

Techniques de Développement

La compilation

The logo of the University of Strasbourg, featuring two blue curved segments forming a stylized 'S' shape.

UNIVERSITÉ DE STRASBOURG

Licence de sciences mention informatique

2022

- 1 Organisation du code
- 2 Un exemple simple
- 3 Les mécanismes de la compilation
- 4 GCC et makefiles évolués
- 5 Bibliothèques

Organisation du code

- Séparation en fichiers (compilation séparée).
- Séparation des headers et des sources.
- Utilisation de répertoires.
- Utilisation de bibliothèques (libraries, ou libs).

Séparation en fichiers

On regroupe dans un fichier les fonctions correspondant à une application, ou un type d'objets utilisés, etc.

Intérêt :

- Améliorer la lisibilité du code.
- Ré-utiliser / mutualiser certains modules.
- Éviter de tout recompiler (long pour les gros projets).

La **difficulté** principale est la gestion des dépendances :

- S'assurer que les fonctions d'un fichier sont connues d'un autre fichier (solution : `#include`).
- Éviter les définitions multiples (solution : séparation `.h` et `.c`).
- Éviter les déclarations multiples (solution : `#ifndef`).
- Compiler tous les fichiers utiles et seulement eux (solution : `makefile`).

Séparation des headers et des sources

On regroupe les déclarations et les macros dans un header (`.h`), et les définitions dans un fichier source (`.c`).

Intérêt :

- Améliorer la lisibilité du code (il suffit de lire les entêtes pour connaître les fonctions disponibles et leur prototype).
- Permettre la compilation séparée.
- Permettre la création de bibliothèques.

Difficulté :

- Des paires de fichiers à maintenir à jour.

Répertoires séparés

Dans différents répertoires, on répartit les fichiers par types : headers, sources, objets, libs, exécutables, etc.

Intérêt :

- Faciliter la recherche de fichiers (gros projets).
- C'est une convention de rangement.

Difficulté :

- Écrire les règles de compilation.

Les bibliothèques

Une bibliothèque est un assemblage de fichiers objet contenant des fonctions déjà compilées.

Intérêt :

- Faciliter la diffusion.
- Garder le code secret, tout en diffusant les headers pour le développement (paquets linux "dev").
- Assurer la stabilité du comportement.

Difficulté :

- Écrire les règles de compilation.
- Gérer les compatibilités de versions entre deux libs, ou entre une lib et un code.

Table des matières

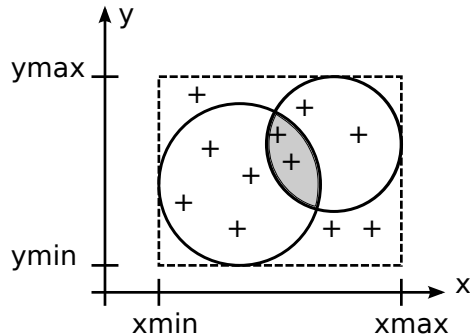
- 1 Organisation du code
- 2 Un exemple simple**
- 3 Les mécanismes de la compilation
- 4 GCC et makefiles évolués
- 5 Bibliothèques

L'algorithme

Méthode du crible

Pour calculer l'aire de l'intersection entre deux cercles, on tire des points aléatoirement dans le rectangle englobant.

La proportion de points qui tombent dans l'intersection donne la proportion d'aire par rapport à l'aire du rectangle.



Version 1 (brutale)

Organisation :

Tous les types, les déclarations (prototypes) et les définitions de fonctions sont dans `main.c`.

Compilation :

```
gcc -o main main.c -lm
```

Remarque : le code appelle des fonctions de la bibliothèque de maths :

Le fichier doit contenir `#include <math.h>`

L'option `-lm` demande au compilateur de faire le lien (link) avec la bibliothèque.

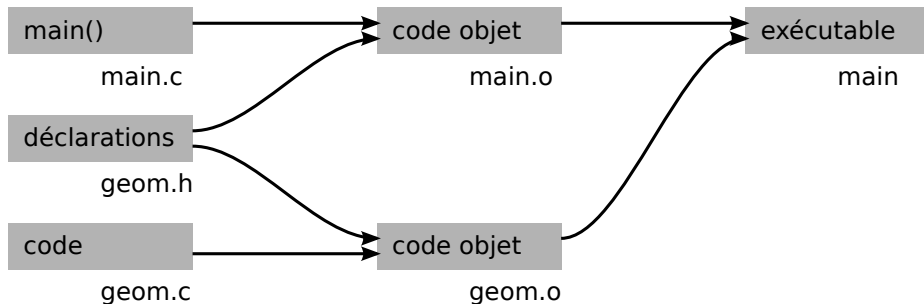
Organisation :

- `main.c` contient uniquement la fonction `main`.
- `geom.h` contient toutes les déclarations de types et de prototypes (fonctions).
- `geom.c` contient toutes les définitions de fonctions.

Conséquences :

- Les fichiers header et source doivent être séparés.
- `main.c` et `geom.c` doivent contenir `#include "geom.h"` pour que les types / prototypes soient connus à la création des fichiers objet.

Version 2 (correcte)



Compilation :

`gcc -c geom.c` qui crée `geom.o`

`gcc -c main.c` qui crée `main.o`

`gcc -o main main.o geom.o -lm` qui crée l'exécutable `main`

Remarque : l'option `-lm` n'a besoin d'apparaître que dans la dernière phase, appelée édition de lien.

Version 3 (meilleure)

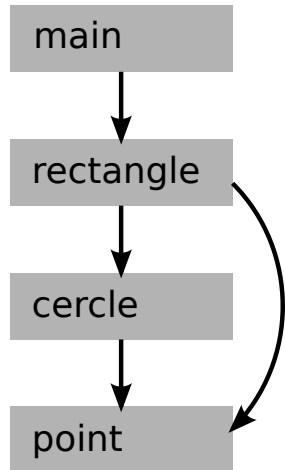
Organisation :

- `point.h` et `point.c`
- `cercle.h` et `cercle.c`
- `rectangle.h` et `rectangle.c`
- `main.c`

Conséquences :

- Pour éviter les déclarations multiples, on encapsule chaque `fichier.h` dans

```
#ifndef __FICHIER_H__
#define __FICHIER_H__
...
#endif
```



Version 3 (meilleure)

Difficultés :

- Beaucoup de fichiers à compiler (ici quatre `.o`, plus l'exécutable) :

```
gcc -c point.c
```

```
gcc -c cercle.c
```

```
gcc -c rectangle.c
```

```
gcc -c main.c
```

```
gcc -o main main.o point.o cercle.o rectangle.o
```

- Il faut savoir quels fichiers re-compiler.

Avantages :

- Pas besoin de tout recompiler si seul un fichier est modifié :

- Si modification de `main.c` ⇒

```
gcc -c main.c
```

```
gcc -o main main.o point.o cercle.o
```

- Si modification de `point.c` ⇒

```
gcc -c point.c
```

```
gcc -o main main.o point.o cercle.o
```

Utilisation de makefile

Pour simplifier et automatiser la compilation, on écrit un fichier `makefile` :

```
main : main.o point.o cercle.o rectangle.o
    gcc -o main main.o point.o cercle.o rectangle.o -lm

main.o : main.c rectangle.h cercle.h point.h
    gcc -c main.c

point.o : point.c point.h
    gcc -c point.c

cercle.o : cercle.c cercle.h point.h
    gcc -c cercle.c

rectangle.o : rectangle.c rectangle.h cercle.h point.h
    gcc -c rectangle.c
```

Le fichier makefile

Un makefile contient des r  gles compos  es d'un nom de cible, d'une liste de d  pendances, puis des commandes shell, le tout r  dig   avec la syntaxe suivante :

```
<cible> : <liste des d  pendances>  
    <commande 1>  
    [<commande 2> ...]
```

- Une cible est un nom de fichier ou bien "all" ou bien n'importe quoi
- Chaque d  pendance est un nom de fichier et/ou une cible.
- En g  n  ral, on trouve une cible "clean" avec des commandes pour effacer les fichiers objet et ex  cutables
 - permet de forcer la recompilation compl  te

Attention : chaque commande doit   tre pr  c  d  e d'une tabulation.

Le fichier makefile

- On l'exécute avec `make <cible>` ou `make` (première cible par défaut)
- Les commandes (shell) sont exécutées (dans l'ordre) seulement si
 - la cible n'est pas un nom de fichier existant, ou
 - une dépendance a été modifiée plus récemment que la cible.
- Avant d'exécuter les commandes, chaque dépendance est explorée récursivement en tant que cible.
- Si ses dépendances n'ont pas été modifiées, les commandes d'une cible ne sont pas exécutées

Table des matières

- 1 Organisation du code
- 2 Un exemple simple
- 3 Les mécanismes de la compilation**
- 4 GCC et makefiles évolués
- 5 Bibliothèques

Les 4 étapes de compilation

	fichiers	étape	utilitaire	option gcc
texte	source .c .h			
	pré-traité .i	pré-processeur	cpp	-E
	assembleur .s	compilateur	pcc	-S
binaire	objet .o .a .so	assembleur	as	-c
	exécutable	éditeur de lien	ld	

Étape 1 : pré-processeur

- Traitement des commandes pré-processeur (commençant par `#`) :
 - idinsertion des fichiers d'entête ;
 - remplacement des macros par leur définition ;
 - “nettoyage” du code : suppression des commentaires, etc.
- Transformation purement textuelle : génère du code C pré-traité.
- Exemple (redirection de la sortie standard vers `main.i`) :
`gcc -E main.c > main.i`

Étape 1 : pré-processeur

Commandes pré-processeur (commençant par #) :

→ Inclusion de fichiers d'entête

- `#include <file.h>` → pour les fichiers d'entête système : cherche dans les répertoires systèmes standards ou spécifiés par "-I"
- `#include "file.h"` → pour les fichiers perso : cherche dans le répertoire courant, puis dans les répertoires systèmes standards ou spécifiés par "-I"

Un fichier d'entête peut en inclure d'autres

- Risque d'inclusions multiples et de redéfinitions d'objets
- Solution : utilisation de `#ifndef` :

`vecteur.h`

```
#ifndef __VECTEUR_H__  
#define __VECTEUR_H__  
...<code>...  
#endif
```

Étape 1 : pré-processeur

Commandes pré-processeur (commençant par #) :

→ Définitions de macros

- Macro simple :

```
#define <nom_macro> <code>
```

- ex : `#define N 25`
- par défaut, `<code>` vaut 1, donc `#define N 1` \Leftrightarrow `#define N`

- Macro à arguments :

```
#define <nom_macro>(arg1, ..., argn) <code>
```

- ex : `#define MIN(X,Y) ((X)<(Y)?(X):(Y))`
- Attention : limiter l'utilisation des macros aux très petites opérations

Étape 1 : pré-processeur

- Macros prédéfinies :

- standard

- `__FILE__` : nom du fichier courant sous la forme d'une chaîne de caractères

- `__LINE__` : num. de ligne courante (entier constant) → permet l'affichage de messages d'erreur

- `__INCLUDE_LEVEL__` : profondeur d'inclusion du fichier d'entête courant

- `__DATE__` : date de pré-compilation du type "Feb 4 2003"

- `__TIME__` : "18:30:12"

- `__VERSION__` : numéro de version de gcc

- spécifiques au système

- infos système

- infos CPU

- ...

Les symboles définis par macros sont remplacés dans le code lors de l'étape de traitement des commandes pré-processeur

Étape 1 : pré-processeur

Commandes pré-processeur (commençant par #) :

→Conditionnelles

```
#if <expression>
    <code>
#endif

    #if <expression>
        <code1>
    #else
        <code2>
    #endif
```

Exemple

```
#define LEVEL 3
#if LEVEL < 2
...
#endif
```

- l'expression permet d'utiliser des entiers, constantes, caractères, opérateurs arithmétiques et logiques, des appels à des macros
- permet d'ignorer une partie du code lors de la compilation
- exemples d'utilisation :
 - code dépendant d'une machine ou d'un système (libs inexistantes, etc.)
 - production d'exécutables différents d'un même source
 - un code toujours ignoré qui sert de commentaire pour plus tard
 - débogage...

Étape 1 : pré-processeur

démon

main.c

```
#include <stdio.h>
#define N 5

int main()
{
    printf("Val N: %d\n", N); // affichage de N
    printf("Ligne du programme: %d\n", __LINE__);
}
```

main.i

```
# 1 "main.c"
# 1 "/usr/include/stdio.h"
. . .
# 2 "main.c"
int main()
{
    printf("Val N: %d\n", 5);
    printf("Ligne du programme: %d\n", 7);
}
```

Étape 2 : compilateur

- Traduction du code C en code assembleur.
- Le langage assembleur dépend de la machine.
- Exemple (génération de `main.s`) :

```
gcc -S main.c
```

Étape 2 : compilateur

démon

main.s (code assembleur x86)

```
LC0:
    .ascii "Val N: %d\12\0"

LC1:
    .ascii "Ligne du programme: %d\12\0"
    .text
.globl _main
    .def      _main;   .scl      2;      .type    32;      .endef
_main:
    pushl     %ebp
    movl      %esp, %ebp
    subl      $24, %esp
    andl      $-16, %esp
    movl      $0, %eax
    addl      $15, %eax
    ...
    call      _printf
```

Étape 3 : assembleur

- Transformation du code assembleur en code binaire.
- Le fichier objet généré contient une version numérique des données et des instructions du fichier assembleur.
- Exemple (génération de `main.o`) :

```
gcc -c main.c
```

Étape 3 : assembleur

main.o (octal dump avec la commande `od -x main.o`)

```
. . .  
0000400 0000 0000 0021 0000 000d 0000 0014 0026  
0000420 0000 000b 0000 0014 0035 0000 0009 0000  
0000440 0006 003a 0000 000e 0000 0014 662e 6c69  
. . .
```

Table des symboles de main.o (commande
`nm main.o`)

```
00000000 d .data  
00000000 r .rdata  
00000000 t .text  
00000000 T _main  
          U _printf
```

T : symbole global défini dans le fichier objet.

t : symbole local défini dans le fichier objet.

U : symbole non défini.

Étape 4 : éditeur de lien

- Recherche dans les bibliothèques les fichiers objets contenant les symboles indéfinis.
- Recopie ou référence le code binaire correspondant.
- Regroupe tout le code binaire dans un fichier exécutable complet.

Table des matières

- 1 Organisation du code
- 2 Un exemple simple
- 3 Les mécanismes de la compilation
- 4 GCC et makefiles évolués**
- 5 Bibliothèques

Options gcc

- `-E -S -c` : arrêt à une étape particulière.
- `-o <output file>` : choix du nom de la destination.
- `-Wall` : warnings exhaustifs : analyse plus fine, et prévient de certaines constructions syntaxiquement correctes mais douteuses (variables non initialisées ou non utilisées, affectation utilisée comme condition, etc.).
- `-w` : aucun warning
- `-v` : verbose : affichage d'informations détaillées pendant les différentes étapes de la compilation.

Options gcc

- `-g` : ajout d'informations de débogage (étape : compilation).
- `-O -O1 -O2 -O3` : optimisation de la rapidité de l'exécution (étape : compilation).
- `-Os` : optimisation de la taille de l'exécutable (étape : compilation).
- `-pedantic -ansi -std=99` : conformité aux normes syntaxiques.
- `-I -L` gestion des répertoires.
- `-l` gestion des bibliothèques.

Make : les variables automatiques

Utilisation des variables automatiques (ou internes) prédéfinies

- `$@` : nom de la cible qui provoque l'exécution de la commande
- `$<` : nom de la 1ère dépendance
- `$?` : nom de toutes les dépendances qui sont plus récentes que la cible
- `$^` : nom de toutes les dépendances

Exemple :

```
main : main.c
    gcc -o $@ $<
```

Equivalut à :

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

Make : règles implicites

- Utilisation du "%" pour créer une règle générique

Exemple :

```
main : main.o vector.o
    gcc -o $@ $^ -lm
%.o : %.c
    gcc -c $<
```

Equivaut à :

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

Attention : on trouve parfois la notation ".c.o : " au lieu de "%.o : %.c"

Make : variables utilisateur

Définition de nouvelles variables

- Modifications ultérieures plus faciles
- Déclaration sous la forme : `NOM = VALEUR`
- Utilisation sous la forme : `$(NOM)` (on peut omettre les parenthèses si le nom de variable n'a qu'une seule lettre)

Exemple :

```
CC = gcc -g
CFLAGS = -Wall
OBJETS = main.o vector.o
EXEC = main
$(EXEC) : $(OBJETS)
    $(CC) $(CFLAGS) -o $@ $^ -lm
%.o : %.c
    $(CC) $(CFLAGS) -c $<
```

Equivaut à :

```
main : main.o vector.o
    gcc -g -Wall -o main main.o vector.o -lm
main.o : main.c
    gcc -g -Wall -c main.c
vector.o : vector.c
    gcc -g -Wall -c vector.c
```

Make : généricité

Idée

- Lister automatiquement les fichiers présents dans le répertoire
 - souplesse aux ajouts / suppressions de fichiers
 - moins de risques d'erreur de typo

Comment

- Utilisation de formes génériques : expansion avec *, ou wildcard :
 - `OBJETS = *.o` → problème car considère littéralement la chaîne "*.o"
 - `OBJETS = $(wildcard *.o)` → si un .o manque il ne sera pas listé
 - `OBJETS = $(patsubst %.c, %.o, $(wildcard *.c))` → création de la liste des .o à partir de la liste des .c
 - `OBJETS = $($ (wildcard *.c) : .c = .o)` → idem, utilisation du raccourci de patsubst

Ne pas confondre

- % dans le makefile (n'importe quel motif)
- * dans le shell (tous les motifs)

Make : généricité

Exemple de makefile générique :

(fonctionne sans modification si un fichier .c est ajouté)

```
CC = gcc -g
CFLAGS = -W -Wall
SOURCES = $(wildcard *.c)
OBJETS = $(SOURCES:.c=.o)
EXEC = main
$(EXEC) : $(OBJETS)
    @echo "\n==== Linking ===="
    $(CC) $(CFLAGS) -o $@ $^ -lm
%.o : %.c
    @echo "\n---- Rule " $@ "----"
    $(CC) $(CFLAGS) -c $<
```

NB :

- Les variables sont expansées récursivement

Règles évoluées

Règles à cibles multiples

- Plusieurs cibles ont les mêmes dépendances et les mêmes commandes

Exemple :

```
all: file1.dat file2.dat
file1.dat file2.dat : data.txt
    echo $@ $? > $@
```

Equivalait à :

```
all: file1.dat file2.dat
file1.dat : data.txt
    echo $@ $? > $@
file2.dat : data.txt
    echo $@ $? > $@
```

La règle "%.o : %.c" est un cas particulier de règle à cibles multiples

Exemple :

```
main : main.o point.o cercle.o
    gcc -o $@ $^ -lm
%.o : %.c
    gcc -c $<
```

Equivalait à :

```
main : main.o point.o cercle.o
    gcc -o $@ $^ -lm
main.o : main.c
    gcc -c $<
vector.o : vector.c
    gcc -c $<
```

Répertoires séparés

Exemple classique de répartition : headers (`include/`), sources (`src/`), objets (`obj/`), exécutables (`bin/`), bibliothèques (`lib/`).

```
<cible> : <liste des dépendances>  
      gcc <options> <fichiers>
```

Il faut, pour compiler dans les bons répertoires :

- Trouver les cibles et les dépendances (make).
- Trouver les fichiers pour les commandes (gcc).
- Placer les fichiers générés.

Solution naïve, avec `#include "include/fichier.h"` :

```
obj/main.o : src/main.c include/fichier.h  
      gcc -o  obj/main.o -c src/main.c
```


Trouver les cibles et les dépendances

Une solution simple est la commande

```
vpath <liste reps>
```

```
vpath <filtre> <liste reps>
```

Remarque :

quand `vpath` est utilisé, les variables automatiques contiennent le chemin d'accès trouvé.

Trouver les fichiers pour les commandes

Pour trouver :

- Les fichiers à inclure sans path dans les `#include` : flag `-I`

```
gcc -c main.c -I include
```

- Les sources et les objets : pas de solution simple.

Les variables du makefile et les règles évoluées simplifient un peu les commandes.

- Les bibliothèques : flag `-L`

```
gcc -o main main.o -lmalib -L libs/
```

Placer les fichiers générés

Solution avec un `-o` :

```
O_REP = obj/  
$(O_REP)main.o : main.c fichier.h  
    gcc -o $(O_REP)main.o -c main.c
```

Solution avec un `mv` :

```
O_REP = obj/  
$(O_REP)main.o : main.c fichier.h  
    gcc -c main.c  
    mv main.o $(O_REP)
```

Table des matières

- 1 Organisation du code
- 2 Un exemple simple
- 3 Les mécanismes de la compilation
- 4 GCC et makefiles évolués
- 5 Bibliothèques**

Les bibliothèques

- Une bibliothèque est un assemblage (archive) de fichiers objets contenant des fonctions déjà compilées.
- Elle est créée (compilée) à partir des fichiers objets, séparément du programme principal.
- Elle peut être utilisée par plusieurs programmes sans re-compilation.
- Elle est “liée” au programme à l’édition de lien.
- Il en existe deux sortes : statique ou dynamique (ou partagée = shared).

Les bibliothèques statiques

Particularité Le code compilé (des fonctions de la bib) est inclus dans l'exécutable.

Avantages L'exécutable contient tout ce qui est nécessaire pour fonctionner.

Inconvénients

- Exécutable volumineux
- Si une bib est modifiée, le programme doit être re-compilé.
- La bib est chargée en mémoire autant de fois que l'exécutable.

Noms `lib<nom>.a` sous Unix/Linux,
(`<nom>.lib` sous Windows)

Exemple : `libc.a` (bib standard), `libm.a` (bib mathématique), etc.

Les bibliothèques dynamiques

Particularité Seul l'emplacement mémoire de la bib est inclus dans l'exécutable.

- Avantages**
- Exécutable moins volumineux.
 - La bib est chargée une seule fois en mémoire.
 - Si une bib est modifiée, le programme n'a pas besoin d'être re-compilé.

- Inconvénients**
- L'exécutable ne fonctionne plus si la bib disparaît ou est déplacée.
 - Les modifications de la bib sont limitées (interface compatible).

Noms `lib<nom>.so` ou
`lib<nom>.so.x.y` (`x.y` = n° de version) sous Unix/Linux,
(`<nom>.dll` sous Windows)

Création

Bibliothèque statique

- Assemblage des `.o` (création de l'archive) :

```
ar -crv libmylib.a <liste des .o>
```

Options : `-c` crée la bib si elle n'existe pas ; `-r` ajoute ou remplace les fichiers objet.

- Indexation des fonctions :

```
ranlib libmylib.a
```

Bibliothèque dynamique

- Compilation avec l'option `-fPIC` (adressage relatif) :

```
gcc -c -fPIC <fichier.c>
```

- Assemblage des `.o` :

```
gcc -shared -o libmylib.so <liste des .o>
```


- Édition de lien : `gcc -o <exec> <liste des .o> -lmylib`
Recherche `libmylib.a`, `libmylib.so`, ou `libmylib.so.x.y`.
- Emplacements par défaut (bibs dynamiques) :
 - `LD_LIBRARY_PATH` est une variable d'environnement qui liste des répertoires.
 - `ldconfig` est un programme de configuration qui prend en compte les répertoires listés dans `/etc/ld.so.conf`
- Emplacement spécifique : option `-L <rep>` à l'édition de liens.
- Liste des symboles référencés par une bib : commande `nm`
- Liste des bibs dont dépend un programme : commande `ldd`