GLCS : Réutilisation

Outils pour la réutilisation logicielle

Réutilisabilité

In computer science and software engineering, reusability is the use of existing assets in some form within the software product development process; these assets are products and by-products of the software development life cycle and include code, software components, test suites, designs and documentation. The opposite concept of reusability is leverage, which modifies existing assets as needed to meet specific system requirements.

https://en.wikipedia.org/wiki/Reusability

Bibliothèque et Cadriciel

- Une bibliothèque (library) est une brique logicielle réutilisable qui offre un ensemble de fonctionnalités
  - Suivant le langage, la bibliothèque offre un ensemble de fonctions, de classes utilisables depuis le code
  - Un code peut s'appuyer sur une ou plusieurs bibliothèques
  - Le code structure le déroulement des appels de la ou des bibliothèques

- Un « cadriciel » (framework) est un ensemble de composants réutilisable qui sert de fondation à la création de codes
  - Il propose une architecture, des fondations pour l'organisation du logiciel, les structures de données à la base du code
  - Il inclus des bibliothèques adaptées à cette base
  - Il fourni souvent la fonction principale qui appelle le code spécifique
  - C'est le cadriciel qui structure l'application et appelle le code

Exemples de bibliothèques

- Transformées de Fourier
  - FFTW, FFTSS, GSL, …
- Algèbre linéaire
  - LAPACK, PETSc, MUMPS, PASTIX, DPLASMA, …
- Écriture disque
  - POSIX, HDF5, NetCDF, SIONlib, MPI I/O, PDI, …
- Paramétrage
  - INI, XML, JSON, …
- Passage de messages
  - MPI

Exemples de frameworks

- Trilinos (https://trilinos.org/)
  - Structures de données parallèles
  - Outils de pré-processing, de calcul, solveurs linéaires, ...
- Salome (http://www.salome-platform.org/)
  - Pré- et post-processing : CAO, maillage, visualisation, ...
  - Interface utilisateur graphique ou texte (python)

Bibliothèque vs. Framework

Exemple pour un problème « *stencil* »

- Bibliothèques utiles
  - Communications entre nœuds (p.ex. MPI)
  - Parallélisme mémoire partagée (p.ex. Pthread)
  - Tableaux multi-dimensionnels (p.ex. Kokkos)
  - Écriture sur disque (p.ex. HDF5)
  - …

    ⇒ On écrit le code qui utilise judicieusement ces bibliothèques
- Framework (p.ex. YASK)
  - On fournit la matrice de stencil

    ⇒ On laisse le framework appliquer le stencil aux données

Bibliothèques : +/-

- (+) Avantages :
  - Moins de code à écrire
  - Souvent bien optimisé, maintenu
  - Beaucoup de choix disponibles

- (-) Inconvénients :
  - Parfois lourd pour une petite fonctionalité
  - Des hypothèses potentiellement pas adaptées
  - Beaucoup de choix disponibles

Interface de bibliothèque

- API : *Application Programming Interface*
- ~ Contenu du fichier d'en-tête
  - Ensemble de fonctions, structure de données, etc.
- Comment appeler ces fonctions depuis le code
  - Le résultat de la compilation peut changer
    - p.ex. Fonction vs. Macro
    - ABI : *Application Binary Interface*
- Plusieurs bibliothèques ⇔ une API (p.ex. MPI)
- Une bibliothèque ⇔ plusieurs APIs (p.ex. HDF5)

Conception d'interface

- Favoriser la conception modulaire
- Code = ensemble de modules indépendants
  - Minimiser la surface de l'interface
- Encapsulation : l'API doit décrire
  - Le **QUOI**
  - Pas le **COMMENT**
- Séparation de préoccupations

Séparation de préoccupations

In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate. A program that embodies SoC well is called a modular program. Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface. Encapsulation is a means of information hiding.

The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well-separated, individual sections can be reused, as well as developed and updated independently. Of special value is the ability to later improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections.

https://en.wikipedia.org/wiki/Separation_of_concerns

Exemple : point de reprise

- Point de reprise
  - Sauvegarde de l'ensemble des données nécessaires à la reprise du calcul
  - Tolérance aux pannes, exécution longue, …
- Potentiellement à chaque itération
  - Souvent moins fréquent (performance)
- Conservation des N derniers fichiers
- Peut tirer parti de matériel avantageux

API I/O (ANSI C)

- `FILE *fopen(const char *pathname, const char *mode);`

- ```
  size_t fwrite(const void *ptr, size_t size,
      size_t nmemb, FILE *stream);
  ```

- `int fclose(FILE *stream);`
  ```

API I/O (HDF5)

- Les concepts :
  - Fichier HDF5 : 1 datagroup racine
  - Datagroup (répertoire) : liste nommée de
    - Dataset et datagoups
  - Dataset (fichier) : tableau multi-D de données
    - Datatype
    - Dataspace (rank, sizes, max sizes)
- 2 niveaux d’API
  - HDF5 complet
  - HDF5-HL simplifié

API I/O (HDF5-HL)

- `hid_t H5Fcreate( const char *name, unsigned flags,`
  `hid_t fcpl_id, hid_t fapl_id )`

- `herr_t H5LTmake_dataset_double ( hid_t loc_id,`
  `const char *dset_name, int rank,`
  `const hsize_t *dims, const double *buffer )`

- `herr_t H5Fclose( hid_t file_id )`

API I/O (FTI)

- `int FTI_Init ( char * configFile , MPI_Comm comm )`
- `int FTI_Protect ( int id, void *ptr, long count, FTIT_type type )`
- `int FTI_Snapshot()`
- `int FTI_Finalize()`

Bibliothèques : en pratique

- Un fichier avec la mise en œuvre (**`.a`** ou **`.so`**)
  - Contient le code des fonctions fournies
  - Un en-tête
    - Combien d'espace réserver pour les variables globales
    - Adresse de chaque fonction, variable : **symboles**
- Un fichier d'en-tête pour le code (**`.h`** ou **`.mod`**)
  - Contient des informations sur les symboles
    - Type des variables globale
    - Signature des fonctions
    - Macros et fonctions inline

Compiler avec une bibliothèque

- Le code source référence le fichier d'en-tête
  - `#include` / `use`
  - Permet de vérifier le type des variables
  - Permet de savoir comment appeler les fonctions
  - Option `-I` `cc -I /usr/include/ -c -o tst.o tst.c`
  - Techniquement pas 100 % nécessaire
    - Mais 100 % obligatoire pour éviter les erreurs
- Le code généré dans le **`.o`**
  - Accède aux fonctions / variables via une fausse adressse
  - Contient un en-tête des **symboles utilisés**

Lier avec une bibliothèque .a

- Fichier **`.a`** = archive de **`.o`**
  - Chaque fichier **`.o`**
    - Symboles fournis (adresse dans ce fichier)
    - Symboles utilisés (adresse utilisée dans le fichier)
- Édition de liens
  - Associer symbole fournis et utilisés
- En pratique
  - `-l` nom `cc -lm -o tst tst.o`
  - `-L` répertoire `cc -L /usr/lib/ -lm -o tst tst.o`
  - Erreur si pbm `undefined reference to symbol 'lib_func'`

Lier avec bibliothèque .so

- Un seul fichier avec code et symboles
  - `objdump -T mylib.so`
- L'exécutable référence le **`.so`** mais ne l'inclus pas
  - `ldd myexe`
- Association symboles dynamique
  - Comme pour un **`.a`**, mais à l'exécution
- Répertoire de chargement spécifié via
  - Configuration système (Cf. `/etc/ld.so.conf`)
  - Environnement `export LD_LIBRARY_PATH="./"`

Make

- Objectif : exécution de commande
  - Seulement si nécessaire
- Commande : make
  - Lit le fichier : « Makefile »

```
target: source1 source2
<TAB> command_to_gen -o target source1 source2
```

  - Executer la commande si
    1) Le fichier « target » n'existe pas
    2) Un des fichiers « source » a été modifié plus récemment que « target »

cmake

- Génération de Makefile
  - Détection des bibliothèques & dépendances
  - Automatisation des aspects complexes & répétitifs
- Un langage de script comme bash
  - Boucles : (`foreach`) , Conditions : (`if`), etc.
  - Variables (`set(VARIABLE <VALUE>`) , `$VAR`)
- Commande « cmake »
  - Lit un fichier « CmakeLists.txt »
  - p.ex.

```
cd build_dir
cmake ${source_dir}
make
```

- Cf https://cmake.org/cmake/help/latest/

« Modern » cmake

- 2 types de variable :
  - ~~Chaines de caractères~~
  - « targets » (exécutables, bibliothèques, etc.)
- Une target comporte :
  - Une liste de répertoires à inclure
  - Une liste de bibliothèques à lier
  - Les éventuelles options de compilation à utiliser
  - **Les targets à intégrer par transitivité : p.ex. Une bibliothèque qui dépend d'une autre**
  - Chaque inclusion, lien, etc. Peut être transitif (PUBLIC) ou non (PRIVATE)

« Modern » cmake en pratique

- En-tête
  - Version de cmake
  - Nom du projet + langages
- `find_package`
  - Cherche une dépendance
  - Met à disposition des « targets »
- `add_executable` / `add_library`
  - Crée une nouvelle « target » : `ex1`
- `target_link_libraries`
  - Lie des « targets » ensembles

```
cmake_minimum_required(
  VERSION 3.19
)
project(TD1 C CXX)


find_package(MPI REQUIRED
  COMPONENTS CXX)



add_executable(ex1
  ex1.c
)

target_link_libraries(ex1
  MPI::MPI_CXX
)
```