Création complète d'un module

Pour créer un module `A`, il faut inclure son fichier d'en-tête `A.h` dans son fichier source `A.c`.

```c
/* A.h */

...
```

```c
/* A.c */

#include "A.h"
...
```

De cette manière,

- d'une part, `A.c` a accès aux types et aux prototypes de fonctions déclarés dans `A.h` ;
- d'autre part, cela permet d'implanter les fonctions déclarées dans `A.h` sans contrainte d'ordre.

Création complète d'un module

Considérons la situation suivante :

```
/* A.h */
...
#include "C.h"
...
```

```
/* A.c */

#include "A.h"
...
```

```
/* B.h */
...
#include "C.h"
...
```

```
/* B.c */

#include "B.h"
...
```

```
/* C.h */
...
int f();
...
```

```
/* C.c */

#include "C.h"
...
```

```
/* D.c */

#include "A.h"
#include "B.h"
...
```

Le pré-processeur transforme ces fichiers en

```
/* A.h */
...
int f();
...
```

```
/* A.c */
/* Copie A.h */
...
```

```
/* B.h */
...
int f();
...
```

```
/* B.c */
/* Copie B.h */
...
```

```
/* C.h */
...
int f();
...
```

```
/* C.c */
/* Copie C.h */
...
```

```
/* D.c */

int f();
int f();
...
```

Problème : le contenu de `C.h` est copié deux fois dans `D.c`. Ceci n'est pas accepté par le compilateur car il y a **multiple déclaration** d'un même symbole (`f` ici).

Création complète d'un module

La parade consiste à **inclure** un fichier d'en-tête de **manière conditionnelle** : on procède à l'inclusion que s'il n'a pas déjà été inclus.

On utilise pour cela les macro-instructions de contrôle de compilation `#ifndef` et `#endif` ainsi que `#define`.

Le schéma général est

```
/* A.h */

#ifndef __A__
#define __A__

    /* Declaration de types */
      ...

    /* Declaration de fonctions */
      ...

#endif
```

Ainsi, lors d'une inclusion de `A.h`, le pré-processeur vérifie si la macro `__A__` n'existe pas.

- Si elle n'existe pas, alors on la définit (`#define __A__`) et le contenu du module est pris en compte ;
- sinon, cela signifie que le contenu a déjà été pris en compte. Celui-ci n'est pas repris en compte une 2^e fois.

Squelette d'un module

Pour résumer, tous les modules doivent avoir le squelette suivant :

```
/* A.h */

#ifndef __A__
#define __A__

    /* Inclusions eventuelles de modules */
    ...

    /* Definitions eventuelles de macros */
    ...

    /* Declarations eventuelles de types */
    ...

    /* Declarations eventuelles de fonctions */
    ...

#endif
```

```
/* A.c */

#include "A.h"

/* Inclusions eventuelles de modules */
...

/* Definitions eventuelles de fonctions privees */
...

/* Definitions de toutes les fonctions declarees
 * dans le fichier d'en-tete */
...
```

Exemple du module `Parseur`

– Exemple –

```c
/* Parseur.h */

#ifndef __PARSEUR__
#define __PARSEUR__

#include "Formule.h"

    /* Convertit la chaine de caracteres `ch` sensee
     * representer une formule en une variable de type
     * `Form`, qui va etre ecrite dans `f`. Renvoie `1`
     * si `ch` represente bien une formule et `0` sinon. */
    int chaine_vers_form(Form *f, char *ch);

#endif
```

```c
/* Parseur.c */

#include "Parseur.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int chaine_vers_form(Form *f, char *ch) {
    ...
    assert(f != NULL);
    ...
}
```

3. Modules

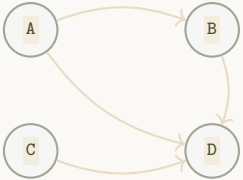
3.1 Notion de modularité
3.2 Découpage d'un projet
3.3 Fichiers sources / d'en-tête
3.4 Création de modules
3.5 Graphes d'inclusions
3.6 Erreurs courantes et bonnes habitudes

Graphes d'inclusions

On rappelle qu'un module `A` **dépend** d'un module `B` si le **fichier d'en-tête** de `A` inclut `B`.

Pour visualiser l'allure d'un projet, on trace son graphe d'inclusions. On représente pour cela chacun des modules qui le composent dans des cercles (sommets) et on trace des flèches (arcs) de `A` vers `B` pour tout module `A` dépendant de `B`.

– Exemple –



Ce graphe d'inclusions signifie que

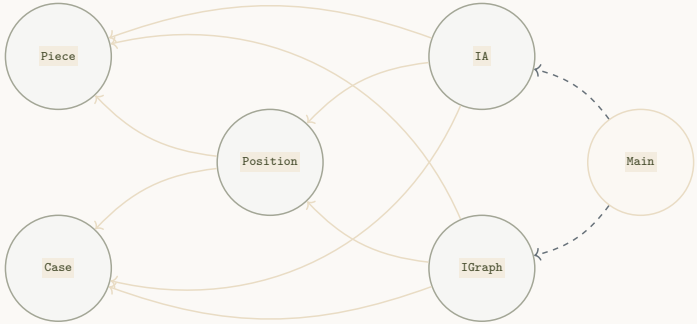
- `A.h` inclut `B.h` et `D.h` ;
- `B.h` inclut `D.h` ;
- `C.h` inclut `D.h` ;
- `D.h` n'inclut rien.

On ne mentionne pas dans les graphes d'inclusions les inclusions aux fichiers d'en-tête standards (`stdio.h`, `stdlib.h`, `assert.h`, *etc.*).

Graphe d'inclusions et fichier principal

– Exemple –

Le graphe d'inclusions d'un projet consistant à faire jouer l'ordinateur aux échecs contre un humain peut être le suivant :



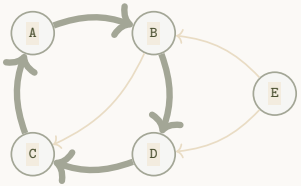
Tout projet contient un fichier source `Main.c`, **le fichier principal** du projet, où figure la fonction `main` (le point d'entrée de l'exécution du programme). Celui-ci apparaît dans le graphe d'inclusions.

Inclusions circulaires

Les graphes d'inclusions permettent d'avoir une vision globale de l'architecture d'un projet.

Ils permettent aussi de mettre en évidence des problèmes de conception et notamment les problèmes d'**inclusion circulaire**. Ce type de problème s'observe par la présence d'un **cycle** dans le graphe d'inclusions.

– Exemple –



Règle importante : il ne doit jamais y avoir de cycle dans le graphe d'inclusions d'un projet. S'il y a un cycle, c'est que le projet est **mal découpé en modules**.

Limiter les inclusions

La plupart des inclusions circulaires peuvent être évitées en **réduisant au maximum les inclusions** de modules dans les **fichiers d'en-tête** en les faisant plutôt si possible dans les fichiers sources.

– Exemple (début) –

Supposons que l'on dispose d'un module `Tri` qui permet de trier des tableaux génériques (nous aborderons plus loin ce concept de généricité). Il est de la forme :

```c
/* Tri.h */
#ifndef __TRI__
#define __TRI__

    void trier_tab(void **t, int n,
        int (*est_inf)(void *, void *));
    ...

#endif
```

```c
/* Tri.c */
#include "Tri.h"
...
void trier_tab(void **t, int n,
    int (*est_inf)(void *, void *)) {
    ...
}
...
```

Limiter les inclusions

– Exemple (suite et fin) –

On souhaite maintenant écrire un module `TabInt` pour gérer des tableaux d'entiers.

On n'écrira pas

```c
/* TabInt.h */
#ifndef __TAB_INT__
#define __TAB_INT__

#include "Tri.h"

    typedef struct {int n; int *tab;} TabInt;
    int trier_tab_int(TabInt *t);

#endif
```

```c
/* TabInt.c */
#include "TabInt.h"

int trier_tab_int(TabInt *t) {
    /* Util. de 'trier_tab' */
}
```

mais plutôt

```c
/* TabInt.h */
#ifndef __TAB_INT__
#define __TAB_INT__

    typedef struct int n; int *tab; TabInt;
    int trier_tab_int(TabInt *t);

#endif
```

```c
/* TabInt.c */
#include "TabInt.h"

#include "Tri.h"

int trier_tab_int(TabInt *t) {
    /* Util. de 'trier_tab' */
}
```

Dépendances étendues

Nous serons amenés dans la suite (dans le cadre de la compilation séparée) — étant donné un fichier `A.c` — à considérer l'ensemble des fichiers d'en-tête qu'il inclut.

Si `A.c` inclut un fichier d'en-tête `B.h`, alors le module `A` dépend de manière étendue au module `B`.

Le graphe d'inclusions étendu d'un projet consiste en le graphe d'inclusions du projet dans lequel sont ajoutées des **flèches en pointillés** pour symboliser les dépendances étendues.

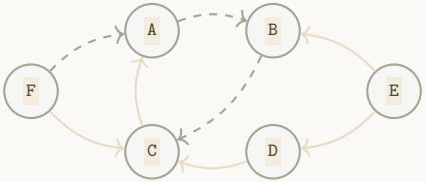
Note 1. : les flèches qui partent du module principal `Main` représentent des inclusions étendues.

Note 2. : les cycles dans lesquels intervient au moins une flèche en pointillés ne posent pas de problème de structure du projet.

Dépendances étendues

– Exemple –

Le graphe d'inclusions étendu



nous renseigne sur le fait que

- `A.h` n'inclut rien ;
- `A.c` inclut `B.h` ;
- `B.h` n'inclut rien ;
- `B.c` inclut `C.h` ;
- `C.h` inclut `A.h` ;
- `C.c` n'inclut rien ;
- `D.h` inclut `C.h` ;
- `D.c` n'inclut rien ;
- `E.h` inclut `B.h` et `D.h`
- `E.c` n'inclut rien ;
- `F.h` inclut `C.h` ;
- `F.c` inclut `A.h`.

On observe que ce projet n'est pas mal structuré car il ne possède pas de cycle formé uniquement par des flèches de dépendance.

3. Modules

3.1 Notion de modularité
3.2 Découpage d'un projet
3.3 Fichiers sources / d'en-tête
3.4 Création de modules
3.5 Graphes d'inclusions
3.6 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`.

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.

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.

– Exemple –

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 inclut 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 inclut 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.

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.

4. Compilation

Plan

4. Compilation

4.1 Étapes de compilation

4.2 Compilation séparée

4.3 `Makefile` simples

4.4 `Makefile` avancés

4.5 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

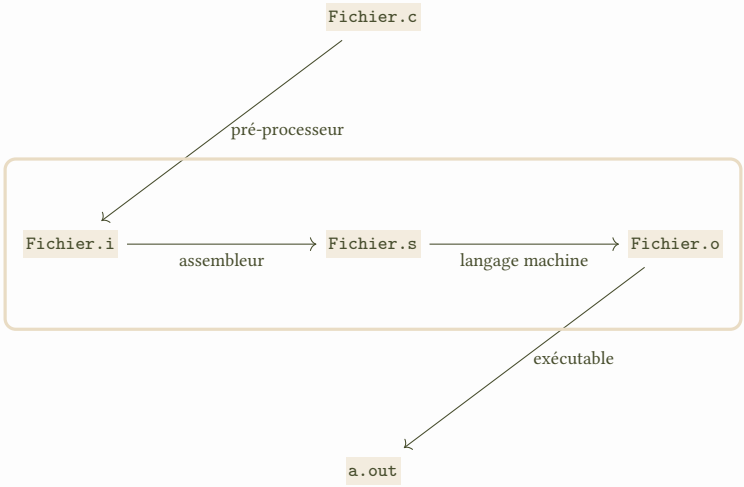
```
gcc Fichier.c
```

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.

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**.

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

– Exemple –

Avec le fichier `Fichier.c` suivant :

```c
/* 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**.

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

– Exemple (1 / 6) –

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 nul
0000020 soh nul   >  nul soh nul 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

– Exemple (2 / 6) –

```
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 nul nul nul nul nul nul nul nul nul nul nul nul nul nul
   *
0000560  sp nul nul nul soh nul nul nul ack nul nul nul nul nul nul nul
0000600 nul nul nul nul nul nul nul nul  @  nul nul nul nul nul nul nul
0000620 sub nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0000640 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0000660 esc nul nul nul eot nul nul nul nul nul nul nul nul nul nul nul
0000700 nul nul nul nul nul nul nul nul dle enq 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
0000760  &  nul nul nul soh nul nul nul etx nul nul nul nul nul nul nul
0001000 nul nul nul nul nul nul nul nul  Z  nul nul nul nul nul nul nul
0001020 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001040 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
```

Traduction en langage machine

– Exemple (3 / 6) –

```
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 nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001140 soh nul nul nul nul nul nul nul nul nul nul nul nul nul 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 nul nul nul nul nul nul nul nul nul nul nul nul nul
0001240 soh nul nul nul nul nul nul nul nul nul nul nul nul nul 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 nul nul nul nul nul nul nul nul nul nul nul 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 nul nul nul nul nul nul nul nul nul
0001400 nul nul nul nul nul nul nul nul  so  nul nul nul nul nul nul nul
0001420 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001440 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001460  W  nul nul nul soh nul nul nul stx nul nul nul nul nul nul nul
```

Traduction en langage machine

– Exemple (4 / 6) –

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

Traduction en langage machine

– Exemple (5 / 6) –

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

Traduction en langage machine

– Exemple (6 / 6) –

```
0002540 dle nul nul nul dle nul nul nul nul nul nul nul nul nul nul nul
0002560 nul nul nul nul nul nul nul nul 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.

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).

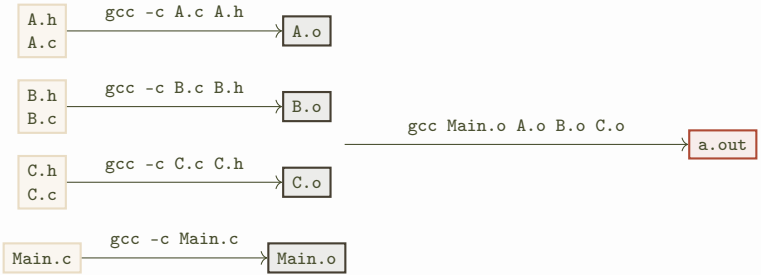
C'est précisément dans cette phase de la compilation que la **résolution des symboles** a lieu. C'est l'étape qui consiste à associer aux identificateurs de fonctions leur implantation.

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`.

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
```

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 2e 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.

– Exemple –

```c
/* Fichier.c */

int g(int x);

int f(int x) {
    return g(x);
}
```

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`.

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

– Exemple (1 / 2) –

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

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

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

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

```c
/* 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 1res commandes n'a aucune incidence sur le résultat produit.

Résolution des symboles — exemple

– Exemple (2 / 2) –

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

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

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

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

```c
/* 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

4. Compilation

4.1 Étapes de compilation

4.2 Compilation séparée

4.3 `Makefile` simples

4.4 `Makefile` avancés

4.5 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).

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`.

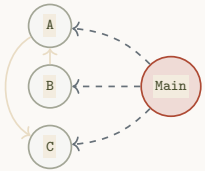
Conséquence : on ne recompile `A` que si `B.h` a été modifié.

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

– Exemple –

Considérons le projet suivant :



On le compile pour la 1re 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 pour mettre à jour l'exécutable du projet.

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

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.

(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.