

Chapitre 9 : Bibliothèques

Construction et maintenance de logiciels

Guy Francoeur

basé sur du matériel pédagogique d'Alexandre Blondin Massé, professeur

UQÀM | **Département d'informatique**

Table des matières

1. Dépendances d'un projet
2. Lier une bibliothèque
3. Concevoir une bibliothèque
4. La bibliothèque CUnit
5. La librairie **libasan.so** *Address Sanitizer*
6. Compilation statique des librairies

Table des matières

1. Dépendances d'un projet
2. Lier une bibliothèque
3. Concevoir une bibliothèque
4. La bibliothèque CUnit
5. La librairie **libasan.so** *Address Sanitizer*
6. Compilation statique des librairies

Dépendances

- ▶ Point important en **construction de logiciels**;
- ▶ Plusieurs types de **dépendances** :
 - ▶ **Logicielle** : notre programme dépend d'un autre **logiciel**, par exemple Graphviz;
 - ▶ **Bibliothèques** : notre programme dépend d'une **bibliothèque C**, par exemple Jansson;
 - ▶ **Système** : notre programme ne fonctionne que sur **Linux, Mac OS**, etc.
 - ▶ **Données** : notre programme utilise des **données** provenant d'un projet particulier.
- ▶ Il est nécessaire de les **identifier** et de les **documenter** le plus possible (par exemple, dans le fichier **README**).

- ▶ permet de lister les symboles dans un fichier exécutable (binaire);
- ▶ permet de voir la dépendance (provenance) du symbole;
- ▶ pour plus d'information consultez : `man nm` ou `nm --help`;

```
$ nm executable  
ou  
$ nm librairie.o
```

- ▶ Permet de savoir :
 - ▶ quels sont les dépendances;
 - ▶ quels sont les requis (bibliothèques, objets);
 - ▶ que votre exécutable ou bibliothèque binaire (.so) utilisera.
- ▶ Exemple simple :

```
$ ldd /bin/ls
```

Comment bien documenter

- ▶ Idéalement, identifier les **plateformes** sur lesquelles...
 - ▶ ...le programme a été **testé** et **développé**;
 - ▶ ...on s'**attend** à ce que le programme soit **fonctionnel**.
- ▶ Indiquer les **bibliothèques** non standards dont le projet dépend, avec un lien vers ces bibliothèques expliquant comment l'**installer** (installation **système** ou **locale** ?);
- ▶ Indiquer les **données** à télécharger (idéalement, automatiser le téléchargement si possible);
- ▶ Indiquer les **logiciels** à installer;
- ▶ Faire attention aux **licenses** !

- ▶ Pour le **code** :
 - ▶ **GPL** : source ouverte, libre, plus **restrictive**;
 - ▶ **BSD**, **MIT**, **Apache** : source ouverte, libre, plus **permissive**.
 - ▶ **propriétaire**.
- ▶ Pour les **documents**, **images**, **vidéos**, etc. :
 - ▶ **CreativeCommon** : prévoit plusieurs niveaux de permission.
- ▶ Idéalement, **tous vos projets** devraient inclure une license, en particulier ceux qui sont **publics**.

Critères à considérer :

- ▶ Est-ce que le code est **ouvert** ou non ?
- ▶ Est-ce que je peux **commercialiser** mon produit s'il utilise ce code ?
- ▶ Est-ce que j'ai des **redevances** (*royalties*) à payer si je commercialise mon projet ?
- ▶ Est-ce que je peux **transformer** une image et considérer qu'elle est à moi ?
- ▶ Est-ce que je peux **distribuer** le code ?
- ▶ Est-ce que je dois **reconnaître** qui a conçu le code ou le document ?

Table des matières

1. Dépendances d'un projet
2. Lier une bibliothèque
3. Concevoir une bibliothèque
4. La bibliothèque CUnit
5. La librairie **libasan.so** *Address Sanitizer*
6. Compilation statique des librairies

- ▶ Rappel sur les **étapes** de compilation :
 - ▶ On **compile** : **.c** → **.o**;
 - ▶ On **établit les liens** : **.o** → exécutable;
- ▶ Lors de la **compilation**, gcc doit trouver les **en-têtes**;
- ▶ Lors de l'**édition des liens**, gcc doit trouver les **binaires** correspondants.

Emplacement des fichiers d'en-tête

- ▶ À la **compilation**, **gcc** tente de trouver les fichiers d'en-têtes seulement (fichier **.h**);
- ▶ Pas de **chemin absolu**, sinon le code n'est pas portable.
- ▶ Si votre fichier **.h** se trouve ailleurs, il faut l'indiquer :

```
$ gcc -I<chemin>
```

Emplacement des bibliothèques (binaires)

- ▶ À l'**édition des liens**, gcc tente de trouver les **implémentations** correspondantes;
- ▶ Il inspecte **plusieurs répertoires**, qu'on peut connaître via la commande

```
$ gcc -v hello.c -Wl,--verbose
```

- ▶ Si votre **bibliothèque** se trouve ailleurs, il faut l'indiquer :

```
$ gcc -L<chemin> ...
```

- ▶ Ici aussi, pas de **chemin absolu**.

Table des matières

1. Dépendances d'un projet
2. Lier une bibliothèque
3. Concevoir une bibliothèque
4. La bibliothèque CUnit
5. La librairie **libasan.so** *Address Sanitizer*
6. Compilation statique des librairies

Deux types de bibliothèques

► Statique :

- Extension : **a** en Unix, **lib** sous Windows;
- La bibliothèque est **incluse** dans l'exécutable;
- **Avantage** : Réduit les **dépendances**;
- **Inconvénient** : Exécutables **plus volumineux**.

► Dynamique (*shared*) :

- Extension : **so** en Unix, **dll** sous Windows;
- La bibliothèque est **liée dynamiquement**;
- **Avantage** : Évite les **redondances**;
- **Inconvénient** : Nécessite une **installation**.

Exemple : Vec3D (1/2)

- ▶ Considérons un exemple simple d'une bibliothèque supportant la manipulation de **vecteurs**;
- ▶ Les fichiers utilisés sont **vec3d.h** et **vec3d.c**;
- ▶ Tout d'abord, on **compile** le fichier **.c** en binaire **.o** :

```
$ gcc -o vec3d.o -c vec3d.c
```

- ▶ Ensuite, on crée la **bibliothèque statique** :

```
$ ar -cvq libvec3d.a vec3d.o
```

- ▶ On peut ensuite l'inclure via l'instruction

```
1 #include "vec3d.h"
```

- ▶ en autant que l'**en-tête** et l'**implémentation** soient disponibles.

Exemple : Vec3D (2/2)

- ▶ Par exemple, supposons que les fichiers **vec3d.h** et **libvec3d.a** se trouvent respectivement dans les répertoires
/usr/gf/clib/include
/usr/gf/clib/lib

- ▶ Alors il suffit de **compiler** avec la commande

```
$ gcc -I/usr/gf/clib/include -c test_vec3d.c
```

- ▶ et de compléter l'**édition des liens** avec

```
$ gcc -L/usr/gf/clib/lib -lvec3d \  
-o test_vec3d test_vec3d.o
```

- ▶ **Note :** Pas de chemin **absolu** lorsque vous distribuez votre bibliothèque!

Construction d'une librairie Dynamique

- En deux étapes :

```
$ gcc -c -fPIC hello.c -o hello.o  
$ gcc hello.o -shared -o libhello.so
```

- En utilisant une ligne :

```
$ # -fPIC indique que le code sera compilé pour  
$ # être à une adresse variable (différent)  
$ # PIC = Position Independent Code  
$ gcc -shared -o libhello.so -fPIC hello.c
```

.so versus .a (résumé)

- ▶ A quoi bon le fichier **.a** ?
- ▶ A quoi bon le fichier **.so** ?
- ▶ Ils seront utilisés à quel moment ?

Table des matières

1. Dépendances d'un projet
2. Lier une bibliothèque
3. Concevoir une bibliothèque
4. La bibliothèque CUnit
5. La librairie **libasan.so** *Address Sanitizer*
6. Compilation statique des librairies

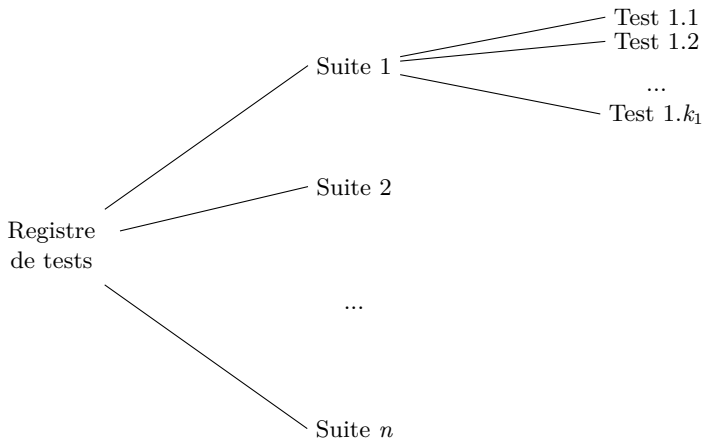
- ▶ **CUnit** est un cadre de **tests unitaires**;
- ▶ Inspiré de **JUnit**;
- ▶ Il est à **code source ouvert** et **libre**;
- ▶ Créé en **2002** par **A. Kumar** et **J. St.Clair**;
- ▶ Disponible à l'adresse <http://cunit.sourceforge.net/>;
- ▶ Facile à **installer** : **Aptitude/apt-get** (sur Debian/Ubuntu/Mint), **yum** (sur Centos/RedHat) **MacPorts/Brew** (sur MacOS);
- ▶ Ensuite, il suffit, avec gcc, de **lier** votre programme avec la **bibliothèque** fournie par **CUnit**.

CUnit est :

- ▶ Une librairie légère en C;
- ▶ Un framework complet;
- ▶ Propose plusieurs interfaces utilisateur.

Il est possible de faire un parallèle avec la bibliothèque native et standard **assert.h**.

Structure générale



Un petit exemple avec **assert.h**

```
// assert.c
#include <assert.h>

int main(void) {
    int a = 5;
    assert(a == 5 && "mon premier assert est invalide");
#ifdef ERR
    assert(a == 6 && "mon deuxieme assert est invalide");
#endif
    return 0;
}
```


Cunit - comment ça marche

Les étapes de réalisation sont :

- ▶ **installer** la librairie CUnit; (déjà dans Java)
- ▶ **construire** un programme en C **main()**;
- ▶ **coder** les fonctions de tests (**Agile TDD**);
- ▶ **compiler** le programme avec les tests;
- ▶ **exécuter** le programme de test;
- ▶ **vérifier** les résultats.

Exemple

Un exemple complet est disponible sur le site du cours

- ▶ dans le site du cours *INF3135__A2020*
- ▶ consulter le répertoire code/
- ▶ https://github.com/guyfrancoeur/INF3135__A2020/blob/master/code/cUnit.md

Table des matières

1. Dépendances d'un projet
2. Lier une bibliothèque
3. Concevoir une bibliothèque
4. La bibliothèque CUnit
5. La librairie **libasan.so** *Address Sanitizer*
6. Compilation statique des librairies

- ▶ Role : Un détecteur de *out-of-bounds* et *usage-after-free*;
- ▶ Utile pour trouver ce que nous ignorons;
- ▶ La librairie doit préalablement être installé dans le système:

```
# pour les systèmes debian et ubuntu  
sudo apt-get install libasan5 -dbg
```

- ▶ Cette librairie binaire **.so** sera include lors de la compilation;

```
$ gcc -fsanitize=address source.c
```

- ▶ Important de bien installer la librairie en fonction du compilateur;
- ▶ **libasan v5** avec **GCC v8**;

Table des matières

1. Dépendances d'un projet
2. Lier une bibliothèque
3. Concevoir une bibliothèque
4. La bibliothèque CUnit
5. La librairie **libasan.so** *Address Sanitizer*
6. Compilation statique des librairies

Compilation statique

- ▶ Il est possible de compiler et lier les bibliothèques dans notre exécutable

```
$ gcc -static -fpic -o mem_static memoire.c
```

- ▶ de cette façon l'exécutable ne requiert aucune bibliothèque dynamique, il ne reste qu'à le distribuer sans maux de tête;
- ▶ La commande suivante indique si des **.so** sont nécessaires

```
$ ldd mem_static
```

- ▶ Il ne devrait pas avoir de dépendance.