

Plan

Modules

Notion de modularité

Découpage d'un projet

Fichiers sources / d'en-tête

Création de modules

Graphes d'inclusions

Erreurs courantes et bonnes habitudes

Erreur : le fichier d'en-tête général

Une **erreur** consiste, pour un projet donné, à développer un fichier `Types.h` et plusieurs fichiers source `F1.c`, `F2.c`, ..., `Fn.c`.

Erreur : le fichier d'en-tête général

Une **erreur** consiste, pour un projet donné, à développer un fichier `Types.h` et plusieurs fichiers source `F1.c`, `F2.c`, ..., `Fn.c`.

Ici le fichier d'en-tête `Types.h` contient les déclarations de tous les types et fonctions nécessaires au projet et les fichiers sources `Fi.c` implantent chacun un sous-ensemble des fonctions déclarées.

Erreur : le fichier d'en-tête général

Une **erreur** consiste, pour un projet donné, à développer un fichier `Types.h` et plusieurs fichiers source `F1.c`, `F2.c`, ..., `Fn.c`.

Ici le fichier d'en-tête `Types.h` contient les déclarations de tous les types et fonctions nécessaires au projet et les fichiers sources `Fi.c` implémentent chacun un sous-ensemble des fonctions déclarées.

Cette conception est erronée puisque :

1. il n'y a plus de notion de module (et donc tous leurs avantages relatifs sont absents);
2. il est impossible de réutiliser du code du projet pour un nouveau (il faudrait copier / coller les types et fonctions importantes, ce qui n'est pas abordable);
3. le fichier `Types.h` peut contenir des déclarations de types et de fonctions qui n'ont pas grand chose à voir.

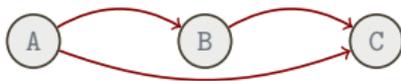
Erreur : économie d'inclusions

Une **erreur** consiste à éviter volontairement de réaliser des inclusions de modules dans d'autres si l'inclusion est déjà réalisée de manière transitive.

Erreur : économie d'inclusions

Une **erreur** consiste à éviter volontairement de réaliser des inclusions de modules dans d'autres si l'inclusion est déjà réalisée de manière transitive.

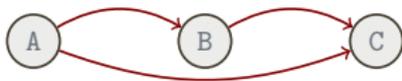
Plus explicitement, soient trois modules A, B et C tels que B inclut C, A inclut B et A inclut C :



Erreur : économie d'inclusions

Une **erreur** consiste à éviter volontairement de réaliser des inclusions de modules dans d'autres si l'inclusion est déjà réalisée de manière transitive.

Plus explicitement, soient trois modules A, B et C tels que B inclut C, A inclut B et A inclut C :

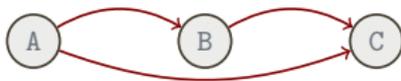


On peut être tenté de n'inclure que B dans A et C dans B car — par transitivité — ceci entraîne que C est inclus dans A. Ceci fonctionne en pratique.

Erreur : économie d'inclusions

Une **erreur** consiste à éviter volontairement de réaliser des inclusions de modules dans d'autres si l'inclusion est déjà réalisée de manière transitive.

Plus explicitement, soient trois modules A, B et C tels que B inclut C, A inclut B et A inclut C :



On peut être tenté de n'inclure que B dans A et C dans B car — par transitivité — ceci entraîne que C est inclus dans A. Ceci fonctionne en pratique.

Cette conception est cependant erronée puisque :

1. savoir de quels modules dépend A simplement en lisant son fichier d'en-tête, sans avoir de surprise sur les modules qui peuvent être inclus de manière cachée par transitivité, est un avantage ;
2. le jour où l'on modifie B de sorte qu'il n'ait plus besoin de dépendre de C provoque le fait que C n'est plus inclus dans A, ce qui est problématique.

Habitude : un module par type

Une **bonne façon de faire** par défaut consiste à créer un module pour chaque type nécessaire à l'écriture d'un projet.

Habitude : un module par type

Une **bonne façon de faire** par défaut consiste à créer un module pour chaque type nécessaire à l'écriture d'un projet.

Avec ce point de vue, il y a dans chaque `A.h` une déclaration de type unique (dont le nom est `A`, celui du module) et des déclarations de fonctions qui agissent sur des éléments de type `A`.

Habitude : un module par type

Une **bonne façon de faire** par défaut consiste à créer un module pour chaque type nécessaire à l'écriture d'un projet.

Avec ce point de vue, il y a dans chaque `A.h` une déclaration de type unique (dont le nom est `A`, celui du module) et des déclarations de fonctions qui agissent sur des éléments de type `A`.

Cette conception est correcte mais un peu limitée car

1. elle dispense d'une réflexion approfondie sur un bon découpage en modules du projet ;
2. des « types de travail » ne méritent pas d'appartenir à un module dédié ;
3. un projet compterait ainsi trop de modules.

Habitude : un module par type

Une **bonne façon de faire** par défaut consiste à créer un module pour chaque type nécessaire à l'écriture d'un projet.

Avec ce point de vue, il y a dans chaque `A.h` une déclaration de type unique (dont le nom est `A`, celui du module) et des déclarations de fonctions qui agissent sur des éléments de type `A`.

Cette conception est correcte mais un peu limitée car

1. elle dispense d'une réflexion approfondie sur un bon découpage en modules du projet ;
2. des « types de travail » ne méritent pas d'appartenir à un module dédié ;
3. un projet compterait ainsi trop de modules.

En pratique, commencer la réflexion d'un découpage en modules d'un projet en se posant la question

« *De quels types ai-je besoin ?* »

fournit un point de départ efficace, à raffiner ensuite.

Plan

Compilation

- Étapes de compilation

- Compilation séparée

- Makefile simples

- Makefile avancés

- Bibliothèques

Plan

Compilation

- Étapes de compilation

- Compilation séparée

- Makefile simples

- Makefile avancés

- Bibliothèques

Compilation d'un projet d'un fichier

La **compilation** d'un projet constitué d'un **unique fichier** `Fichier.c` contenant la fonction principale `main` se fait par la commande

```
gcc Fichier.c
```

Elle permet d'obtenir finalement un fichier **exécutable**.

Compilation d'un projet d'un fichier

La **compilation** d'un projet constitué d'un **unique fichier** `Fichier.c` contenant la fonction principale `main` se fait par la commande

```
gcc Fichier.c
```

Cette commande réalise à la suite les étapes suivantes :

1. traitement préliminaire par le **pré-processeur**;

Elle permet d'obtenir finalement un fichier **exécutable**.

Compilation d'un projet d'un fichier

La **compilation** d'un projet constitué d'un **unique fichier** `Fichier.c` contenant la fonction principale `main` se fait par la commande

```
gcc Fichier.c
```

Cette commande réalise à la suite les étapes suivantes :

1. traitement préliminaire par le **pré-processeur** ;
2. compilation en **langage assembleur** ;

Elle permet d'obtenir finalement un fichier **exécutable**.

Compilation d'un projet d'un fichier

La **compilation** d'un projet constitué d'un **unique fichier** `Fichier.c` contenant la fonction principale `main` se fait par la commande

```
gcc Fichier.c
```

Cette commande réalise à la suite les étapes suivantes :

1. traitement préliminaire par le **pré-processeur** ;
2. compilation en **langage assembleur** ;
3. traduction du langage assembleur en **langage machine** ;

Elle permet d'obtenir finalement un fichier **exécutable**.

Compilation d'un projet d'un fichier

La **compilation** d'un projet constitué d'un **unique fichier** `Fichier.c` contenant la fonction principale `main` se fait par la commande

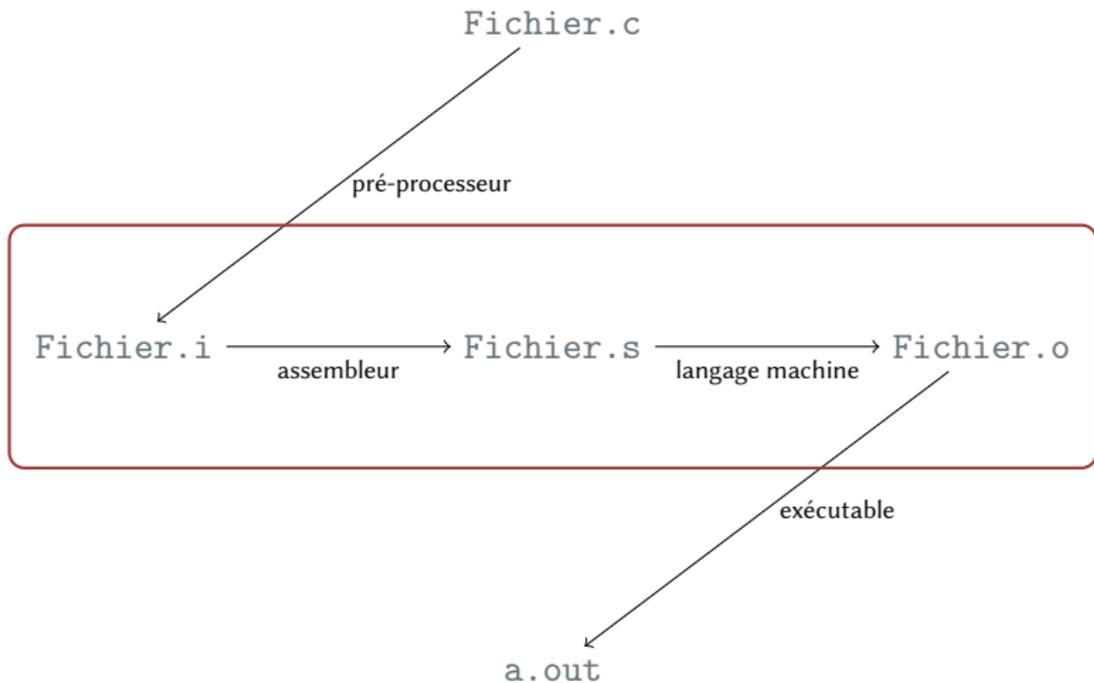
```
gcc Fichier.c
```

Cette commande réalise à la suite les étapes suivantes :

1. traitement préliminaire par le **pré-processeur**;
2. compilation en **langage assembleur**;
3. traduction du langage assembleur en **langage machine**;
4. **édition des liens**.

Elle permet d'obtenir finalement un fichier **exécutable**.

Compilation d'un projet d'un fichier



Pré-processeur

Le **pré-processeur** réalise un pré-traitement du fichier source pour le rendre traduisible en langage machine.

Pré-processeur

Le **pré-processeur** réalise un pré-traitement du fichier source pour le rendre traduisible en langage machine.

Il procède en

1. supprimant les commentaires;

Pré-processeur

Le **pré-processeur** réalise un pré-traitement du fichier source pour le rendre traduisible en langage machine.

Il procède en

1. supprimant les commentaires;
2. incluant les fichiers d'en-tête (copie / colle les fichiers `.h` inclus);

Pré-processeur

Le **pré-processeur** réalise un pré-traitement du fichier source pour le rendre traduisible en langage machine.

Il procède en

1. supprimant les commentaires;
2. incluant les fichiers d'en-tête (copie / colle les fichiers `.h` inclus);
3. traitant les définitions de symboles par un mécanisme de substitution (`#define`);

Pré-processeur

Le **pré-processeur** réalise un pré-traitement du fichier source pour le rendre traduisible en langage machine.

Il procède en

1. supprimant les commentaires;
2. incluant les fichiers d'en-tête (copie / colle les fichiers `.h` inclus);
3. traitant les définitions de symboles par un mécanisme de substitution (`#define`);
4. traitant les macro-instructions de contrôle de compilation (`#ifndef`, `#endif`, *etc.*).

Pré-processeur

Le **pré-processeur** réalise un pré-traitement du fichier source pour le rendre traduisible en langage machine.

Il procède en

1. supprimant les commentaires;
2. incluant les fichiers d'en-tête (copie / colle les fichiers `.h` inclus);
3. traitant les définitions de symboles par un mécanisme de substitution (`#define`);
4. traitant les macro-instructions de contrôle de compilation (`#ifndef`, `#endif`, *etc.*).

Il est possible de récupérer le fichier d'extension `.i` ainsi obtenu par la commande

```
gcc -E Fichier.c >> Fichier.i
```

Compilation en assembleur

Après avoir été traité par le pré-processeur, le fichier `Fichier.i` est traduit en assembleur.

Compilation en assembleur

Après avoir été traité par le pré-processeur, le fichier `Fichier.i` est traduit en assembleur.

Il est possible de récupérer le fichier d'extension `.s` ainsi obtenu par la commande

```
gcc -S Fichier.c
```

Compilation en assembleur

Après avoir été traité par le pré-processeur, le fichier `Fichier.i` est traduit en assembleur.

Il est possible de récupérer le fichier d'extension `.s` ainsi obtenu par la commande

```
gcc -S Fichier.c
```

L'assembleur est un langage très proche de la machine. Il peut se traduire assez facilement en un langage directement exécutable par le processeur.

Il existe plusieurs langages d'assemblage différents : au moins un par architecture.

Compilation en assembleur

Par exemple, avec le fichier `Fichier.c` suivant :

```
/* Fichier .c */
#include <stdio.h>

int main() {
    printf("Bonjour");
    return 0;
}
```

on obtient le fichier assembleur `Fichier.s` suivant :

```
.file "Fichier.c"
.section .rodata
.LC0:
.string "Bonjour"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
    pushq %rbp

.cfi_def_cfa_offset 16
.cfi_offset 6, -16
    movq %rsp, %rbp
.cfi_def_cfa_register 6
    movl $.LC0, %edi
    movl $0, %eax
    call printf
    movl $0, %eax
    popq %rbp
.cfi_def_cfa 7, 8

    ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC:_(Ubuntu/
        Linaro_4.8.1-10
        ubuntu9)_4.8.1"
.section .note.GNU-
        stack,"",
        @progbits
```

Traduction en langage machine

Le code assembleur `Fichier.s` est traduit en langage machine.

Traduction en langage machine

Le code assembleur `Fichier.s` est traduit en langage machine.

On obtient ce fichier d'extension `.o` par la commande

```
gcc -c Fichier.c
```

Traduction en langage machine

Le code assembleur `Fichier.s` est traduit en **langage machine**.

On obtient ce fichier d'extension `.o` par la commande

```
gcc -c Fichier.c
```

Ce fichier s'appelle **fichier objet**. Il est illisible pour un humain mais peut cependant être affiché au moyen de la commande

```
od -x Fichier.o ou bien od -a Fichier.o
```

Traduction en langage machine

Le code assembleur `Fichier.s` est traduit en **langage machine**.

On obtient ce fichier d'extension `.o` par la commande

```
gcc -c Fichier.c
```

Ce fichier s'appelle **fichier objet**. Il est illisible pour un humain mais peut cependant être affiché au moyen de la commande

```
od -x Fichier.o ou bien od -a Fichier.o
```

Le langage machine est directement compris par le processeur qui peut de ce fait exécuter directement les instructions qu'il contient.

Traduction en langage machine

Par exemple, avec le programme précédent, le contenu de `Fichier.o` est

```
0000000 del  E   L   F  stx soh soh nul nul nul nul nul nul nul nul
0000020 soh nul >  nul soh nul nul nul nul nul nul nul nul nul nul
0000040 nul nul nul nul nul nul nul nul 0  soh nul nul nul nul nul nul
0000060 nul nul nul nul @  nul nul nul nul nul @  nul cr nul nl nul
0000100 U   H  ht  e   ?  nul nul nul nul 8  nul nul nul nul  h  nul
0000120 nul nul nul 8  nul nul nul nul ]  C  B  o  n  j  o  u
0000140 r  nul nul G  C  C  :  sp (  U  b  u  n  t  u  /
0000160 L  i  n  a  r  o  sp 4  .  8  .  1  -  1  0  u
0000200 b  u  n  t  u  9  )  sp 4  .  8  .  1  nul nul nul
0000220 dc4 nul nul nul nul nul nul nul soh z  R  nul soh x dle soh
0000240 esc ff bel bs dle soh nul nul fs nul nul nul fs nul nul nul
0000260 nul nul nul nul sub nul nul nul nul A  so dle ack stx C  cr
0000300 ack U  ff bel bs nul nul nul nul .  s  y  m  t  a  b
0000320 nul .  s  t  r  t  a  b  nul .  s  h  s  t  r  t
0000340 a  b  nul .  r  e  l  a  .  t  e  x  t  nul .  d
0000360 a  t  a  nul .  b  s  s  nul .  r  o  d  a  t  a
```

Traduction en langage machine

```
0000400 nul . c o m m e n t nul . n o t e .
0000420 G N U - s t a c k nul . r e l a .
0000440 e h _ f r a m e nul nul nul nul nul nul nul nul
0000460 nul nul
*
0000560 sp nul nul nul soh nul nul nul ack nul nul nul nul nul nul nul nul
0000600 nul nul nul nul nul nul nul nul @ nul nul nul nul nul nul nul nul
0000620 sub nul nul
0000640 soh nul nul
0000660 esc nul nul nul eot nul nul
0000700 nul nul nul nul nul nul nul nul dle enq nul nul nul nul nul nul nul
0000720 0 nul nul nul nul nul nul nul vt nul nul nul soh nul nul nul
0000740 bs nul nul nul nul nul nul nul can nul nul nul nul nul nul nul nul
0000760 & nul nul nul soh nul nul nul etx nul nul nul nul nul nul nul nul
0001000 nul nul nul nul nul nul nul nul Z nul nul nul nul nul nul nul nul
0001020 nul nul
0001040 soh nul nul
```

Traduction en langage machine

```
0001060  , nul nul nul bs nul nul nul etx nul nul nul nul nul nul nul
0001100 nul nul nul nul nul nul nul nul Z nul nul nul nul nul nul nul
0001120 nul nul
0001140 soh nul nul
0001160 1 nul nul nul soh nul nul nul stx nul nul nul nul nul nul nul
0001200 nul nul nul nul nul nul nul nul Z nul nul nul nul nul nul nul
0001220 bs nul nul
0001240 soh nul nul
0001260 9 nul nul nul soh nul nul nul 0 nul nul nul nul nul nul nul
0001300 nul nul nul nul nul nul nul nul b nul nul nul nul nul nul nul
0001320  , nul nul
0001340 soh nul nul nul nul nul nul nul soh nul nul nul nul nul nul nul
0001360 B nul nul nul soh nul nul
0001400 nul nul nul nul nul nul nul nul so nul nul nul nul nul nul nul
0001420 nul nul
0001440 soh nul nul
0001460 W nul nul nul soh nul nul nul stx nul nul nul nul nul nul nul
```

Traduction en langage machine

```
0001500 nul nul nul nul nul nul nul nul dle nul nul nul nul nul nul nul
0001520 8 nul nul
0001540 bs nul nul
0001560 R nul nul nul eot nul nul
0001600 nul nul nul nul nul nul nul nul @ enq nul nul nul nul nul nul nul
0001620 can nul nul nul nul nul nul nul nul vt nul nul nul bs nul nul nul
0001640 bs nul nul nul nul nul nul nul nul can nul nul nul nul nul nul nul
0001660 dc1 nul nul nul etx nul nul
0001700 nul nul nul nul nul nul nul nul H nul nul nul nul nul nul nul nul
0001720 a nul nul
0001740 soh nul nul
0001760 soh nul nul nul stx nul nul
0002000 nul nul nul nul nul nul nul nul p eot nul nul nul nul nul nul nul
0002020 bs soh nul nul nul nul nul nul ff nul nul nul ht nul nul nul
0002040 bs nul nul nul nul nul nul nul can nul nul nul nul nul nul nul
0002060 ht nul nul nul etx nul nul
0002100 nul nul nul nul nul nul nul nul x enq nul nul nul nul nul nul nul
```

Traduction en langage machine

```
0002120 etb nul nul
0002140 soh nul nul
0002160 nul nul
0002200 nul nul nul nul nul nul nul nul nul soh nul nul nul eot nul q del
0002220 nul nul
0002240 nul nul nul nul etx nul soh nul nul nul nul nul nul nul nul nul nul
0002260 nul etx nul etx nul
0002300 nul nul
0002320 nul nul nul nul etx nul eot nul nul nul nul nul nul nul nul nul nul
0002340 nul etx nul enq nul
0002360 nul nul
0002400 nul nul nul nul etx nul bel nul nul nul nul nul nul nul nul nul nul
0002420 nul etx nul bs nul
0002440 nul nul
0002460 nul nul nul nul etx nul ack nul nul nul nul nul nul nul nul nul nul
0002500 nul nul nul nul nul nul nul nul vt nul nul nul dc2 nul soh nul
0002520 nul nul nul nul nul nul nul nul sub nul nul nul nul nul nul nul nul
```

Traduction en langage machine

```
0002540 dle nul nul nul dle nul nul
0002560 nul F i c h i e r
0002600 . c nul m a i n nul p r i n t f nul nul
0002620 enq nul nul nul nul nul nul nul nl nul nul nul enq nul nul nul
0002640 nul nul nul nul nul nul nul nul si nul nul nul nul nul nul nul
0002660 stx nul nul nul nl nul nul nul | del del del del del del del
0002700 sp nul nul nul nul nul nul nul stx nul nul nul stx nul nul nul
0002720 nul nul nul nul nul nul nul nul
0002730
```

Édition des liens

L'**édition des liens** réunit le fichier objet et le code propre aux fonctions et types de la bibliothèque standard utilisés (comme `printf`, `scanf`, *etc.*) pour produire l'exécutable complet.

Édition des liens

L'**édition des liens** réunit le fichier objet et le code propre aux fonctions et types de la librairie standard utilisés (comme `printf`, `scanf`, *etc.*) pour produire l'exécutable complet.

Avant l'édition des liens, seuls les prototypes des fonctions sont connus du compilateur. Cela lui permet de vérifier que les types sont bien respectés.

Édition des liens

L'**édition des liens** réunit le fichier objet et le code propre aux fonctions et types de la librairie standard utilisés (comme `printf`, `scanf`, *etc.*) pour produire l'exécutable complet.

Avant l'édition des liens, seuls les prototypes des fonctions sont connus du compilateur. Cela lui permet de vérifier que les types sont bien respectés.

Cependant, pour l'obtention finale de l'exécutable qui va suivre, il est nécessaire de **connaître le comportement des fonctions** (c.-à-d. leur définition).

Édition des liens

L'**édition des liens** réunit le fichier objet et le code propre aux fonctions et types de la librairie standard utilisés (comme `printf`, `scanf`, *etc.*) pour produire l'exécutable complet.

Avant l'édition des liens, seuls les prototypes des fonctions sont connus du compilateur. Cela lui permet de vérifier que les types sont bien respectés.

Cependant, pour l'obtention finale de l'exécutable qui va suivre, il est nécessaire de **connaître le comportement des fonctions** (c.-à-d. leur définition).

C'est précisément dans cette phase de la compilation que la **résolution des symboles** a lieu. C'est l'étape qui consiste à associer aux identificateurs de fonctions leur implantation.

Compilation d'un projet de plusieurs fichiers

On suppose que l'on travaille sur un projet constitué de trois modules A, B et C et d'un fichier principal `Main.c` contenant la fonction `main`.

A.h
A.c

B.h
B.c

C.h
C.c

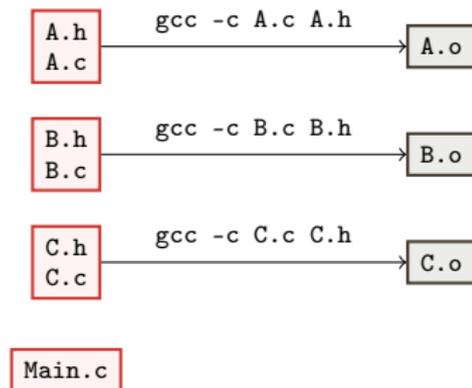
Main.c

Compilation d'un projet de plusieurs fichiers

On suppose que l'on travaille sur un projet constitué de trois modules A, B et C et d'un fichier principal `Main.c` contenant la fonction `main`.

La compilation de ce projet se réalise au moyen des étapes suivantes :

1. obtenir les fichiers objets de chaque module;

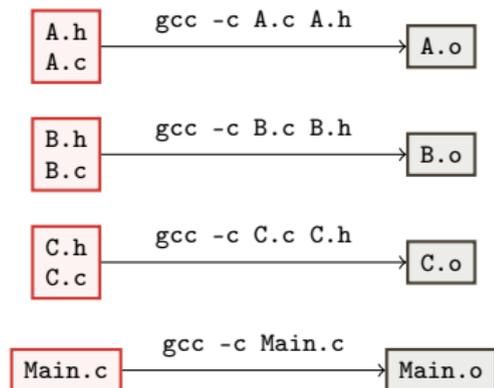


Compilation d'un projet de plusieurs fichiers

On suppose que l'on travaille sur un projet constitué de trois modules A, B et C et d'un fichier principal `Main.c` contenant la fonction `main`.

La compilation de ce projet se réalise au moyen des étapes suivantes :

1. obtenir les fichiers objets de chaque module;
2. obtenir le fichier objet de `Main.c`;

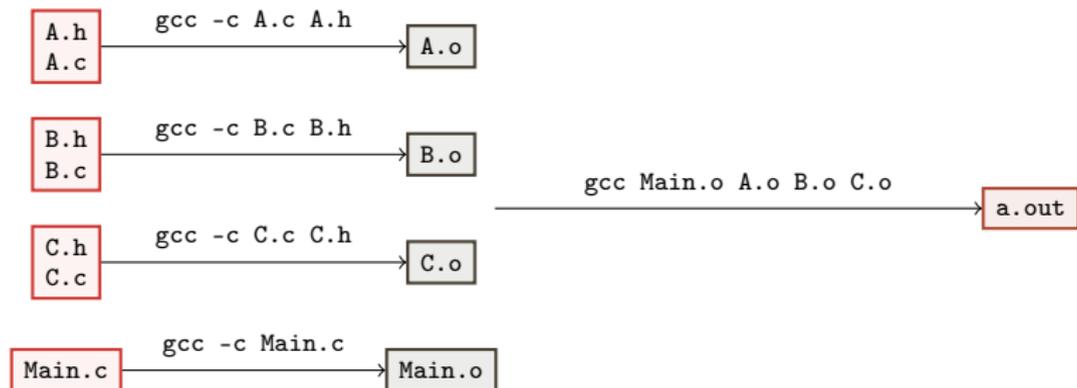


Compilation d'un projet de plusieurs fichiers

On suppose que l'on travaille sur un projet constitué de trois modules A, B et C et d'un fichier principal `Main.c` contenant la fonction `main`.

La compilation de ce projet se réalise au moyen des étapes suivantes :

1. obtenir les fichiers objets de chaque module ;
2. obtenir le fichier objet de `Main.c` ;
3. lier les fichiers objets ainsi obtenus en un exécutable.



Création des fichiers objets

Pour compiler le projet, on commence par créer un fichier objet pour chaque module M . On utilise pour cela la commande

```
gcc -c M.c M.h
```

Création des fichiers objets

Pour compiler le projet, on commence par créer un fichier objet pour chaque module M . On utilise pour cela la commande

```
gcc -c M.c M.h
```

Cette commande est équivalente à

```
gcc -c M.c
```

car le fichier source $M.c$ inclut le fichier d'en-tête $M.h$.

Création des fichiers objets

Pour compiler le projet, on commence par créer un fichier objet pour chaque module M . On utilise pour cela la commande

```
gcc -c M.c M.h
```

Cette commande est équivalente à

```
gcc -c M.c
```

car le fichier source $M.c$ inclut le fichier d'en-tête $M.h$.

On utilisera donc de préférence cette 2^e commande.

Création des fichiers objets

Pour compiler le projet, on commence par créer un fichier objet pour chaque module M . On utilise pour cela la commande

```
gcc -c M.c M.h
```

Cette commande est équivalente à

```
gcc -c M.c
```

car le fichier source $M.c$ inclut le fichier d'en-tête $M.h$.

On utilisera donc de préférence cette 2^e commande.

Chaque module est ainsi **compilé séparément** et dans un **ordre quelconque**.

Symboles non résolus

Rappel : pour compiler un module **A**, il n'est pas nécessaire que **A** ait connaissance des définitions des symboles qu'il utilise.

Seules **leurs déclarations sont suffisantes**. Celles-ci se trouvent dans les fichiers d'en-tête inclus dans **A**.

On dit qu'un **symbole n'est pas résolu** à un stade donné de la compilation si sa définition n'est pas encore connue.

```
/* Fichier .c */  
int g(int x);  
  
int f(int x) {  
    return g(x);  
}
```

Par exemple, ce fichier permet de produire un fichier objet sur la commande `gcc -c Fichier.c` même si le symbole `g` est non résolu pour le moment.

Sa déclaration (dans le fichier lui-même ou dans un fichier inclus) est cependant nécessaire (pour que le compilateur connaisse son prototype).

Résolution des symboles

Lors de l'édition des liens, un exécutable est créé. On utilise pour cela la commande

```
gcc Main.o A1.o ... An.o
```

dans le cadre d'un projet constitué des modules `A1`, ..., `An` et du module principal `Main`.

Résolution des symboles

Lors de l'édition des liens, un exécutable est créé. On utilise pour cela la commande

```
gcc Main.o A1.o ... An.o
```

dans le cadre d'un projet constitué des modules `A1`, ..., `An` et du module principal `Main`.

Cette étape **lie à chaque symbole sa définition**.

Résolution des symboles

Lors de l'édition des liens, un exécutable est créé. On utilise pour cela la commande

```
gcc Main.o A1.o ... An.o
```

dans le cadre d'un projet constitué des modules `A1`, ..., `An` et du module principal `Main`.

Cette étape **lie à chaque symbole sa définition**.

Tous les symboles utilisés dans le projet doivent être résolus (sinon, un message d'erreur est produit et l'exécutable ne peut pas être construit).

Résolution des symboles — exemple

```
/* A.h */  
#ifndef __A__  
#define __A__  
    int f(int x);  
#endif
```

```
/* A.c */  
#include "A.h"  
int f(int x) {  
    return x * x;  
}
```

```
/* B.h */  
#ifndef __B__  
#define __B__  
    int g(int x);  
#endif
```

```
/* B.c */  
#include "B.h"  
#include "A.h"  
int g(int x) {  
    return f(x);  
}
```

```
/* Main.c */  
#include "B.h"  
int main() {  
    g(5);  
    return 0;  
}
```

Résolution des symboles — exemple

```
/* A.h */  
#ifndef __A__  
#define __A__  
    int f(int x);  
#endif
```

```
/* A.c */  
#include "A.h"  
int f(int x) {  
    return x * x;  
}
```

```
/* B.h */  
#ifndef __B__  
#define __B__  
    int g(int x);  
#endif
```

```
/* B.c */  
#include "B.h"  
#include "A.h"  
int g(int x) {  
    return f(x);  
}
```

```
/* Main.c */  
#include "B.h"  
int main() {  
    g(5);  
    return 0;  
}
```

Pour compiler ce projet, on emploie les commandes

```
gcc -c A.c
```

```
gcc -c B.c
```

```
gcc -c Main.c
```

```
gcc Main.o A.o B.o
```

Résolution des symboles — exemple

```
/* A.h */  
#ifndef __A__  
#define __A__  
    int f(int x);  
#endif
```

```
/* A.c */  
#include "A.h"  
int f(int x) {  
    return x * x;  
}
```

```
/* B.h */  
#ifndef __B__  
#define __B__  
    int g(int x);  
#endif
```

```
/* B.c */  
#include "B.h"  
#include "A.h"  
int g(int x) {  
    return f(x);  
}
```

```
/* Main.c */  
#include "B.h"  
int main() {  
    g(5);  
    return 0;  
}
```

Pour compiler ce projet, on emploie les commandes

```
gcc -c A.c
```

```
gcc -c B.c
```

```
gcc -c Main.c
```

```
gcc Main.o A.o B.o
```

L'ordre d'exécution des trois 1^{res} commandes n'a aucune incidence sur le résultat produit.

Résolution des symboles — exemple

```
/* A.h */  
#ifndef __A__  
#define __A__  
    int f(int x);  
#endif
```

```
/* A.c */  
#include "A.h"  
int f(int x) {  
    return x * x;  
}
```

```
/* B.h */  
#ifndef __B__  
#define __B__  
    int g(int x);  
#endif
```

```
/* B.c */  
#include "B.h"  
#include "A.h"  
int g(int x) {  
    return f(x);  
}
```

```
/* Main.c */  
#include "B.h"  
int main() {  
    g(5);  
    return 0;  
}
```

Lors de la création de `B.o`, le compilateur **ignore** ce que fait le symbole `f`. Il sait seulement (grâce à l'inclusion de `A` dans `B`) que `f` est un symbole de fonction paramétrée par un entier et renvoyant un entier et peut donc **vérifier la correspondance des types**.

C'est au moment de l'édition des liens que le compilateur va **chercher l'implantation** du symbole `f` pour créer l'exécutable de la bonne manière.

Observation importante : la compilation d'un projet à plusieurs fichiers ne dépend pas de la manière dont ses modules sont inclus les uns dans les autres. Le schéma de compilation est toujours le même.

Plan

Compilation

Étapes de compilation

Compilation séparée

Makefile simples

Makefile avancés

Bibliothèques

Compilation séparée — intuition

Fait 1. : pour compiler un module **A**, il n'est pas nécessaire d'avoir les fichiers objets des autres modules du projet dont **A** ne dépend pas (de manière étendue ou non).

Compilation séparée — intuition

Fait 1. : pour compiler un module **A**, il n'est pas nécessaire d'avoir les fichiers objets des autres modules du projet dont **A** ne dépend pas (de manière étendue ou non).

Conséquence : si un module **B** est modifié, il n'est nécessaire de recompiler que **B** et l'ensemble des modules qui dépendent (de manière étendue) à **B**.

Compilation séparée — intuition

Fait 1. : pour compiler un module **A**, il n'est pas nécessaire d'avoir les fichiers objets des autres modules du projet dont **A** ne dépend pas (de manière étendue ou non).

Conséquence : si un module **B** est modifié, il n'est nécessaire de recompiler que **B** et l'ensemble des modules qui dépendent (de manière étendue) à **B**.

Fait 2. : si **A** dépend de **B** et seul le fichier source de **B** a été modifié, il n'est pas nécessaire de recompiler **A**.

Compilation séparée — intuition

Fait 1. : pour compiler un module A , il n'est pas nécessaire d'avoir les fichiers objets des autres modules du projet dont A ne dépend pas (de manière étendue ou non).

Conséquence : si un module B est modifié, il n'est nécessaire de recompiler que B et l'ensemble des modules qui dépendent (de manière étendue) à B .

Fait 2. : si A dépend de B et seul le fichier source de B a été modifié, il n'est pas nécessaire de recompiler A .

Conséquence : on ne recompile A que si $B.h$ a été modifié.

Compilation séparée — intuition

Fait 1. : pour compiler un module `A`, il n'est pas nécessaire d'avoir les fichiers objets des autres modules du projet dont `A` ne dépend pas (de manière étendue ou non).

Conséquence : si un module `B` est modifié, il n'est nécessaire de recompiler que `B` et l'ensemble des modules qui dépendent (de manière étendue) à `B`.

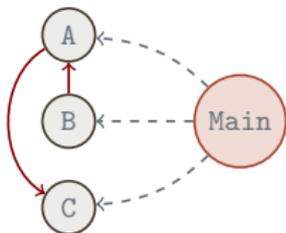
Fait 2. : si `A` dépend de `B` et seul le fichier source de `B` a été modifié, il n'est pas nécessaire de recompiler `A`.

Conséquence : on ne recompile `A` que si `B.h` a été modifié.

Attention à ne pas oublier de recompiler le module principal `Main` si celui-ci dépend de manière étendue à des modules modifiés.

Compilation séparée — exemple introductif

Considérons par exemple le projet suivant :



On le compile pour la 1^{re} fois par

```
gcc -c Main.c
```

```
gcc -c A.c
```

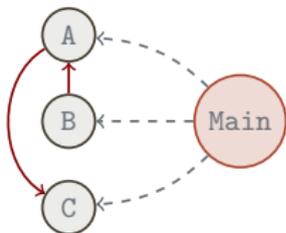
```
gcc -c B.c
```

```
gcc -c C.c
```

```
gcc Main.o A.o B.o C.o
```

Compilation séparée — exemple introductif

Considérons par exemple le projet suivant :



On le compile pour la 1^{re} fois par

```
gcc -c Main.c
```

```
gcc -c A.c
```

```
gcc -c B.c
```

```
gcc -c C.c
```

```
gcc Main.o A.o B.o C.o
```

Si on modifie par la suite le module A, il suffit d'exécuter les commandes

```
gcc -c A.c
```

```
gcc -c B.c
```

```
gcc -c Main.c
```

```
gcc Main.o A.o B.o C.o
```

pour mettre à jour l'exécutable du projet.

Note : les 3 premières lignes commutent.

Il est inutile de recompiler C car il ne dépend pas (de manière étendue) à A.

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

(1) Pour chaque module A du projet :

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

(1) Pour chaque module A du projet :

(a) compiler A si au moins l'une des conditions suivante est vérifiée :

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

(1) Pour chaque module A du projet :

(a) compiler A si au moins l'une des conditions suivante est vérifiée :

▶ A . \circ n'existe pas ;

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

(1) Pour chaque module A du projet :

(a) compiler A si au moins l'une des conditions suivante est vérifiée :

- ▶ $A.o$ n'existe pas;
- ▶ $A.c$ ou $A.h$ ont été modifiés **après** $A.o$;

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

(1) Pour chaque module A du projet :

(a) compiler A si au moins l'une des conditions suivante est vérifiée :

- ▶ $A.o$ n'existe pas;
- ▶ $A.c$ ou $A.h$ ont été modifiés **après** $A.o$;
- ▶ il existe un module B dont A dépend (de manière étendue) et tel que $B.h$ a été modifié **après** $A.o$;

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

(1) Pour chaque module A du projet :

(a) compiler A si au moins l'une des conditions suivante est vérifiée :

- ▶ $A.o$ n'existe pas;
- ▶ $A.c$ ou $A.h$ ont été modifiés **après** $A.o$;
- ▶ il existe un module B dont A dépend (de manière étendue) et tel que $B.h$ a été modifié **après** $A.o$;

(2) si au moins un module du projet a été (re)compilé, reconstruire l'exécutable.

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

- (1) Pour chaque module `A` du projet :
 - (a) compiler `A` si au moins l'une des conditions suivante est vérifiée :
 - ▶ `A.o` n'existe pas;
 - ▶ `A.c` ou `A.h` ont été modifiés **après** `A.o`;
 - ▶ il existe un module `B` dont `A` dépend (de manière étendue) et tel que `B.h` a été modifié **après** `A.o`;
- (2) si au moins un module du projet a été (re)compilé, reconstruire l'exécutable.

Nous allons utiliser l'utilitaire `make` et les fichiers `Makefile` pour rendre cette procédure automatique.

Plan

Compilation

Étapes de compilation

Compilation séparée

Makefile simples

Makefile avancés

Bibliothèques

Fichiers Makefile

L'utilitaire `make` est paramétré par un fichier dont le nom est imposé :

« `Makefile` » ou « `makefile` ».

Il doit se trouver dans le répertoire de travail (là où se trouvent les autres fichiers du projet ou au niveau des répertoires `include` et `src`).

Ce fichier fait partie intégrante du projet.

Tout fichier `Makefile` est constitué de **règles**. Elles sont de la forme

```
CIBLE: DEPENDANCES
```

```
→ COMMANDE
```

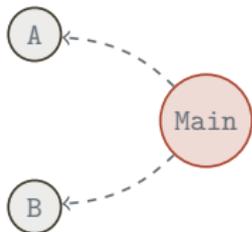
```
⋮
```

```
→ COMMANDE
```

Le symbole « `→` » désigne une tabulation.

Fichiers Makefile et fonctionnement

Considérons le projet et le Makefile suivants.



```
1 Prog: A.o B.o Main.o
2     gcc -o Prog Main.o A.o B.o
3 Main.o: Main.c A.h B.h
4     gcc -c Main.c
5 A.o: A.c A.h
6     gcc -c A.c
7 B.o: B.c B.h
8     gcc -c B.c
```

Lorsque l'on exécute la commande `make`, `make` tente de résoudre la 1^{re} règle (l. 1). Pour cela, il doit résoudre ses dépendances. Ensuite, il exécute les commandes de la règle si la cible n'est pas à jour.

Concrètement, pour pouvoir créer l'exécutable `Prog`, il est nécessaire que `A.o`, `B.o` et `Main.o` soient à jour. Une fois qu'ils le sont, il suffit de procéder à l'édition des liens (l. 2).

Fichiers Makefile et fonctionnement

Pour savoir si une cible est à jour, `make` regarde les dépendances de la règle et :

Fichiers Makefile et fonctionnement

Pour savoir si une cible est à jour, `make` regarde les dépendances de la règle et :

- ▶ si la dépendance est la cible d'une autre règle, `make` procède **récurivement** à sa résolution;

Fichiers Makefile et fonctionnement

Pour savoir si une cible est à jour, `make` regarde les dépendances de la règle et :

- ▶ si la dépendance est la cible d'une autre règle, `make` procède **récurivement** à sa résolution ;
- ▶ si la dépendance est le nom d'un fichier, `make` compare la **date de dernière modification** de la cible par rapport à celle du fichier.

Fichiers Makefile et fonctionnement

Pour savoir si une cible est à jour, `make` regarde les dépendances de la règle et :

- ▶ si la dépendance est la cible d'une autre règle, `make` procède **récurivement** à sa résolution ;
- ▶ si la dépendance est le nom d'un fichier, `make` compare la **date de dernière modification** de la cible par rapport à celle du fichier.

S'il y a au moins un fichier dans les dépendances avec une date supérieure à celle de la cible, les commandes de la règle sont exécutées.

Fichiers Makefile et fonctionnement

Pour savoir si une cible est à jour, `make` regarde les dépendances de la règle et :

- ▶ si la dépendance est la cible d'une autre règle, `make` procède **récurivement** à sa résolution ;
- ▶ si la dépendance est le nom d'un fichier, `make` compare la **date de dernière modification** de la cible par rapport à celle du fichier.

S'il y a au moins un fichier dans les dépendances avec une date supérieure à celle de la cible, les commandes de la règle sont exécutées.

On peut imposer à `make` de commencer par la résolution de la règle dont la cible est `CIBLE` par

```
make CIBLE
```

Déclarations de types et dépendances

Soient `A` et `B` deux modules tels que `B` dépend (de manière étendue) à `A`, qu'un type `T` soit déclaré dans `A` et que `B` utilise ce type.

Toute modification de `A.h` doit être suivie d'une nouvelle compilation de `B`. En effet, si la déclaration de `T` a été modifiée, `B` doit être recompilé pour la prendre en compte.

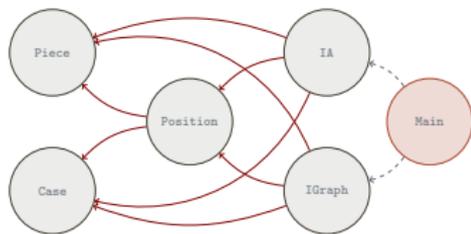
En conséquence, dans le `Makefile` du projet doit figurer la règle

```
1 B.o: B.c B.h A.h
2   gcc -c B.c
```

pour déclarer que la construction de `B.o` **dépend aussi** de `A.h`.

Attention : ceci ne s'applique pas aux modifications de l'implantation des fonctions de `A` dans `A.c` (comme nous l'avons déjà vu). La cible `B.o` ne dépend ainsi pas de `A.c`. Elle ne dépend que des **déclarations** du module `A` (et donc de `A.h`).

Exemple complet de Makefile



Le Makefile du projet dont le graphe d'inclusions (étendues) est donné ci-contre est

```
1 Echecs: Main.o Piece.o Case.o Position.o IA.o IGraph.o
2   gcc -o Echecs Main.o Piece.o Case.o Position.o IA.o IGraph.o
3 Main.o: Main.c IA.h IGraph.h
4   gcc -c Main.c
5 Piece.o: Piece.c Piece.h
6   gcc -c Piece.c
7 Case.o: Case.c Case.h
8   gcc -c Case.c
9 Position.o: Position.c Position.h Piece.h Case.h
10  gcc -c Position.c
11 IA.o: IA.c IA.h Piece.h Case.h Position.h
12  gcc -c IA.c
13 IGraph.o: IGraph.c IGraph.h Piece.h Case.h Position.h
14  gcc -c IGraph.c
```

Écriture de Makefile simples — résumé

Le Makefile d'un projet contenant des modules A_1, \dots, A_n et un module principal `Main` est génériquement de la forme

```
1 NOM: Main.o A1.o ... An.o
2     gcc -o NOM Main.o A1.o ... An.o
3
4 Main.o: Main.c EXTRAmain
5     gcc -c Main.c
6
7 A1.o: A1.c A1.h EXTRA1
8     gcc -c A1.c
9
10 ...
11
12 An.o: An.c An.h EXTRAn
13     gcc -c An.c
```

où `EXTRAmain` est la suite des noms des fichiers `.h` que `Main.c` inclut et pour tout $1 \leq k \leq n$, `EXTRA k` est la suite des noms des fichiers `.h` dont le module `A k` dépend (de manière étendue).