1 Programmation modulaire

*Algorithmique et Programmation C**++ 2***

Introduction

Lors de la réalisation du premier projet technologique de *Data-Mining*, la plupart des groupes ont utilisé un seul fichier source, considérant que pour un programme donné, un fichier source suffisait. Mais cette technique a de nombreux inconvénients :

- Impossible de travailler à plusieurs sur un seul fichier source ;
- Réutilisabilité quasiment nulle et reste faible si on dispose des sources ;
- Inefficacité pour la compilation : il faut recompiler systématiquement tout le projet à chaque modification ;
- Correction et maintenances difficiles.

Modules et librairies

Les modules

L'objectif des modules est de **structurer** les sources :

- faire des composants logiciels autonomes
- indépendance des composants (dépendances explicites)
- séparation claire de l'interface et de l'implémentation. L'implémentation sera compilée, et l'interface servira aux développeurs pour utiliser l'implémentation dont les détails sont inutiles.

Un **module** est un fichier d'implémentation, d'extension *.cc*, accompagné d'un ou plusieurs fichiers d'en-tête (interface). C'est une unité de compilation, c'est-à-dire qu'il ne peut être compilé par morceaux.

En C++, on confond interface et déclaration, mais ce n'est pas la même chose dans d'autres langages. Ainsi, en C++, la déclaration d'une classe inclut les membres privés, alors que, comme leur nom l'indique, ils ne font pas partie de l'interface, puisqu'ils ne sont pas accessibles de l'extérieur.

On distingue :

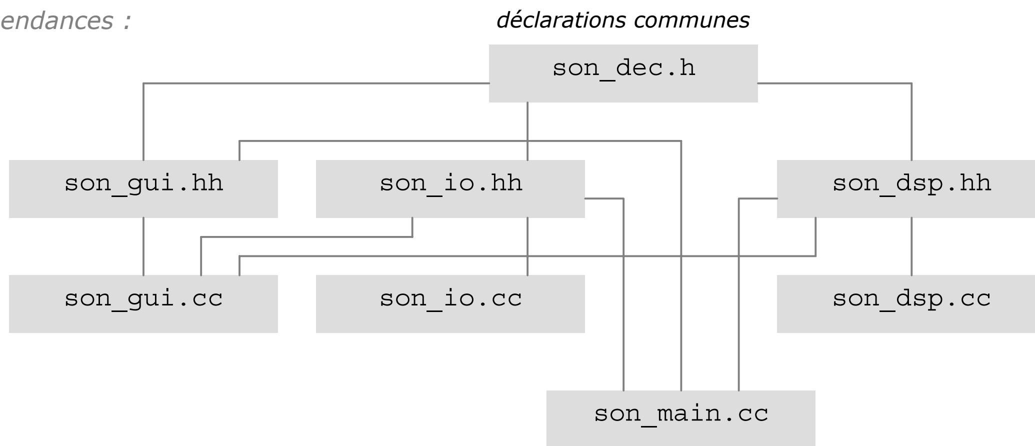
- l'interface (fichiers *.h* ou *.hh*) : ce sont uniquement les déclarations, sans code, sans définitions de variables globales, sans création d'objet. Un fichier d'en-tête est seulement composé des déclarations de types, de classes, de macros, de déclarations de variables et de fonctions externes.
- l'implémentation (fichiers *.cc*) : le code, sans oublier l'inclusion des fichiers d'en tête.

Exemple :

Imaginons que nous travaillons avec un composant pour la manipulation des fichiers sons pour Unix, composé de 4 modules :

- son_io : entrée-sortie
- son_dsp : traitements acoustiques
- son_gui : interface graphique
- son_main : programme principal

Dépendances :



Les librairies

Les **librairies** regroupent un ensemble de modules. Dans une implémentation complète de C++, on trouve un certain nombre de librairies :

- librairies C++ standard (*libc*) ;
- librairies mathématiques ;
- etc.

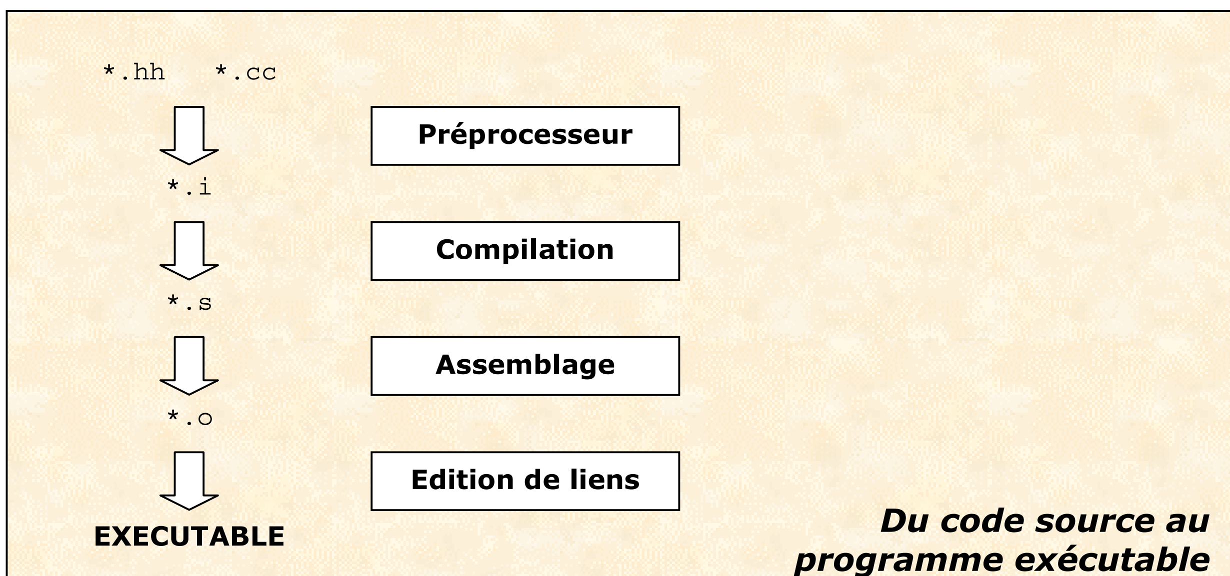
Il y a différents types de librairies :

- les archives (fichiers *.a*) qui sont statiques
- les librairies dynamiques dont le code est partagé entre les différents utilisateurs, et dont le code est **réentrant**, c'est-à-dire exécutable simultanément par plusieurs utilisateurs.

Construction d'une librairie
Regroupement des modules :
`ar <commande> <nom librairie> <liste des modules>`
Création de la table des matières (optionnelle) :
`ranlib <nom librairie>`

Les librairies sont utilisées en incluant les en-têtes des modules utilisés, ce qui provoque le 'linkage' de la librairie.

Exemple :
`aCC son_main.cc –lson`
Ceci provoque l'inclusion de `libson.a`



Quelques outils de développement dans un environnement Unix

Make

- Génération de commandes à partir de fichiers de dépendances.
- Cet outil n'est pas spécifique à la compilation
- Gestion des mises à jour basée sur l'heure système (synchronisation des serveurs, etc.)

Makefile
Forme générale d'un fichier de dépendance :
`<cible> : <liste de fichiers>` `<tab> <commande>`

Exemple de Makefile :

```
son : son_io.o son_gui.o son_dsp.o son_main.cc
  aCC -o son son_main.cc son_gui.o son_dsp.o son_io.o
```

On compile *son* après avoir vérifié les mises à jour des dépendances, et si une des dépendances possède une date de dernière modification plus récente que *son*, alors le fichier est recompilé, en exécutant la ligne située juste en dessous.

```
son_io.o : son_io.h son_io.cc son_dec.h
  aCC -c son_io.cc
son_gui.o : son_gui.h son_gui.cc son_dex.h
  aCC -c son_gui.cc
son_dsp.o : son_dsp.h son_dsp.cc son_dec.h
  aCC -c son_dsp.cc
```

Make
Utilisation de la commande Make :
`make –f <nom de fichier>`

Si on tape `make` tout court, cela revient a taper `make –f makefile`.

Définition des variables dans un Makefile
Déclaration et initialisation :
`<id>=<valeur>`
Utilisation (évaluation) :
`$(<id>)`

Exemple :

<u>Sous Unix :</u>

```
CC=aCC
OPTS=-or
```

<u>Sous Linux :</u>

```
CC=g++
OPTS=-04 -m pentiumpro
```

<u>Partie commune :</u>

```
son : son_io.o son_gui.o son_dsp.o son_main.cc
  $(CC) $(OPTS) son_main.cc son_gui.o son_io.o son_dsp.o
```

Remarque : **Make** n'est pas forcément associé au langage C et à une plate-forme de type Unix. On peut l'utiliser par exemple pour compiler des programmes Java sur n'importe quelle type de plate-forme, notamment sous Linux.

Le Debugger

Sur un projet, la partie correction des bugs peut prendre du temps. Un **debugger** permet d'accélérer grâce à ses fonctionnalités :

- aide à la correction du programme
- mode pas à pas
- trace du contenu des variables
- retour arrière : revenir en arrière dans l'exécution n'est pas toujours possible
- interruption de l'exécution par des breakpoints, qui peuvent être en fonction de la valeur d'une expression (pas à pas dans une passage donné d'une itération, par exemple)
- analyse des fichiers core (segmentation fault) pour retrouver l'état du programme au moment du plantage

Le Debugger s'utilise en compilant avec l'option `–g` : `aCC –g –o exemple exemple.c`

Le Profiler

C'est un outil qui permet d'analyser les performances d'un programme :

- temps consommé par les différentes fonctions,
- nombre d'appels.

Principe :

- inclusion de traces dans le code compilé,
- production automatique d'un rapport.

Utilisation :

```
aCC -G -o exemple exemple.cc
g++ -p -o exemple exemple.cc
```

Exécution :

- analyse à posteriori ;
- dépendante des paramètres du programme (entrés par l'utilisateur parfois) ;
- production du rapport `gmon.out`

Pour la mise en forme du rapport, utiliser par exemple *Gprof*.

Exemple :

```
gmon.out > exemple.gprof
```

Important : Quelque soient les optimisations du code que l'on fera, elles seront toujours fortement négligeables par rapport aux optimisations algorithmiques qui pourront être effectuées.

Gestionnaire de versions

Le **gestionnaire de versions** permet de :

- conserver et restituer les différentes versions des composants,
- maintenir l'historique des modifications,
- gérer les conflits d'accès
- distribuer les versions (avec un accès éventuellement sécurisé)

Citons par exemple des outils tels que : *rCS*, *cVS*, ou *SCCS*.

Outils de configuration

autoconf : Recherche la configuration de l'ordinateur sur à peu près n'importe quelle plateforme.

automake : A partir du fichier de configuration, crée un makefile

Principe :

- adaptation au contexte de compilation et d'exécution,
- génération automatique de fichiers de configuration et de makefiles.

Divers

- **Documentation** (à partir de l'analyse des sources et des commentaires) : génération automatique en *latex*, en *html*, en *sgml*, citons *doc++*, *doxygen++* sous C et *idl* sous C++ ;
- **Mise en page du code** : *lclint* (formatte les codes C++) ;
- **Générateurs d'interfaces** (comme sous *Visual Basic*) ;
- **Outils de modélisation** (génère le code en fonction du schéma de dépendance).