Programmation Modulaire

Module de Programmation Avancée

L. Gonnord & J. Dequidt - Polytech'Lille

Constat

- Difficulté de gérer un logiciel de grande ampleur (>> 10'000 lignes de code)
  - répartition du travail entre plusieurs personnes
  - temps de compilation
  - maintenance du code
  - réutilisation du code
  - [...]

Génie Logiciel

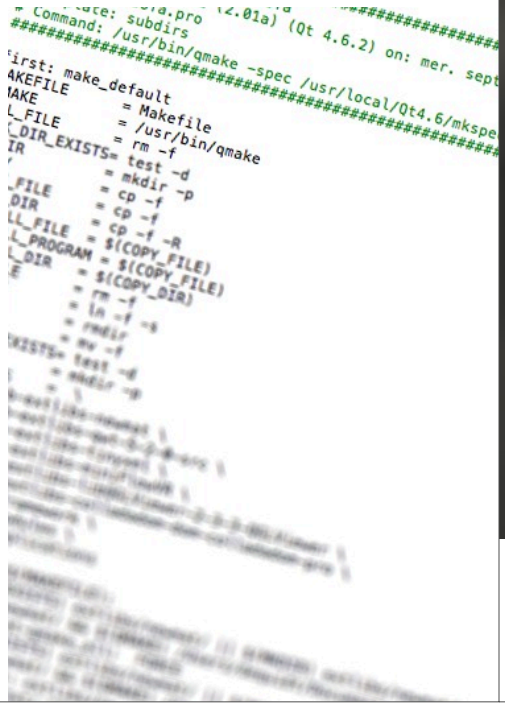
- Génie Logiciel: branche de l'informatique qui étudie les méthodes de travail et les bonnes pratiques des développeurs.
- Couvre les différents stades de la réalisation d'un logiciel
  - Analyse / Spécifications
  - Conception
  - Compilation
  - Optimisation / Tests / Maintenance ...

Programmation Modulaire

- Idée de base est de découper le code en différents "modules" pour permettre:
  - Meilleure lisibilité
  - Réutilisation de modules (*bibliothèques*)
  - Diviser / structurer / hiérarchiser le code
- Optimiser la compilation (uniquement les modules modifiés sont recompilés)
- Favoriser le travail en équipe
- Facilité de l'entretien / maintenance
- Possibilité de tests sélectifs
- Dissimulation du contenu des fonctions

Plan

- Quelques principes
- La compilation séparée
- L'outil make
- L'utilisation / La création de bibliothèques



Quelques principes

1. Utilisation de constantes symboliques

Exemple élementaire

```
double aire_disque(double rayon)
{
    double aire;
    aire = 3.141516927 * rayon * rayon;
    return aire;
}

double perimetre_cercle(double rayon)
{
    double perimetre;
    perimetre = 3.141516927 * rayon * 2.0;
    return perimetre;
}

double degre_vers_radian(double angle)
{
    double radian;
    radian = 3.141516927 * angle / 180.0;
    return radian;
}

double degre_vers_radian(double angle)
{
    double radian;
    radian = 180 * angle / 3.141516927;
    return radian;
}
```


1. Utilisation de constantes symboliques

Exemple élementaire

1. Utilisation de constantes symboliques

Exemple élementaire

```c
#define PI 3.1415927

double aire_disque(double rayon)
{
    double aire;
    aire = PI * rayon * rayon;
    return aire;
}

double perimetre_cercle(double rayon)
{
    double perimetre;
    perimetre = PI * rayon * 2.0;
    return perimetre;
}

double degre_vers_radian(double angle)
{
    double radian;
    radian = PI * angle / 180.0;
    return radian;
}

double degre_vers_radian(double angle)
{
    double radian;
    radian = 180 * angle / PI;
    return radian;
}
```


2. Factorisation de Code

- Objectif: éviter les duplications de code (difficile à maintenir) et favoriser la réutilisation
- Création de fonctions (même petites) pour généraliser des suites d'instructions qui peuvent être utiles à plusieurs endroits d'un programme

```c
void ma_fonction()
{
    /* Déclaration */
    int i;
    double u[3], v[3];
    double w[9], y[9];
    double prod_scalaire_3, prod_scalaire_9 = 0.0;

    /* Initialisation */
    [...]

    /* Calcul du produit scalaire */
    prod_scalaire_3 = 0.0;
    for(i = 0; i < 3; i++)
         prod_scalaire_3 += u[i] * v[i];

    prod_scalaire_9 = 0.0;
    for(i = 0; i < 9; i++)
         prod_scalaire_3 += w[i] * y[i];
}
```

```c
double produit_scalaire(double * a, double * b, int
n)
{
    int i;
    double resultat = 0.0;
    for(i = 0; i < n; i++)
         resultat += a[i] * b[i];
    return resultat;
}

void ma_fonction()
{
    /* Déclaration */
    int i;
    double u[3], v[3];
    double w[9], y[9];
    double prod_scalaire_3, prod_scalaire_9 = 0.0;

    /* Initialisation */
    [...]

    /* Calcul du produit scalaire */
    prod_scalaire_3 = produit_scalaire(u,v,3);
    prod_scalaire_9 = produit_scalaire(w,y,9);
}
```

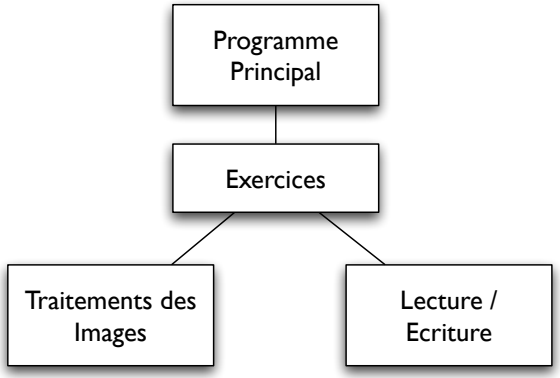

3. Fragmentation de Code

- Découpage en plusieurs fichiers:
  - un fichier principal (“main.c”)
  - des fichiers auxiliaires avec à chaque fois
    - un fichier en-tête (*header*) .h pour les déclarations de fonctions. Ce fichier en-tête sera
    - un fichier source .c pour les définitions de fonctions

3. Fragmentation de Code

- Exemple :
  projet de programmation structurée IMA3 S5 (programme de traitement d'image)
- Décomposition cohérente du programme en *blocs fonctionnels*

3. Fragmentation de Code



- lecture_ecriture.h
  lecture_ecriture.c
- traitement_image.h
  traitement_image.c
- exercices.h
  exercices.c
- main.c

3. Fragmentation de Code

lecture_ecriture.h

```
int chargeImage(const char * nom_fichier, unsigned char ** tableau, int * largeur, int * hauteur);
int sauvegardeImage(const char * nom_fichier, const unsigned char * tableau, int largeur, int hauteur);
```

traitement_image.h

```
void minimaxi(const float * r, float * m, float * M, int * argmin, int * argmax);
void pixelRVBVersTSV(const unsigned char * pixel, float * dest);
void pixelTSVVersRVB(const float * pixel, unsigned char * dest);
int RVBversTSV(const unsigned char * img_source, int largeur, int hauteur, float ** img_dest);
int TSVversRVB(const float * img_source, int largeur, int hauteur, unsigned char ** img_dest);
```

exercices.h

```
/* Q3.1 */
int recopie(const unsigned char * img_source, int largeur, int hauteur);
/* Q3.2 */
int genereBruit(float proba, unsigned char * img_source, int largeur, int hauteur, unsigned char * img_dest);
/* Q3.3 */
int vieillirImage(float * img_source, int largeur, int hauteur);
/* Q3.4 */
int recadrage(float * img_source, int largeur, int hauteur, int orig_X, int orig_Y, int fin_X, int fin_Y);
/* [...] */
```


3. Fragmentation de Code

main.c

```
int main()
{
   /* Déclarations des variables */
   int i, l, h, taille, resultat;
   unsigned char * vect, * dest;

   /* Initialisation du générateur de nombres aléatoires */
   srand((int)time(NULL));

   /* Q0- On charge une image */
   if(!chargeImage("Orig_Lille_Color.ppm", &vect, &l, &h))
        return -1;

   /* Q1- Recopie */
   resultat = recopie(vect, l, h);

   /* [...] */

   return 0;
}
```

Compilation Séparée

Compilation Séparée

- Objectif: Obtenir un exécutable à partir de plusieurs fichiers
- Procédé en deux étapes
  - 1) Génération des fichiers objets pour chacun des fichiers .c

```
gcc -c lecture_ecriture.c
gcc -c traitement_image.c
gcc -c exercices.c
gcc -c main.c
```

  - 2) Edition de liens (= assemblage) pour obtenir l'exécutable

```
gcc -o projetIMA09 main.o lecture_ecriture.o traitement_image.o exercices.o
```

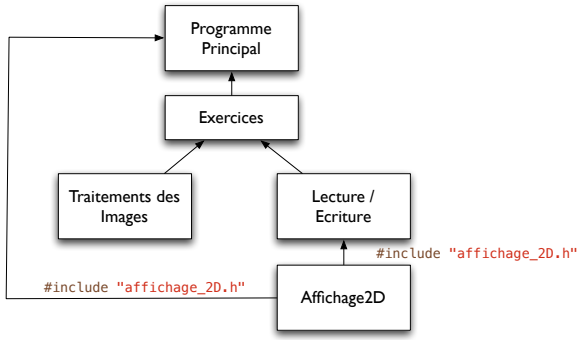

Compilation Séparée

- **Attention**: pour être compilés, chacun des fichiers doit connaître le prototype des fonctions qu'il utilise.

```
#include "lecture_ecriture.h"
#include "traitement_image.h"

/* Q3.1 */
int recopie(const unsigned char * img_source, int largeur, int hauteur);
/* Q3.2 */
int genereBruit(float proba, unsigned char * img_source, int largeur, int hauteur, unsigned char *
img_dest);
/* Q3.3 */
int vieillirImage(float * img_source, int largeur, int hauteur);
/* Q3.4 */
int recadrage(float * img_source, int largeur, int hauteur, int orig_X, int orig_Y, int fin_X, int
fin_Y);
/* [...] */
```

- Rappel:
  `#include "fichier.h"` si fichier.h est dans le répertoire courant
  `#include <fichier.h>` si fichier.h est dans les répertoire d'inclusions par défaut (`/usr/include/, /usr/local/include/`,...) ou ceux ajoutés par l'utilisateur (on verra ça plus tard).

Inclusions multiples



Résultat GCC:

```
traitement_image.o(.data+0x0): multiple definition of `affichePixel'
main.o(.data+0x0): first defined here
[...]
```

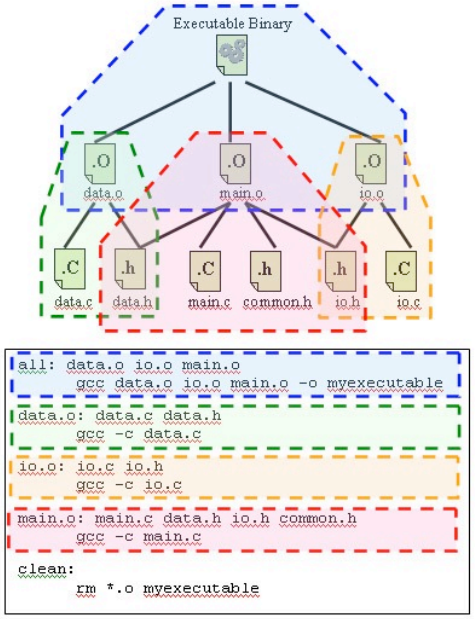

Inclusions multiples

- Solution: Include / Header **guards**

```
#ifndef __AFFICHAGE_2D_H__
#define __AFFICHAGE_2D_H__

void affichePixel(int x, int y, float * couleur);

/* [...] */

#endif
```

- La première fois que ce fichier est inclus, la variable `__AFFICHAGE_2D_H__` sera définie et le contenu du fichier "*copié*". A l'appel d'autres inclusions, le test ifndef sera faux donc le contenu du fichier ne sera pas inclus.

L'outil **make**



Objectifs

- Automatiser la procédure de compilation (= éviter d'avoir à écrire toutes les lignes de compilation)
- Optimiser la compilation (= ne recompiler que ce qui est nécessaire)
- Pouvoir facilement compiler / déployer des gros projets logiciels
- Un certains nombre d'outils "make" existent: GNU make, qmake, nmake, Scons ...
- Ici, on s'attarde à GNU make

GNU make

- Se base sur un fichier qui contient les règles de compilation
- La syntaxe est make -f nom_de_mon_fichier
- Par défaut ce fichier s'appelle `Makefile (make == make –f Makefile)`
- Un fichier make est constitué d'une liste de règles de dépendance

```
cible: liste_des_fichiers_dependants
    [TABULATION] commandes_a_effectuer
```


Exemple: un makefile minimal

```
# Ceci est un commentaire
projetIMA09: lecture_ecriture.o traitement_image.o exercices.o main.o
    gcc -o projetIMA09 lecture_ecriture.o traitement_image.o exercices.o main.o

lecture_ecriture.o: lecture_ecriture.c lecture_ecriture.h
    gcc -c lecture_ecriture.c

traitement_image.o: traitement_image.c traitement_image.h
    gcc -c traitement_image.c

exercices.o: exercices.c exercices.h
    gcc -c exercices.c

main.o: main.c
    gcc -c main.c
```

Ajout de variables

```
CC=gcc
CFLAGS= -W -Wall
LFLAGS=
DEL_FILE = rm -rf
EXEC= projetIMA09

all: $(EXEC)

projetIMA09: lecture_ecriture.o traitement_image.o exercices.o main.o
    $(CC) -o projetIMA09 lecture_ecriture.o traitement_image.o exercices.o main.o $(LFLAGS)

lecture_ecriture.o: lecture_ecriture.c lecture_ecriture.h
    $(CC) -c lecture_ecriture.c $(CFLAGS)

traitement_image.o: traitement_image.c traitement_image.h
    $(CC) -c traitement_image.c $(CFLAGS)

exercices.o: exercices.c exercices.h
    $(CC) -c exercices.c $(CFLAGS)

main.o: main.c
    $(CC) -c main.c $(CFLAGS)

clean:
    $(DEL_FILE) *.o

mrproper: clean
    $(DEL_FILE) $(EXEC)
```


Variables internes

$@	Le fichier cible (main.o par exemple)
$*	Le fichier cible sans extension (main avec l'exemple précédent)
$<	Le premier fichier de la liste de dépendances
$?	L'ensemble de fichiers de la liste de dépendances

Variables Internes

```
CC=gcc
CFLAGS= -W -Wall
LFLAGS=
DEL_FILE = rm -rf
EXEC= projetIMA09

all: $(EXEC)

projetIMA09: lecture_ecriture.o traitement_image.o exercices.o main.o
    $(CC) -o $@ $^ $(LFLAGS)

lecture_ecriture.o: lecture_ecriture.c lecture_ecriture.h
    $(CC) -c $< $(CFLAGS)

traitement_image.o: traitement_image.c traitement_image.h
    $(CC) -c $< $(CFLAGS)

exercices.o: exercices.c exercices.h
    $(CC) -c $< $(CFLAGS)

main.o: main.c
    $(CC) -c $< $(CFLAGS)

clean:
    $(DEL_FILE) *.o

mrproper: clean
    $(DEL_FILE) $(EXEC)
```


Règles d'inférences / Implicit Rules

- Objectif éviter d'avoir à écrire n-fois `$(CC) -c $< $(CFLAGS)`
- On utilise des règles implicites
  - exemple:

```
.cpp.o:
$(CXX) -c $(CXXFLAGS) $(INCPATH) -o "$@" "$<"

.cc.o:
$(CXX) -c $(CXXFLAGS) $(INCPATH) -o "$@" "$<"

.cxx.o:
$(CXX) -c $(CXXFLAGS) $(INCPATH) -o "$@" "$<"

.C.o:
$(CXX) -c $(CXXFLAGS) $(INCPATH) -o "$@" "$<"

.c.o:
$(CC) -c $(CFLAGS) $(INCPATH) -o "$@" "$<"
```

- Il est possible de rajouter d'autres dépendances

Règles d'inférences / Implicit Rules

- Ex: la génération de traitement_image.o dépend de traitement_image.c mais aussi de traitement_image.h
- La règle `.c.o: $(CC) -c $(CFLAGS) $(INCPATH) -o "$@" "$<"` permet de faire dépendre traitement_image.o de traitement_image.c
- On peut ajouter `traitement_image.o: traitement_image.h $(CC) -c traitement_image.c $(CFLAGS) $(INCPATH)` pour intégrer la deuxième dépendance.

Forcer la reconstruction de certaines cibles

- Cas où une cible n'a pas de dépendance.
- Si jamais cette cible correspond à un fichier du répertoire courant, forcément ce fichier est plus récent que les dépendances .... donc ne sera jamais reconstruit

```
.PHONY: clean mrproper

clean:
   $(DEL_FILE) *.o

mrproper: clean
   $(DEL_FILE) $(EXEC)
```


Conditions / Commandes Silencieuses

- Ajout de conditions

```
ifeq ($(DEBUG),yes)
   CFLAGS=-W -Wall -ansi -g
else
   CFLAGS=-W -Wall -ansi
endif
```

- Commandes Silencieuses:
  - pour éviter de générer trop d'affichage à l'écran.
  - utilisation de @

```
clean:
   @$(DEL_FILE) *.o

mrproper: clean
   @$(DEL_FILE) $(EXEC)
```


Générartion automatique de la liste des fichiers ...

- ... OBJET

```
SRC= lecture_ecriture.c traitement_image.c exercices.c main.c
OBJ= $(SRC:.c=.o)

projetIMA09: $(OBJ)
   $(CC) -o $@ $^ $(LFLAGS)
```

- ... SOURCE

```
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

projetIMA09: $(OBJ)
   $(CC) -o $@ $^ $(LFLAGS)
```

Makefile / Sous makefile

- Utile quand le projet contient des modules sous formes de repertoires

```
CC=gcc
CFLAGS=  -Wall -pipe
LDFLAGS=
DEL_FILE = rm -rf
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)
SUBDIR= module1 module2

.SUFFIXES: .c
.c.o:
    $(CC) -c $(CFLAGS) $<

all:
    for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
    $(MAKE) projetIMA09

projetIMA09: $(OBJ)
    $(CC) -o $@ $(LDFLAGS) $^

clean:
    @$(DEL_FILE) *.o
    for i in $(SUBDIRS); do (cd $$i; $(MAKE) clean); done
```

Bibliothèques

Utilisation / Création

Bibliothèques

- Module réutilisable et partageable, inutile de donner tout le code source
- Vous avez déjà utilisé des bilbiothèques
  - libc (bibliothèque standard) qui contient printf, scanf, ...
  - libm (bibliothèque de mathématiques) qui contient ceil, floor, ...
- Spécification de la bibliothèque maths : gcc [...] -lm
- Contenu d'une librairie: un fichier en-tête (.h) et du code binaire
- Pour la compilation -L pour spécifier un répertoire de recherche de lib. -l pour utiliser une lib

Statique / Dynamique

- Deux possibilités pour les bibliothèques:
  - Statique:
    - lib*****.lib (sous Windows) ou lib****.a sous linux / Mac OS
    - la bibliothèque est liée (= incluse) à l'exécutable. Exécutable relativement lourd, une modif dans la lib = recompilation de la lib et de l'exécutable. La bibliothèque n'est pas réellement partagée mais pas de problème de dépendances.
  - Dynamique:
    - lib****.dll (sous Windows) lib****.so (sous Linux) lib****.dylib sous Mac OS
    - La lib est liée à l'exécution: exécutable plus léger mais lancement plus long
    - La lib n'est chargée qu'une seule fois en mémoire même si utilisée par plusieurs programmes
    - Si une modif est faite sur la lib, l'exécutable n'a pas à être recompilé (sauf si modif sur l'interface de la lib).

Bilbiothèque Statique : exemple simple

```c
/* fichier test1.h */
void ctest1(int *i);

/* fichier test1.c */
#include "test1.h"

void ctest1(int *i)
{
    *i=5;
}

/* fichier test2.h */
void ctest2(int *i);

/* fichier test2.c */
#include "test2.h"

void ctest2(int *i)
{
    *i=100;
}

/* fichier prog.c */
#include <stdio.h>
#include "test1.h"
#include "test2.h"

int main()
{
    int x;
    ctest1(&x);
    printf("Valx=%d\n",x);

    return 0;
}
```

```
# compilation des fichiers de la lib
gcc -Wall -c ctest1.c ctest2.c

# creation de la lib
ar -cvq libctest.a ctest1.o ctest2.o

# lister les fichiers .o dans la lib
ar -t libctest.a

# Compiler prog avec la lib, 2 possibilités
gcc -o prog prog.c libctest.a
gcc -o prog prog.c -L. -lctest
```

__.SYMDEF SORTED
test1.o
test2.o

```
nm -s libctest.a

libctest.a(test1.o):
0000000000000000 T _test1

libctest.a(test2.o):
0000000000000000 T _test2
```

permet de voir la liste des symboles dans la lib. T présent dans la lib, U absent...

Bibliothèque dynamique: même exemple

```
# Compilation
#  option -fPIC pour Position Independant Code (requis pour les SO)
gcc -Wall -fPIC -c *.c
# Creation de la lib
gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0   *.o
# Déplacement (optionnel)
mv libctest.so.1.0 /opt/lib
# Création de liens symboliques
# pour pouvoir utiliser -lctest
ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so
# pour pouvoir lier à l execution
ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1

# Compilation du programme
gcc -Wall -L/opt/lib prog.c -lctest -o prog
```

```
# Voir la liste des dépendances
ldd prog

libctest.so.1 => /opt/lib/libctest.so.1 (0x00002aaaaaaac000)
libc.so.6 => /lib64/tls/libc.so.6 (0x0000003aa4e00000)
/lib64/ld-linux-x86-64.so.2 (0x0000003aa4c00000)

# Executer
export LD_LIBRARY_PATH=/opt/lib:$LD_LIBRARY_PATH
./prog
```