaxe Doxygen (JavaDoc ou Qt) change dans l'exemple ci-dessous.

Syntaxe JavaDoc

```
/** Description brève de A. Description longue
 qui s'étend sur plusieurs lignes. Et aussi
 plusieurs phrases.
 @warning Attention à ce détail.
 @author E. Remy
*/
class A {
 int a; ///< L'entier (description courte).
 double b; ///< Le réel (descr. courte). Description longue.
public:
 /** Description courte. Description plus longue.
 @param x la valeur à traiter.
 @return Le résultat est un entier valant x+10.
 */
 int fonction(int x);
};
```

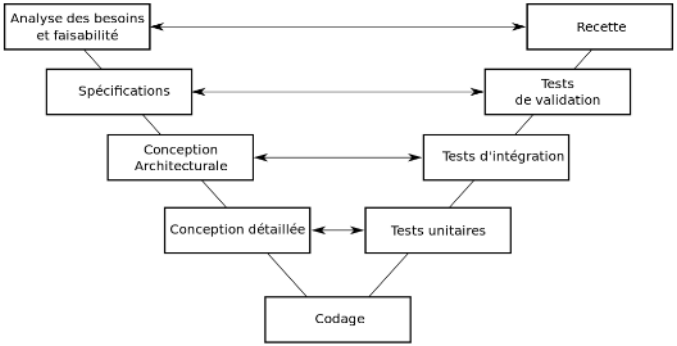
Syntaxe Qt

```
/*! Description brève de A. Description longue
 qui s'étend sur plusieurs lignes. Et aussi
 plusieurs phrases.
 \warning Attention à ce détail.
 \author E. Remy
*/
class A {
 int a; //!< L'entier (description courte).
 double b; //!< Le réel (descr. courte). Description longue.
public:
 /*! Description courte. Description plus longue.
 \param x la valeur à traiter.
 \return Le résultat est un entier valant x+10.
 */
 int fonction(int x);
};
```

Troisième partie

Contrôle de la qualité des programmes

Vie d'un projet : le cycle en V



- Juste après le développement de chaque fonctionnalité, on implante les tests permettant de vérifier que chaque fonctionnalité détaillée est bien respectée.

  $\Rightarrow$ tests unitaires

- Ensuite, le programme complet sera testé pour vérifier que l'intégration de chacune des différentes unités ne fait pas apparaître un problème.

  $\Rightarrow$ tests d'intégration

Exemples de tests unitaires

Quelques exemples d'attentes légitimes :

- Sur la classe Polynome, on peut vouloir vérifier que calculer la valeur d'un polynôme pour $x = 0$ donne bien la valeur de la constante c_0 correspondant au monôme de degré 0.
- Toujours sur Polynome, on peut vérifier que la valeur pour $x = 1$ est égale à la somme de tous les coefficients c_i de chaque monôme.
- Sur toute classe conteneur (vector, string, Polynome, etc.), les operator [] en lecture et écriture doivent donner accès au bon élément.
- Sur la classe Ratio, on peut vouloir vérifier que créer un rationnel $1/0$ déclenche bien une (exception) à l'exécution (et du type décidé à la conception).
- Sur chaque classe, on peut tester qu'une construction de copie donne bien une instance dont chaque membre est identique à ceux de sa source.
- De manière générale, on cherchera à tester toutes les valeurs limites, les valeurs problématiques, etc.
- Avez-vous d'autres idées ?

Intérêt des tests

L'ensemble des tests disponible pour un code présente un certain nombre d'avantages :

- Évidemment, c'est une mesure de qualité du code existant.
- Cela fournit aussi une mesure de l'avancement du développement, lorsque les composants non développés font échouer les tests.
- Les tests illustrent la façon correcte (ou non) d'utiliser un composant... et ainsi participent à l'effort de documentation de ce composant.

Régression et couverture du code source

- Faillir à un test suite à une modification du code, alors qu'il était validé auparavant signifie qu'une régression a été introduite involontairement, et doit être analysée et corrigée.
- Corriger le code... ou le test.
- « Tester tout ce qui peut casser »... mais, dans le cas général, impossible de tout tester.
- Couvrir une partie importante (si ce n'est tout) du code source choisie à l'avance :
  - Function coverage : appeler chaque fonction ;
  - Statement coverage : tester chaque instruction ;
  - Condition coverage : tester chaque cas de `if` () ;
  - Path coverage : généralisation du précédent à tous les chemins possibles dans un code.
- Outil `gcov` pour `g++` (fait aussi du profilage, c'est-à-dire tracer les temps d'exécution)

- Pour aller plus loin, depuis quelques dizaines d'années, il devient possible des démontrer formellement (comme un théorème) les algorithmes (ou un circuit électronique) et leurs propriétés... mais ça reste encore essentiellement un domaine de Recherche scientifique.

Alternatives au cycle en V (1/2)

Le cycle en V qui est le plus répandu comporte quelques inconvénients qui le rendent impropre à certains projets :

- Le développement de toutes les briques doit en général être fini avant d'entamer les tests. Il est donc possible qu'un problème majeur de conception trouvé au moment des tests, oblige à tout recommencer !
- Le client ne voit le programme que lors des dernières phases de test qui le concernent. Il ne peut donc pas indiquer si les tests prévus par l'équipe de projet sont judicieux/corrects.
- Le client ne peut pas non plus indiquer si le produit lui convient ou non avant la livraison !
- Difficile d'appliquer un cycle en V à un domaine mouvant (par exemple, le Web) ou incertain (la recherche scientifique) puisque les attentes ne peuvent pas toujours être fixées au début du projet autrement que sous forme de souhaits !

Alternatives au cycle en V (2/2)

D'autres formes de gestion de projets sont donc apparues :

- *extreme programming* (XP) : faire beaucoup de cycles courts, appelés « run », enchaînant développement, tests, consultation du client, révision de la conception, etc. Préconisations sur la façon de développer ayant des conséquences sur le projet (vision bottom-up).
- *SCRUM* arrive est très comparable à XP mais en partant de la gestion du projet (vision top-down).
- *test driven development* (TDD) : réaliser les tests d'abord (en suivant un cahier de conception très détaillé) puis seulement après les classes à tester, afin d'éviter que les tests soient guidés par des idées préconçues.
- XP+TDD sont souvent utilisés pour le développement en binôme : l'un code les tests, l'autre le programme source à tester.

« Méthodes agiles »

Confronter une réalisation à sa spécification... tout au long du développement !

Mock object

- Tester un composant (une classe par exemple) dépend souvent de son environnement d'exécution :
  - un service Web dont la stabilité ou le temps de réponse sont un problème ;
  - une base de données qui peut être lente ou parfois indisponible ;
  - un système pas encore disponible au moment du développement ;
  - tout système au fonctionnement non garanti et extérieur au composant testé.
- Le test unitaire doit être protégé des faiblesses de ces composants extérieurs !
- Le coût de calcul pour l'accès à ces services est parfois trop important lorsqu'on les tests génèrent des milliers d'accès. Les tests doivent être rapides et fiables.
- Il faut alors créer un composant qui simule, pour le test uniquement, le(s) composant(s) extérieur(s) dont on doit se passer : un simulacre (*mock object*).
- Un *mock object* permet aussi de simuler des erreurs difficiles à produire autrement (manque de mémoire, erreur matérielle, etc.).
- Il s'agit en général d'une classe simpliste dont les méthodes renvoient une valeur décrite par la spécification.

Séquence générale d'un test unitaire

- Un test unitaire adopte toujours le schéma général suivant :
  1. mise en place de l'environnement de test complet pour ce composant (*SetUp*) ;
  2. test(s) proprement dit(s) ;
  3. vérification des résultats (*success*, *fail*, *skipped*) ;
  4. netoyage de l'environement de test (*TearDown*).
- Chaque test unitaire doit être indépendant des autres :
  - Une erreur de l'un ne doit pas contaminer les autres ;
  - On fait souvent passer une sous-partie des tests... ce qui n'est possible que s'ils sont indépendants.
- Chaque infrastructure de test (*framework*) vous imposera sa façon de faire.

Tests d'intégration

- Couche de tests située « au dessus » des tests unitaires.
- Le but est de vérifier que les composants fonctionnent ensemble.
- Tester la fiabilité et la performance du logiciel *complet*.

Implanter des tests

Quelques mécanismes de tests que vous pourriez utiliser en C++ (parmi un très large choix) :

- CppUnit : la variante C++ de la (célèbre) bibliothèque de tests JUnit créée pour Java ;
- Boost::Test : la bibliothèque de tests faisant partie de l'ensemble de bibliothèques C++ boost ;
- Qt Test : la bibliothèque de tests du projet Qt ; elle permet également de tester les applications graphiques ;
- GoogleTest : pour C++ elle aussi, également facile à utiliser ;
- GNU DejaGnu : la plate-forme de tests du projet GNU.
- ...

J'ai choisi de vous faire manipuler boost::test...

Généralités sur Boost

- Boost est un ensemble de bibliothèques et d'outils portables, utiles en C++.
- D'un *très bon* niveau de programmation C++.
- Les fonctionnalités de Boost ont vocation à être graduellement introduites dans la norme elle-même (par exemple les « smart pointers »).
- http://www.boost.org/
- Très vaste nombre de compléments à la norme, avec (non-exhaustif) : Array, Container, Exception, Filesystem, Foreach, Geometry, Graph, Lambda, Locale, Math, Python, Ratio, Regex, Serialization, Thread, Timer, uBLAS, etc.
- boost::test n'est qu'une de ces nombreuses fonctionnalités !
- Si vous codez une application portable ne dépendant pas déjà d'une grosse bibliothèque (Qt, Microsoft MFC, etc.), vous devriez aller voir d'abord dans boost si vous ne trouvez pas votre bonheur.

Module de test

- Vous créez un module de test (ici « MesTests »).
- Vous regroupez par thèmes vos tests individuels dans une ou plusieurs catégories de tests (nommée ici « GroupeUn »).

Structure générale d'un programme de test : test0.cpp

```
1 #define BOOST_TEST_MODULE MesTests
2 #include <boost/test/unit_test.hpp>
3
4 // Ici vos includes, déclarations, etc.
5
6 BOOST_AUTO_TEST_CASE( GroupeUn )
7 {
8     // Mettez ici vos tests
9 }
```

- L'exécution affichera le détail ainsi qu'un bilan des tests réussis et échoués.
- Attention : c'est Boost::Test qui s'occupe d'ajouter le main() à vos tests !

Quelques tests possibles (1/2)

- BOOST_CHECK(exp); : Si exp est faux, une erreur comportant le texte de exp est affichée. Les tests se poursuivent mais le TEST_CASE est *failed*.
- BOOST_WARN(exp); : Comme ci-dessus mais le TEST_CASE n'est pas raté.
- BOOST_REQUIRE(exp); : Si exp est faux, le message est affiché puis une exception est lancée. Dans le cas général, cette exception arrêtera la série de tests. Ce test est donc prévu pour les cas où les tests ultérieurs ne pourraient pas être effectués si celui-là rate.

- if (exp) BOOST_ERROR("Ouch..."); : similaire à BOOST_CHECK(exp); mais en ayant le contrôle direct sur le if () et le message d'erreur.
- if (exp) BOOST_FAIL("Ouch..."); : *idem* mais pour BOOST_REQUIRE(exp);.

- BOOST_CHECK_MESSAGE(exp,msg); : Si exp est faux, l'expression msg (qui peut être complexe et contenir l'opérateur <<) est affichée à la place du message standard de BOOST_CHECK(exp);.

Quelques tests possibles (2/2)

- BOOST_CHECK_EQUAL(exp1,exp2); : rate si les deux expressions exp1 et exp2 ne sont pas égales. Affiche explicitement les valeurs de exp1 et exp2.
- BOOST_CHECK_EQUAL_COLLECTIONS(start1,end1,start2,end2) : comme ci-dessus mais pour comparer deux séries de valeurs pointées par les itérateurs start (inclu) et end (exclu).

- BOOST_CHECK_NO_THROW(exp) ; vérifie que l'évaluation de exp ne lance aucune exception.
- BOOST_CHECK_THROW(exp,exception) ; vérifie que l'évaluation de exp lance une exception du type exception.

- BOOST_CHECK_CLOSE(val1,val2,tolerance) : pour des valeurs réelles, vérifie si val1 == val2 à tolerance pourcent près. Les expressions val1 et val2 doivent être du même type (float ou double).

Exemple (1/2)

Quelques exemples de tests : test1.cpp

```
1  #define BOOST_TEST_MODULE STD_STRING_TESTS
2  #include <boost/test/unit_test.hpp>
3
4  #include <string>
5  using namespace std;
6
7  BOOST_AUTO_TEST_CASE( Construction )
8  {
9   string s1, s2("C++");
10  BOOST_CHECK_EQUAL(s1.length(),0); // Une chaîne vide doit avoir une longueur nulle.
11  BOOST_CHECK_EQUAL(s2.length(),3); // La chaîne « C++ » doit avoir une longueur de 3.
12 }
13
14 BOOST_AUTO_TEST_CASE( Concatenation )
15 {
16  string s1("Hello"),s2(" world!");
17  s1+=s2; // Concaténation
18  BOOST_CHECK_EQUAL(s1.length(),12); // La longueur de la chaîne concaténée doit être de 12...
19  BOOST_CHECK(strcmp(s1.c_str(),"Hello world!")==0); // et elle doit valoir « Hello world! ».
20 }
```

Exemple (2/2)

- boost::test étant implantée sous Linux en bibliothèque à liaison dynamique, il y a deux choses à rajouter lors de la compilation :
  - il faut définir le symbole BOOST_TEST_DYN_LINK qui indique aux includes que la bibliothèque de test est dynamique ;
  - il faut lier votre programme à la bibliothèque « boost_unit_test_framework ».
- On a donc une ligne de compilation de la forme :
  g++ −D BOOST_TEST_DYN_LINK test1.cpp −lboost_unit_test_framework −o test1
- L'exécution, par « ./test1 », donne le message suivant (aucun test échoué) :

```
Running 2 test cases...
*** No errors detected
```

Intégration continue

- Système automatique basé sur un gestionnaire de version (git, subversion, etc.)
- Automatiser la compilation, éventuellement vers plusieurs architectures (i386, amd64, arm, Windows, MacOS, Linux, etc.).
- Automatiser les tests (unitaires et d'intégration, mis à égalité) pour qu'ils soient contrôlés très régulièrement pendant le développement (chaque nuit, à chaque commit, etc.).
- Jenkins et GitLab peuvent faire ces deux derniers points.
- Donner au chef de projet des mesures de qualité objectives pour suivre l'évolution du projet.
- SonarQube permet le calcul et la visualisation de métriques de qualités du code.
- Les *frameworks* de tests fournissent des outils pour manipuler les résultats (sortie texte, sortie XML, intégration à une forge logicielle sur le Web, lien avec des tickets de panne, etc.).

- On peut même viser la *livraison continue*...

Forge logicielle

- Système logiciel favorisant les conditions de travail de l'équipe de développement en englobant :
  - la gestion de version et l'intégration continue ;
  - des listes de diffusion ou des forums, éventuellement un tchat ;
  - un outil de suivi de bug ;
  - une gestion de documentation souvent sous forme de Wiki ;
  - la gestion des tâches (Gantt, etc.) ;
  - une communication externe (site Web, présentation du projet, capture d'écran, actualités, publication de la licence d'utilisation)
- Les plus connues en logiciels libres : GitLab, Redmine, ... ; en logiciels propriétaires : BitBucket, GitHub, Microsoft Team Foundation Server + Visual Studio Team System

- Attention : comme pour beaucoup de services « gratuits » actuels (GAFAM), lisez bien les conditions d'utilisation ! Ce serait une faute grave de rendre public un code source qui ne doit pas l'être ! Les travaux de SAÉ, votre travail de stage en particulier n'ont pas à être rendu publics car vous ne disposez pas de *l'intégralité* des droits à leur sujet...

Avez-vous des questions ?