

Chapitre 3

Chaîne de Compilation



2021-2022



Connaître les différents étages d'une chaîne de compilation C



Comprendre le processus de compilation (rôle des étapes)



Comprendre le processus d'édition des liens



Analyser les fichiers issus des différents étages de la toolchain

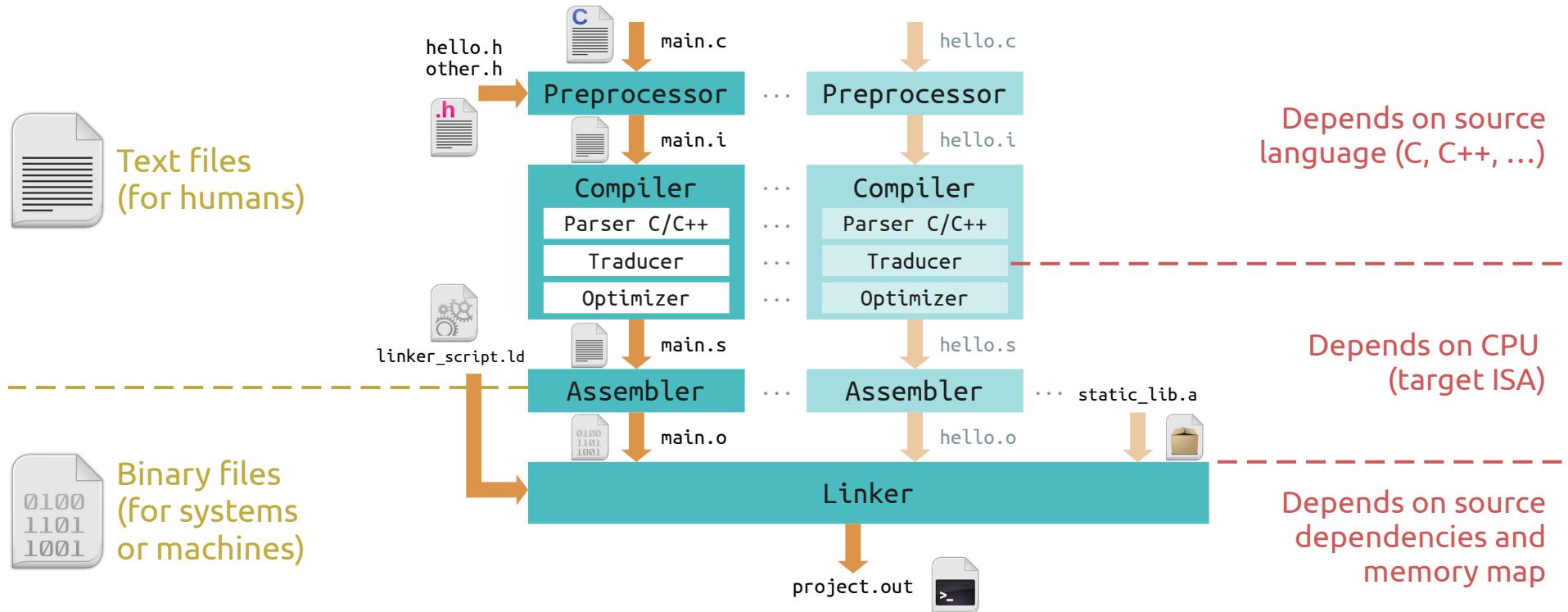
gcc, as, ld, objdump, readelf, strip, ar, ...



Savoir créer ou debugger un projet C/C++/asm en cours de construction

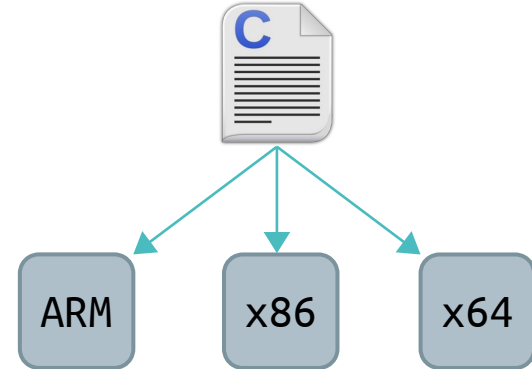
OBJECTIFS

Schéma à connaître et comprendre



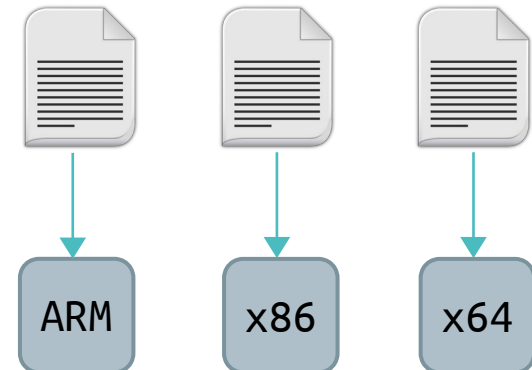
Le langage C est qualifié de **portable**.

Cela signifie qu'un programme initialement rédigé en C peut être exécuté sur n'importe quelle architecture de processeur.



Cela s'oppose aux **langages d'assemblage**, qui eux sont **spécifiques à une architecture**.

Ainsi, une nouvelle version du programme doit être écrite pour chaque architecture sur laquelle on veut exécuter le programme.



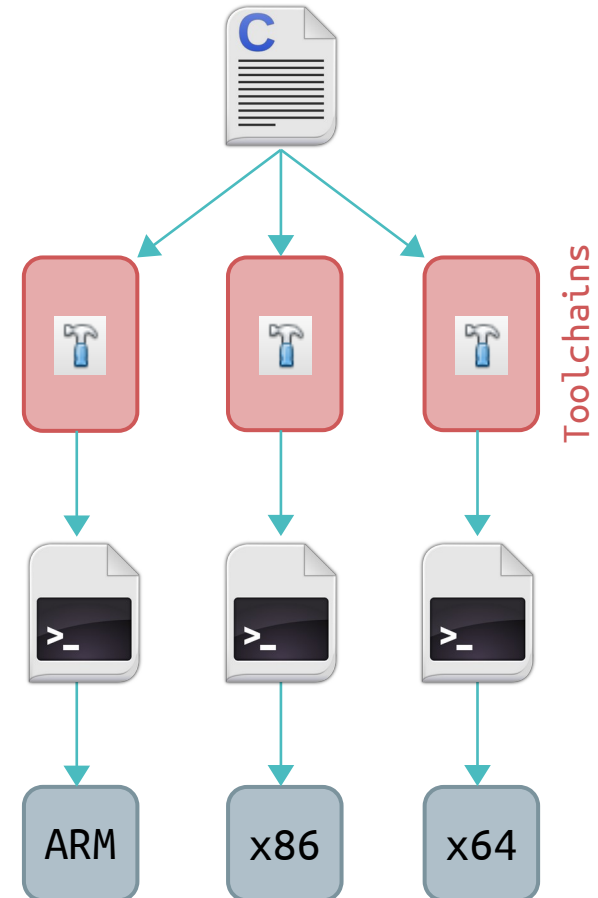
Toutefois, la diapo précédente est imprécise.

“un programme ***initialement*** rédigé en C peut être exécuté sur n'importe quelle architecture de processeur.”

En effet les fichiers C ne sont pas exécutables. Ils doivent être compilés afin de créer un exécutable.

Ce processus est réalisé par la **chaîne de compilation, ou toolchain**.

En partant d'un seul fichier C, il faut autant de *toolchains* que de processeurs sur lesquels exécuter le programme.



Ce chapitre est consacré à la chaîne de compilation.

Plusieurs *toolchains* existent. Vous avez peut-être entendu parler de GCC ou MinGW, qui sont pour les architectures x86 et x64. Vous avez aussi utilisé XC8, la *toolchain* Microchip pour MCU PIC18.

La *toolchain* GCC (GNU Compiler Collection, <https://gcc.gnu.org>) sera utilisée à titre d'exemple dans ce chapitre.

Le processus reste le même dans n'importe quelle *toolchain* C. En revanche les formats et extensions des fichiers intermédiaires ne sont pas standardisés, ils peuvent donc varier d'une *toolchain* à une autre, ou d'une plateforme matérielle à une autre.

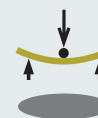


LA TOOLCHAIN GCC

Du fichier source C vers le fichier exécutable

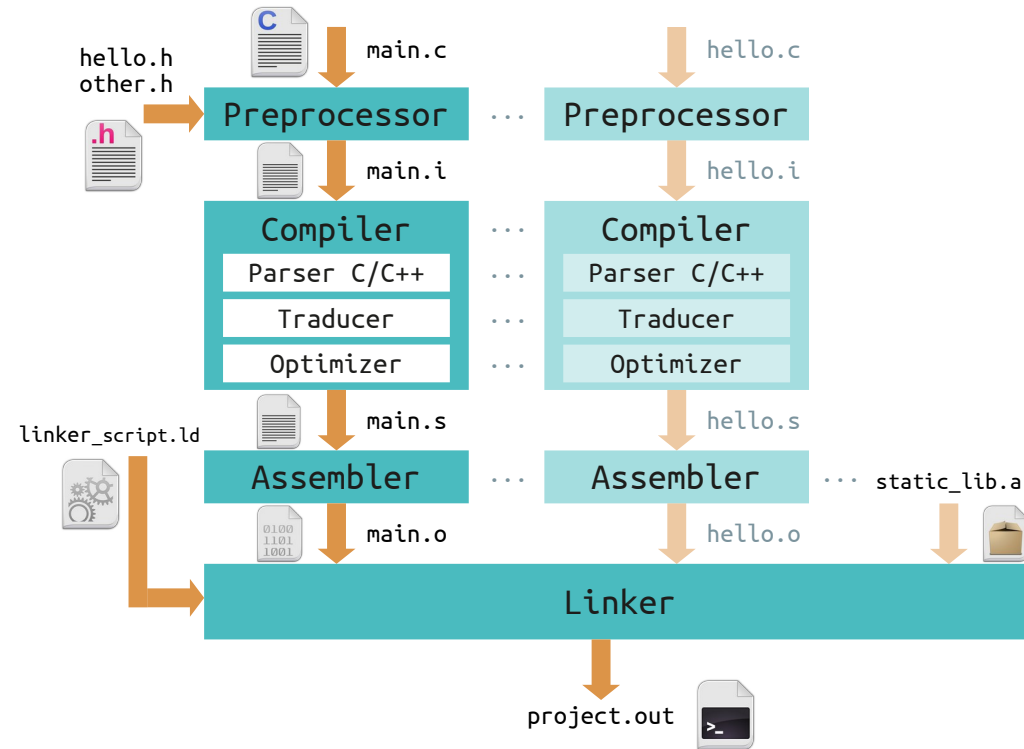


Les pages de cette partie sont à connaître par cœur.



LA TOOLCHAIN GCC

Vue d'ensemble



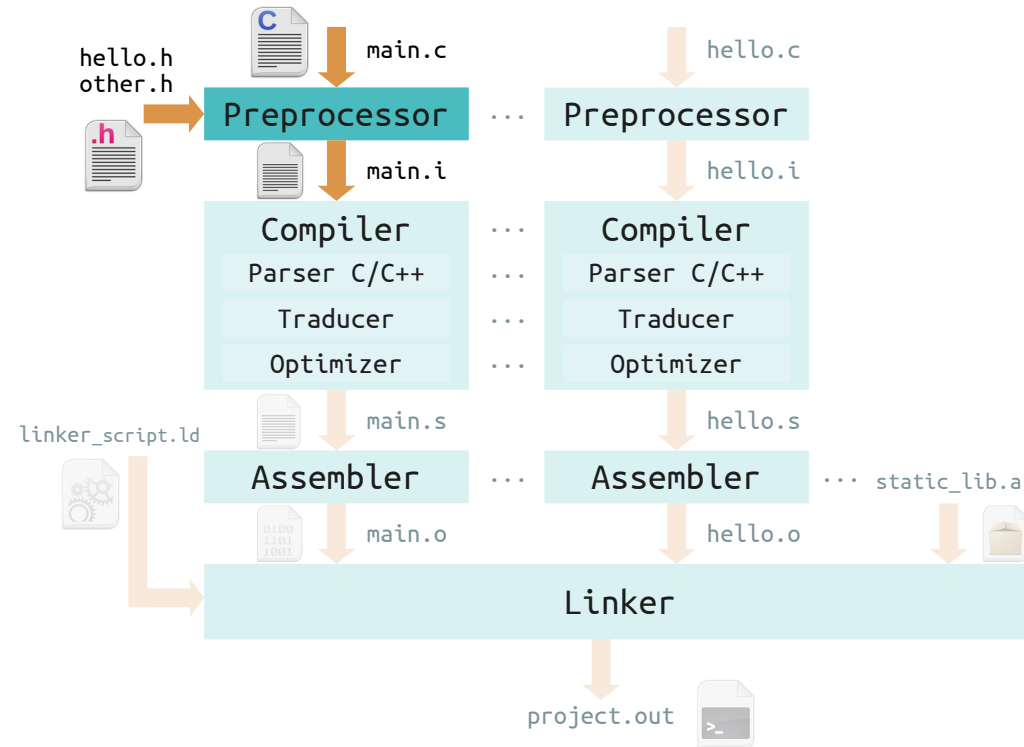
La toolchain est composée de 4 étages, pouvant produire un fichier de sortie.

Chaque fichier C passe indépendamment à travers les trois premiers étages, pour produire autant de **fichiers objets** ou **object files** (*.o).

Le dernier étage récupère tous les fichiers objets pour construire le **fichier exécutable** final.

C'est tout ce processus qui se déroule quand vous appuyez sur le bouton 'build' ou quand vous appelez la commande 'gcc'.

Étage de pré-traitement



L'étage de *preprocessing* s'occupe de:

Inclusion de code (`#include <hello.h>`)

Compilation conditionnelle

`#if CONDITION`

`#ifndef HELLO_H`

Toute autre directive pré-processeur (débuté par '#')

Opérateurs `#pragma` (depuis la norme C99)

Le fichier de sortie se nomme *preprocessed file*. C'est un fichier texte, générique, d'extension `.i`.

Étage de pré-traitement

Exemple de pré-traitement avec `gcc -E test.c > test.i`

```
#define QUARANTE      40
#define QUARANTE_UN  41
int a = QUARANTE + QUARANTE_UN;

#define CHAINE_DE_CARACTERES "du texte"
char * c = CHAINE_DE_CARACTERES;

#define DEBUG
#ifdef DEBUG
printf("Version debug \n");
#endif
#ifndef DEBUG
printf("Version non-debug \n");
#endif
```

test.c



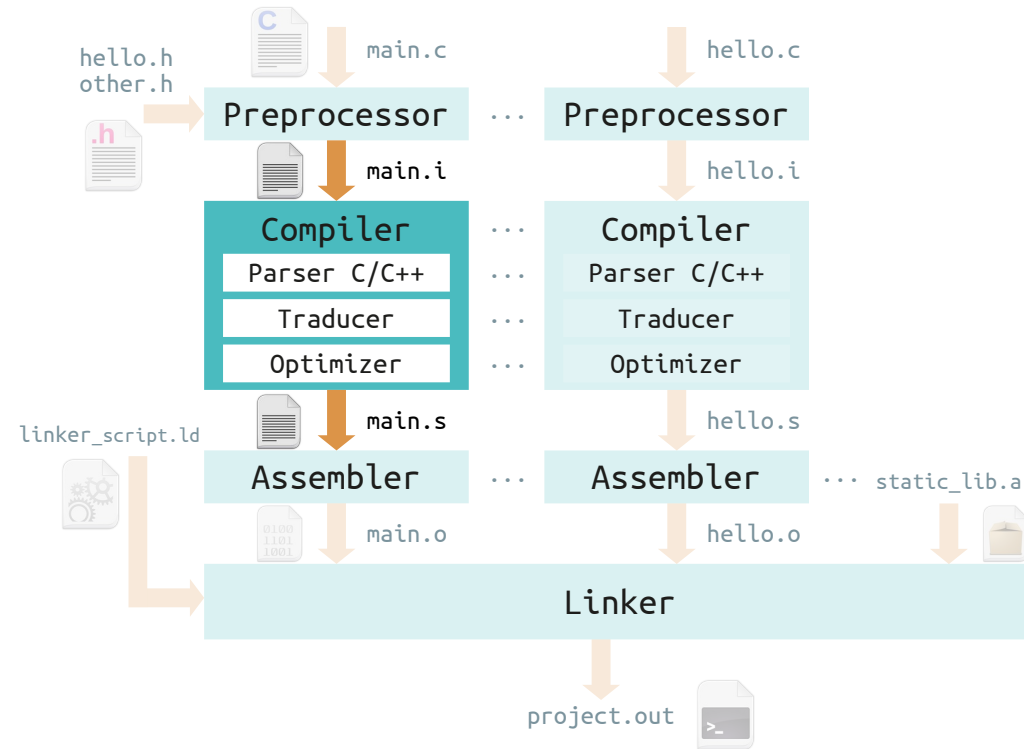
test.i

```
int a = 40 + 41;
```

```
char * c = "du texte";
```

```
printf("Version debug \n");
```

Étage de compilation



Le fichier généré passe ensuite à travers l'**étage de compilation**, qui comporte :

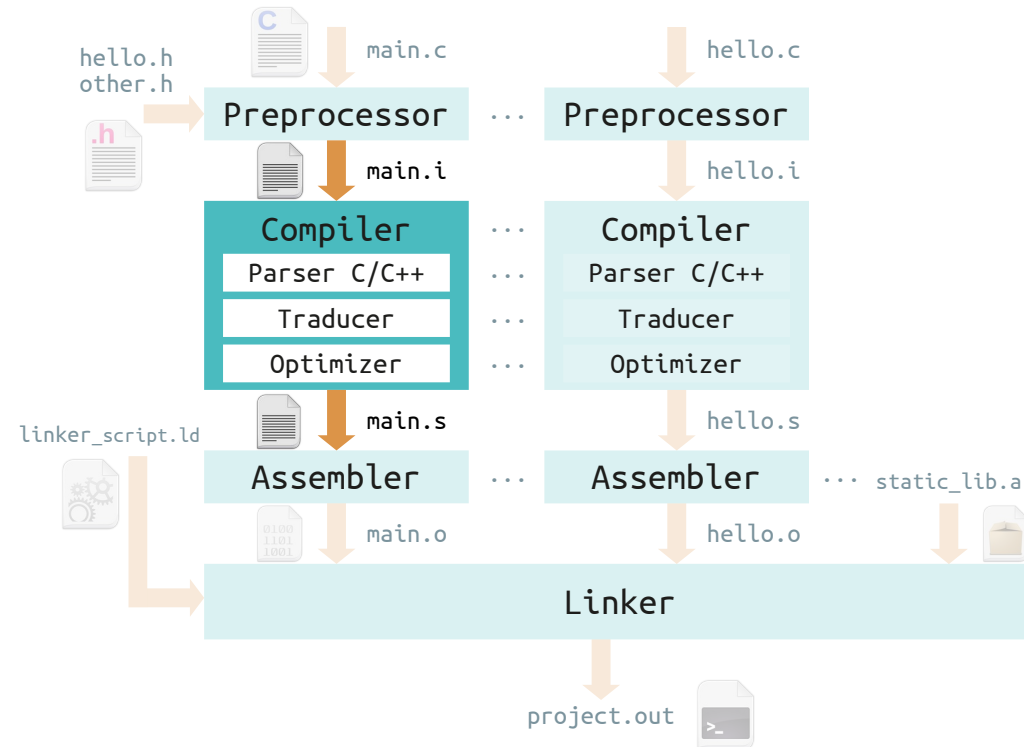
Parser : vérifie le respect des règles du langage.

Traducteur : fabrique un nouveau programme dans la langue d'assemblage du processeur cible.

Optimiseur : améliore le code grâce à sa connaissance poussée de l'architecture cible.

Le fichier de sortie est au format **langage d'assemblage** (.a, .asm). Il s'agit toujours d'un fichier texte, mais il est maintenant spécifique à l'architecture cible.

Étage de compilation



Tâche du *parser*

Analyse lexicale

Les mots-clés sont-ils corrects ?

ex : `"float"` vs `"flaot"`

Analyse syntaxique (ou *parsing*)

Les mots forment-ils des phrases correctes ?

ex : `"float tab[4];"` vs `"float[4] tab;"`

Analyse sémantique

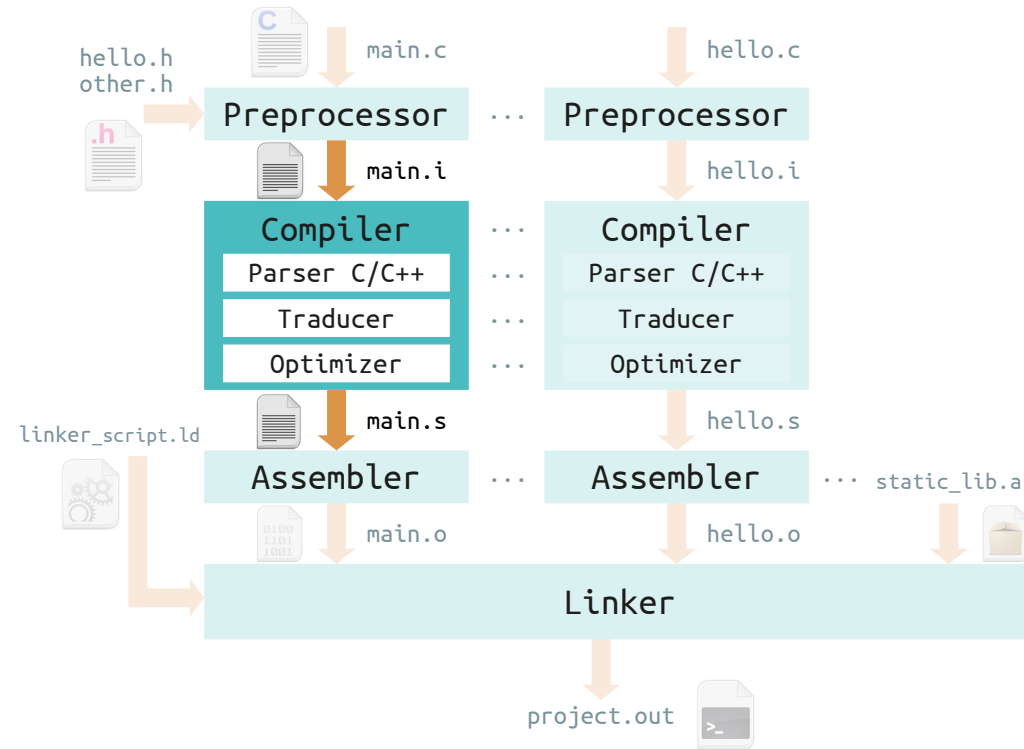
Les phrases font-elles sens ?

ex : `"a = b / 0;"`.

Cohérence entre déclaration et définition ?

L'analyseur construit la **table des symboles**.

Étage de compilation



Traducer (ou code generator)

Spécifique à l'architecture cible.

Si l'architecture cible est différente de l'architecture sur laquelle la toolchain tourne, on appelle cela de la cross-compilation.

Optimizer

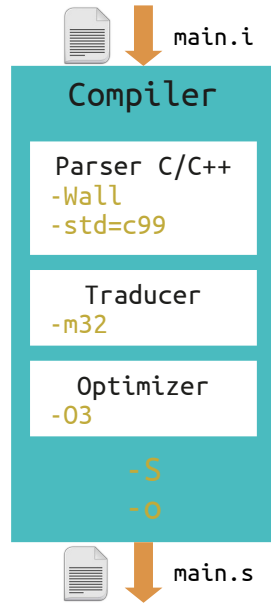
Phase optionnelle. Spécifique à la cible.

Modifie le code dans une forme plus rapide (ou plus compacte).

inline expansion, dead code elimination, constant folding, loop transformation, parallelization

Étage de compilation

Ex.: `gcc -S -Wall -std=c99 -m32 -O3 ./build/main.i -o ./build/main.o`



`./build/main.i`

Spécifie le fichier d'entrée à compiler

`-Wall`

Afficher tous les warnings (*Warning All*)

`-std=c99`

Norme du langage à respecter

`-m32`

Traduction en assembleur pour cible 32-bit

`-O3`

Niveau 3 d'optimisation

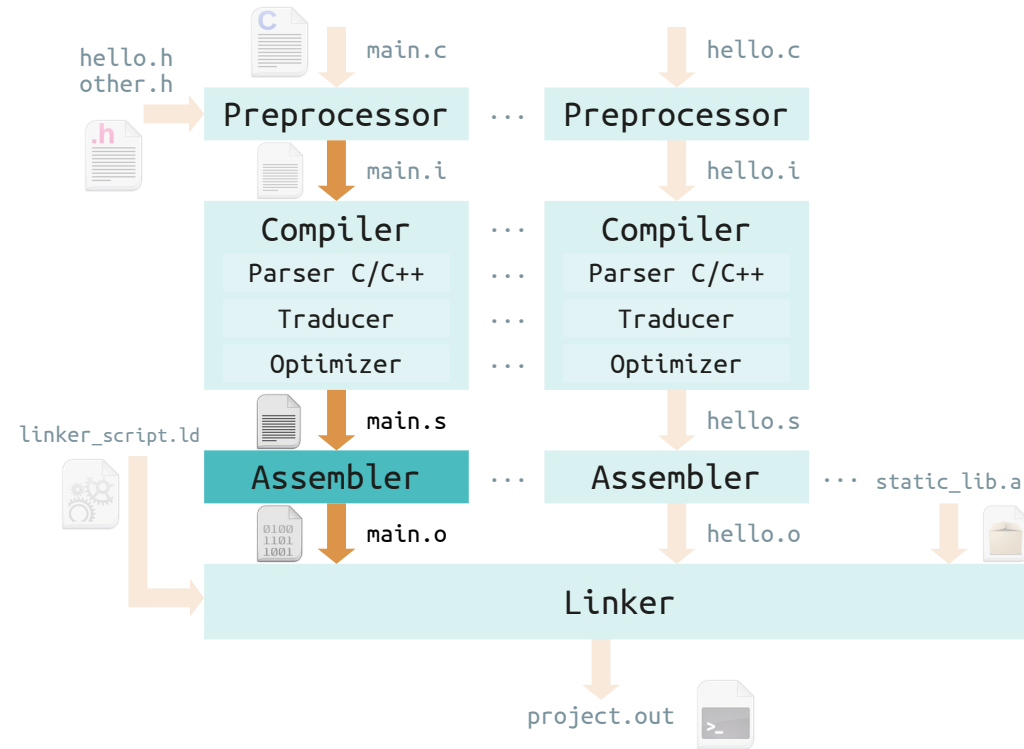
`-S`

S'arrêter à l'étage d'assemblage

`-o ./build/main.o`

Nom du fichier de sortie

Étage assembleur



L'**assembleur** traduit le fichier en langage d'assemblage vers un fichier objet binaire.

Le fichier objet est relogeable et non exécutable.

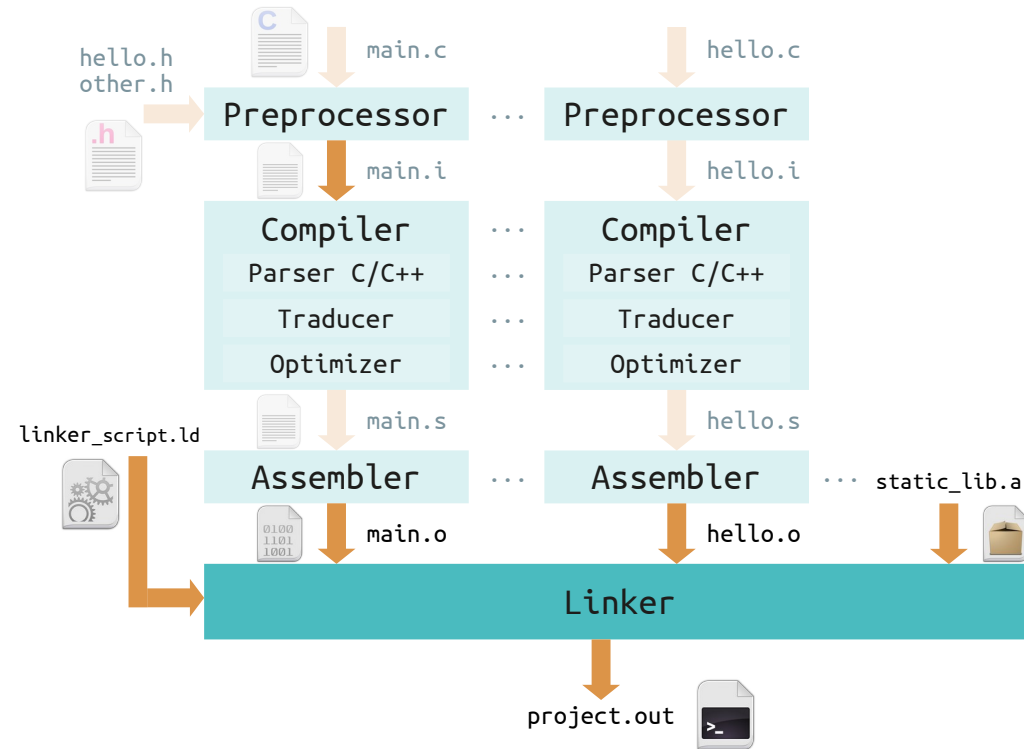
Il utilise des références symboliques : les variables et fonctions externes ont un nom, mais leur adresse est encore inconnue pour le moment.

Le fichier objet est spécifique à l'architecture cible, mais ne dépend pas de son modèle mémoire.

Le fichier de sortie est désormais binaire : il est illisible pour l'humain, compris seulement par la cible.

Note : « langage d'assemblage » != « étage assembleur »

Étage d'édition des liens



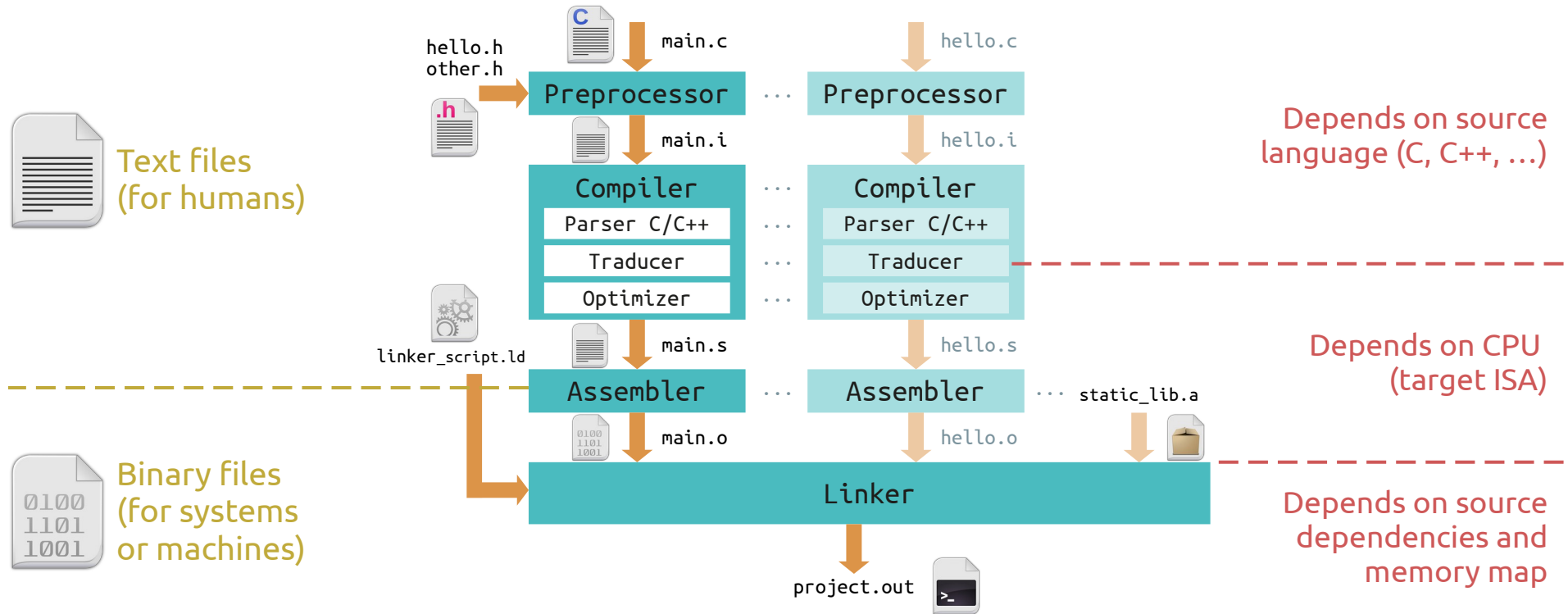
L'éditeur des liens (*linker*) fusionne tous les fichiers objet en un seul exécutable.

Plus précisément, le *linker* lie les fichiers ensemble : *object files*, *static libraries* (ensemble de fonctions pré-compilées), et *dynamic libraries* (chargées à l'exécution du programme).

C'est la résolution de la table des symboles et la résolution des liens avec le modèle mémoire de la cible.

Le fichier généré est habituellement un exécutable, mais ça peut être un fichier relogeable (*static library*).

Le fichier généré dépend de l'architecture et du modèle mémoire de l'architecture cible.



Plusieurs choix disponibles pour les utilisateurs pro-console !

Si seulement quelques sources, compiler avec la commande `gcc`.
Taper `man gcc` pour le manuel, ou <https://linux.die.net/man/1/gcc>.
Si le projet devient plus gros, n'utiliser que gcc devient vite pénible.



On automatise le processus de compilation avec la commande `make`.
Toutes les règles de compilation sont écrites dans un fichier `Makefile`.

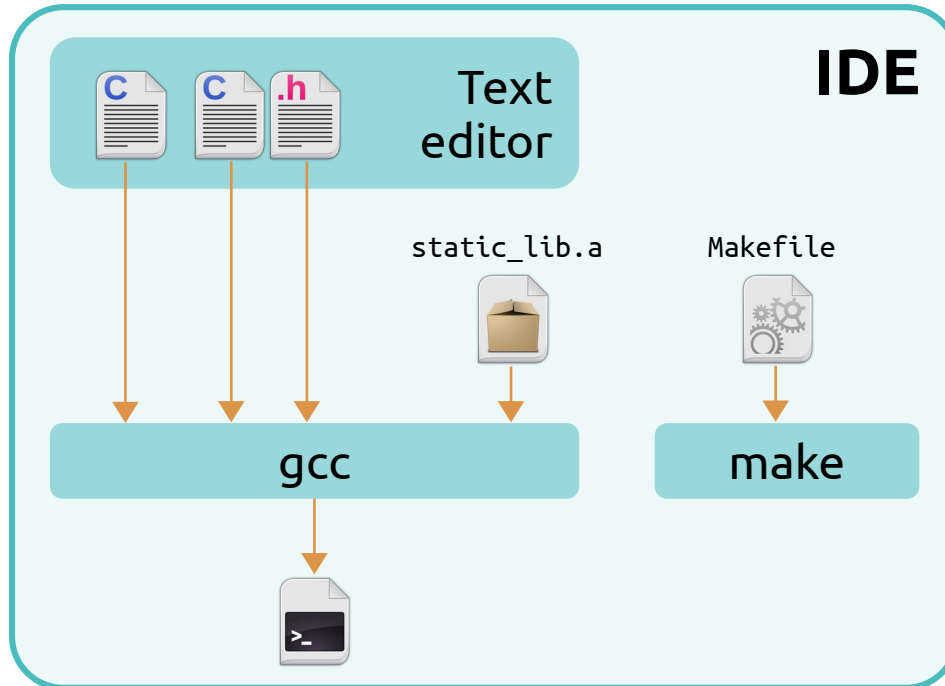


On peut même automatiser l'automatisation du processus de compilation avec `CMake` !



Mais la plupart des développeurs utilisent un IDE :

Integrated Development Environment ou Environnement de Développement Intégré.



Un IDE est pratique pour les projets de grande envergure.

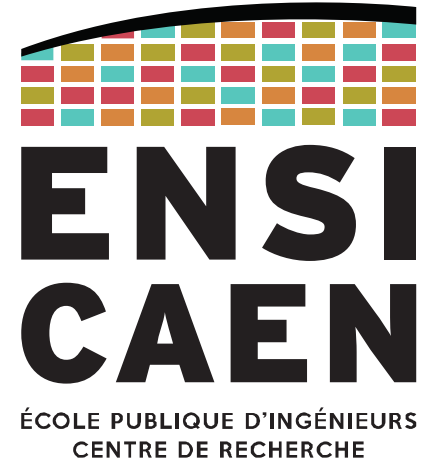
Éditeur de texte avec coloration syntaxique, auto-complétion, ...

Processus de compilation automatisé.

Plusieurs outils d'aide, le plus important étant le *debugger*.

FICHIERS BINAIRES

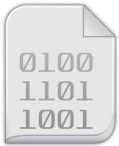
Analyse du format ELF
(*Executable and Linkable Format*)



Format ELF



Les fichiers sources (*.c, *.cpp, *.h, ...), *preprocessed* (*.i), et compilés (*.a) sont des **fichiers textes** : ils sont lisibles par l'humain mais pas par la machine.



Les fichiers objets (*.o), bibliothèques statiques (*.a), exécutable (*, *.out) sont quant à eux **binares** : impossible de les lire avec un éditeur de texte, seule la machine peut les comprendre.

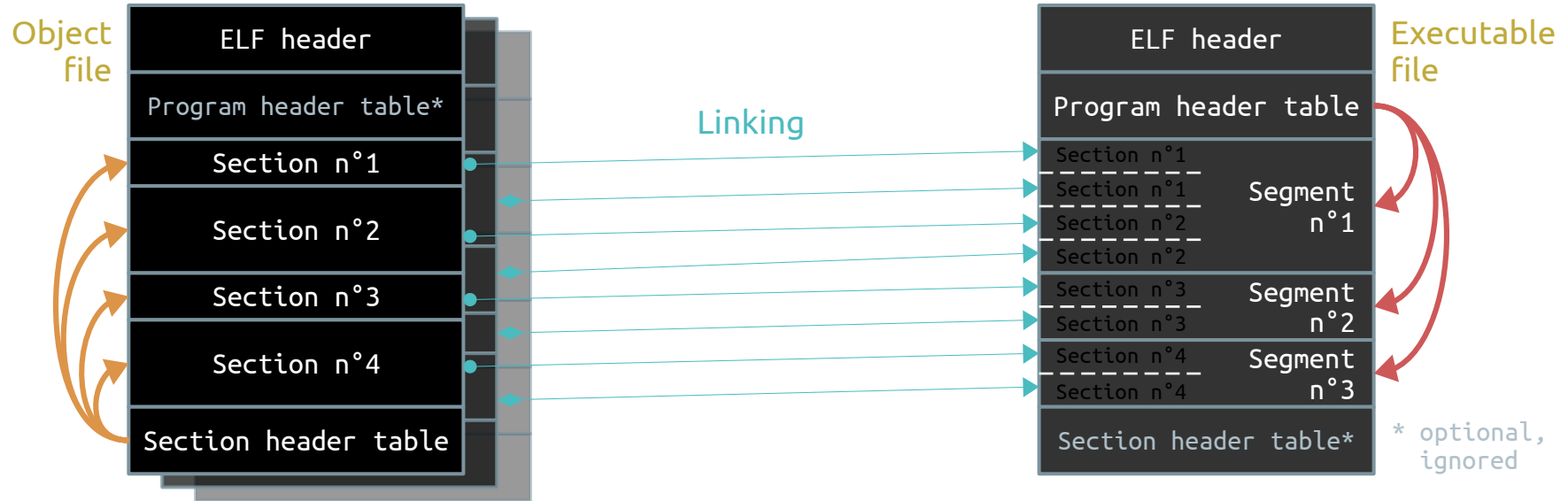
Tous ces fichiers binaires sont au format ELF (*Executable and Linkable Format*).

Format utilisé pour l'enregistrement de code compilé : object (*.o), archive (*.a), shared object (*.so), kernel object (*.ko), core dumps, executable (*, *.out).

Extrêmement répandu sur systèmes UNIX-like (GNU/Linux, FreeBSD, OpenBSD, Solaris, Android, ...) et sur d'autres plateformes (PlayStation 1 à 5, Dreamcast, Nintendo 64 à Wii U, PowerPC, ...), MCU Atmel et Texas Instruments.

Contenu d'un fichier ELF

Un fichier ELF contient toujours un en-tête (*ELF header*). Le reste dépend du type de fichier.



Un fichier objet est décomposé en **sections**, qui sont ensuite listées dans la *Section header table*.

Les sections contiennent des informations pour l'édition des liens et la relocalisation.

Un fichier exécutable contient une *Program header table*, qui décrit les **segments** qui suivent.

Un segment contient les informations utiles à l'exécution du fichier.

Contenu d'un fichier ELF

DISSECTED FILE

```
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
```

SIMPLE.ARM

HEADER^{1/2}

TECHNICAL DETAILS FOR IDENTIFICATION AND EXECUTION

SECTIONS

CONTENTS OF THE EXECUTABLE

HEADER^{2/2}

TECHNICAL DETAILS FOR LINKING (IGNORED FOR EXECUTION)

ELF HEADER

IDENTIFY AS ELF TYPE
SPECIFY THE ARCHITECTURE

PROGRAM HEADER TABLE

EXECUTION INFORMATION

CODE

EXECUTABLE INFORMATION

DATA

INFORMATION USED BY THE CODE

SECTIONS' NAMES

SECTION HEADER TABLE

LINKING CONNECTING PROGRAM OBJECTS INFORMATION

HEXADECIMAL DUMP

ASCII DUMP

FIELDS

VALUES

EXPLANATION

1	e_ident	0x7F, "ELF"	CONSTANT SIGNATURE
	EI_MACH	1	32 BITS, LITTLE-ENDIAN
	EI_CLASS, EI_DATA	1	ALWAYS 1
	EI_VERSION	1	EXECUTABLE
	e_type	2	ARM PROCESSOR
	e_machine	28	ALWAYS 1
	e_version	1	ADDRESS WHERE EXECUTION STARTS
	e_entry	0x00000000	PROGRAM HEADERS OFFSET
	e_phoff	0x40	SECTION HEADERS OFFSET
	e_shoff	0xB0	ELF HEADERS SIZE
	e_eshsize	0x34	SIZE OF A SINGLE PROGRAM HEADER
	e_phsize	0x20	COUNT OF PROGRAM HEADERS
	e_phnum	1	SIZE OF A SINGLE SECTION HEADER
	e_shsize	0x28	COUNT OF SECTION HEADERS
	e_shnum	4	INDEX OF THE NAMES' SECTION IN THE TABLE
	e_shstrndx	3	
2	p_type	1	THE SEGMENT SHOULD BE LOADED IN MEMORY
	p_offset	0	OFFSET WHERE IT SHOULD BE READ
	p_vaddr	0x00000000	VIRTUAL ADDRESS WHERE IT SHOULD BE LOADED
	p_paddr	0x00000000	PHYSICAL ADDRESS WHERE IT SHOULD BE LOADED
	p_filesz	0x90	SIZE ON FILE
	p_memsz	0x90	SIZE IN MEMORY
	p_flags	5	READABLE AND EXECUTABLE

ARM ASSEMBLY

EQUIVALENT C CODE

```
3
mov r2, #13
add r1, pc, #20
mov r0, #1
mov r7, #4
svc #0
write(STDOUT_FILENO, "Hello world!\n", len("Hello world!\n"))

mov r0, #1
mov r7, #1
svc #0
exit(1)
```

STRINGS

"Hello World!\n", 0

SECTION NAMES

.shstrtab .text .rodata

SECTION HEADER TABLE

INDEX	NAME	TYPE	FLAGS	ADDRESS	OFFSET	SIZE
0	<null>	0				
1	.text	1	0x00000000	0x00000000	0x60	0x20
2	.rodata	2	0x00000000	0x00000000	0x80	0x00
3	.shstrtab	3	0x00000000	0x00000000	0x90	0x19

relative offsets in names' section

Fichiers ELF : utilitaires d'analyse

Un fichier binaire ne peut être lu qu'à l'aide d'utilitaires dédiés. Les commandes `readelf` (issue de `binutils`) et `objdump` permettent d'interpréter les informations des fichiers ELF.

Lisons l'en-tête (*ELF header*) de différents fichiers ELF avec la commande "`readelf -h`".

```
dboudier:toolchain$ readelf -h build/obj/hello.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:          ELF32
  Data:          2's complement, little endian
  Version:       1 (current)
  OS/ABI:        UNIX - System V
  ABI Version:   0
  Type:          REL (Relocatable file)
  Machine:       Intel 80386
  Version:       0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 320 (bytes into file)
  Flags:         0x0
  Size of this header: 52 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 40 (bytes)
  Number of section headers: 9
  Section header string table index: 8
```

```
dboudier:~$ readelf -h /lib/i386-linux-gnu/libc.so.6
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  Class:          ELF32
  Data:          2's complement, little endian
  Version:       1 (current)
  OS/ABI:        UNIX - GNU
  ABI Version:   0
  Type:          DYN (Shared object file)
  Machine:       Intel 80386
  Version:       0x1
  Entry point address: 0x1f0a0
  Start of program headers: 52 (bytes into file)
  Start of section headers: 2020040 (bytes into file)
  Flags:         0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 13
  Size of section headers: 40 (bytes)
  Number of section headers: 68
  Section header string table index: 67
```

```
dboudier:toolchain$ readelf -h build/bin/hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:          ELF32
  Data:          2's complement, little endian
  Version:       1 (current)
  OS/ABI:        UNIX - System V
  ABI Version:   0
  Type:          EXEC (Executable file)
  Machine:       Intel 80386
  Version:       0x1
  Entry point address: 0x8049000
  Start of program headers: 52 (bytes into file)
  Start of section headers: 4388 (bytes into file)
  Flags:         0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 3
  Size of section headers: 40 (bytes)
  Number of section headers: 6
  Section header string table index: 5
```

Fichiers objet : table des sections

Lisons maintenant la table des sections d'un fichier objet avec "`readelf -S`".

```

1 /**
2  * @file objdump-minimal.c
3  */
4
5 #include <stdio.h>
6
7 char my_data[] = "Bonjour le monde\n";
8
9 /**
10 * program entry point
11 */
12 int main(void)
13 {
14     char* my_rodata = "Hello World\n";
15
16     printf("%s%s", my_data, my_rodata);
17
18     return 0;
19 }

```

```

dboudier:toolchain$ gcc objdump-minimal.c -c
dboudier:toolchain$ readelf -S -W objdump-minimal.o
There are 14 section headers, starting at offset 0x3a8:

Section Headers:
 [Nr] Name                Type              Address            Off    Size   ES Flg Lk  Inf Al
 [ 0]                      NULL              0000000000000000  000000  000000  00      0  0  0
 [ 1] .text                  PROGBITS          0000000000000000  000040  00003d  00    AX  0  0  1
 [ 2] .rela.text             RELA              0000000000000000  0002b8  000060  18    I 11  1  8
 [ 3] .data                  PROGBITS          0000000000000000  000080  000012  00    WA  0  0 16
 [ 4] .bss                   NOBITS            0000000000000000  000092  000000  00    WA  0  0  1
 [ 5] .rodata                PROGBITS          0000000000000000  000092  000012  00    A  0  0  1
 [ 6] .comment                PROGBITS          0000000000000000  0000a4  00002b  01    MS  0  0  1
 [ 7] .note.GNU-stack        PROGBITS          0000000000000000  0000cf  000000  00      0  0  1
 [ 8] .note.gnu.property     NOTE              0000000000000000  0000d0  000020  00    A  0  0  8
 [ 9] .eh_frame              PROGBITS          0000000000000000  0000f0  000038  00    A  0  0  8
[10] .rela.eh_frame          RELA              0000000000000000  000318  000018  18    I 11  9  8
[11] .symtab                 SYMTAB            0000000000000000  000128  000150  18      12 10  8
[12] .strtab                 STRTAB            0000000000000000  000278  00003a  00      0  0  1
[13] .shstrtab               STRTAB            0000000000000000  000330  000074  00      0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)

dboudier:toolchain$ readelf -l objdump-minimal.o
There are no program headers in this file.

```

Note : on remarque avec "`readelf -l`" qu'il n'y a pas de *program header* dans un fichier objet.

Parmi les nombreuses sections d'un fichier ELF, nous nous concentrons sur celles-ci.

La section `.text` contient le **code binaire du programme**.

Les sections liées à l'**allocation statique** des variables :

- `.bss` : contient les variables globales et variables statiques non-initialisées
- `.data` : contient les variables globales et variables statiques initialisées
- `.rodata` : contient les constantes (variables en lecture seule)

L'**allocation statique** est l'allocation mémoire à la compilation (*compile-time*).

Les variables qui seront utilisées tout au long du programme sont donc stockées dans les fichiers compilés (fichiers objets, puis exécutable).

Petite parenthèse : on verra en CM et TP les deux autres types d'allocation rencontrés.

- L'**allocation automatique**, utilisée pour les variables locales.
- L'**allocation dynamique**, utilisée pour de grandes quantités de données (**malloc**, **free**)

Ces deux types d'allocation s'effectuent pendant l'exécution du programme (*run-time*), par opposition à l'allocation statique qui se fait pendant la compilation (*compile-time*).

Fichiers objet : sections

Analysons le contenu des sections qui nous intéressent avec la commande "`objdump -s`".

On remarque que l'adresse des informations (instructions ou données) est relative au début de section.

```
1/**
2* @file objdump-minimal.c
3*/
4
5#include <stdio.h>
6
7char my_data[] = "Bonjour le monde\n";
8
9/**
10* program entry point
11*/
12int main(void)
13{
14    char* my_rodata = "Hello World\n";
15    printf("%s%s", my_data, my_rodata);
16
17    return 0;
18}
19
```

```
dboudier:toolchain$ objdump -s objdump-minimal.o

objdump-minimal.o:      file format elf64-x86-64

Contents of section .text:
0000 f30f1efa 554889e5 4883ec10 488d0500  ....UH..H...H...
0010 00000048 8945f848 8b45f848 89c2488d  ...H.E.H.E.H..H.
0020 35000000 00488d3d 00000000 b8000000  5....H.=.....
0030 00e80000 0000b800 000000c9 c3          .....

Contents of section .data:
0000 426f6e6a 6f757220 6c65206d 6f6e6465  Bonjour le monde
0010 0a00          ..

Contents of section .rodata:
0000 48656c6c 6f20576f 726c640a 00257325  Hello World..%s%
0010 7300          s.
```

Code binaire
(instructions)

Allocation
statique
(données)

Adresse relative
(format hexadécimal)

Contenu des sections
(représentation hexadécimale)

Contenu des sections
(représentation ASCII)

Fichiers objet : section .text et désassemblage

Il est possible de retrouver le programme en langage d'assemblage avec l'option *disambly*:

```
dboudier:toolchain$ objdump -s objdump-minimal.o
objdump-minimal.o:      file format elf64-x86_64

Contents of section .text:
 0000 f30f1efa 554889e5 4883ec10 488d0500
 0010 00000048 8945f848 8b45f848 89c2488d
 0020 35000000 00488d3d 00000000 b8000000
 0030 00e80000 0000b800 000000c9 c3
```

```
dboudier:toolchain$ objdump -d objdump-minimal.o
objdump-minimal.o:      file format elf64-x86_64

Disassembly of section .text:

0000000000000000 <main>:
 0:  f3 0f 1e fa      endbr64
 4:  55              push    %rbp
 5:  48 89 e5        mov     %rsp,%rbp
 8:  48 83 ec 10     sub     $0x10,%rsp
 c:  48 8d 05 00 00 00 00 lea     0x0(%rip),%rax      # 13 <main+0x13>
13:  48 89 45 f8     mov     %rax,-0x8(%rbp)
17:  48 8b 45 f8     mov     -0x8(%rbp),%rax
1b:  48 89 c2        mov     %rax,%rdx
1e:  48 8d 35 00 00 00 00 lea     0x0(%rip),%rsi      # 25 <main+0x25>
25:  48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi      # 2c <main+0x2c>
2c:  b8 00 00 00 00  mov     $0x0,%eax
31:  e8 00 00 00 00  callq   36 <main+0x36>
36:  b8 00 00 00 00  mov     $0x0,%eax
3b:  c9              leaveq  %eax
3c:  c3              retq
```


Fichiers objet : table des symboles

En lisant la table des sections, on peut noter la présence de la section `.symtab` : il s'agit de la **table des symboles**.

Les fichiers étant traités individuellement pendant les premières phases de la compilation, les données (variables, fonctions, ...) sont employés uniquement par **référence symbolique** (par leur nom).

C'est l'éditeur des liens (*linker*) qui, ayant tous les fichiers nécessaires à sa portée, remplacera ces symboles par leur adresse respective.

```
1/**
2* @file objdump-minimal.c
3*/
4
5#include <stdio.h>
6
7char my_data[] = "Bonjour le monde\n";
8
9/**
10* program entry point
11*/
12int main(void)
13{
14    char* my_rodata = "Hello World\n";
15    printf("%s%s", my_data, my_rodata);
16
17    return 0;
18}
19*/
```

```
dboudier:toolchain$ readelf -s objdump-minimal.o
```

Symbol table '.symtab' contains 14 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	objdump-minimal.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	7	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	8	
8:	0000000000000000	0	SECTION	LOCAL	DEFAULT	9	
9:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
10:	0000000000000000	18	OBJECT	GLOBAL	DEFAULT	3	my_data
11:	0000000000000000	61	FUNC	GLOBAL	DEFAULT	1	main
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_GLOBAL_OFFSET_TABLE_
13:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf

Lire de droite à gauche :

my_data, dans la section 3 (.data), est un *global object* (variable globale) de 18 o. Son adresse dans sa section est 0000.

main, dans la section 1 (.text) est une fonction de 61 octets. Son adresse dans sa section est 0000.

printf, existe mais sa localisation est inconnue.

Adresse dans la section

Symboles
(noms de fonction, de variable, ...)

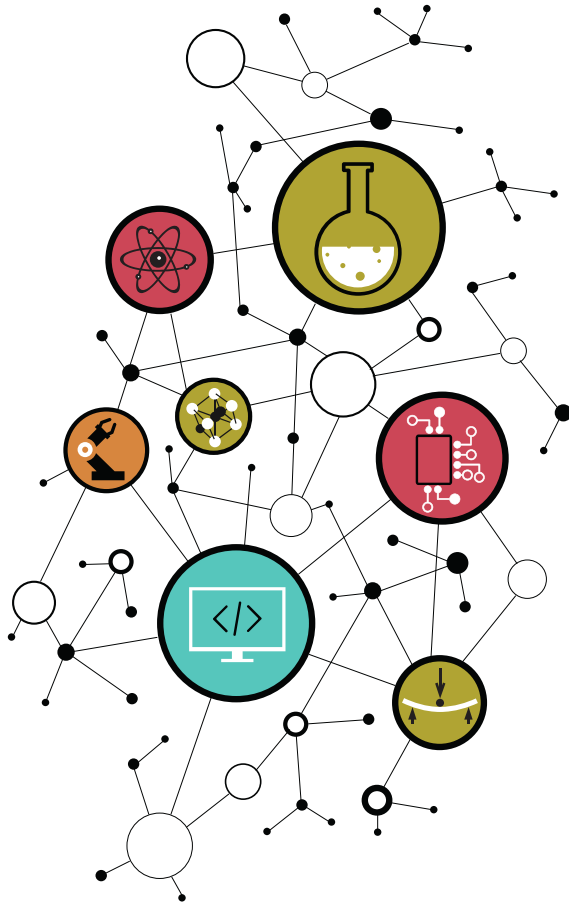
Fichiers exécutable : désassemblage

Passons finalement au désassemblage de l'exécutable.

On remarque que la fonction `printf` est appelée via son symbole et non son adresse. En effet, cette fonction sera appelée à l'exécution puisqu'elle est compilée dans une bibliothèque dynamique (*shared object *.so*)

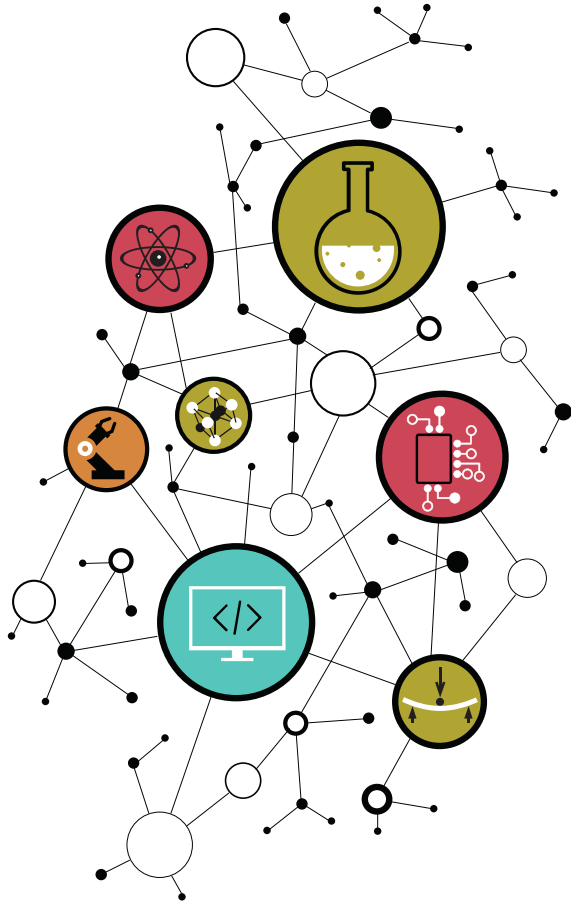
Pour cela, la section `.plt` (*procedure linkage table*) effectue une redirection à l'exécution (*run-time*) vers l'adresse absolue de la procédure cible (`printf` dans notre cas).

```
00000000000001149 <main>:
1149:    f3 0f 1e fa          endbr64
114d:    55                   push    %rbp
114e:    48 89 e5             mov     %rsp,%rbp
1151:    48 83 ec 10          sub     $0x10,%rsp
1155:    48 8d 05 a8 0e 00 00 lea     0xea8(%rip),%rax      # 2004 <_IO_stdin_used+0x4>
115c:    48 89 45 f8          mov     %rax,-0x8(%rbp)
1160:    48 8b 45 f8          mov     -0x8(%rbp),%rax
1164:    48 89 c2             mov     %rax,%rdx
1167:    48 8d 35 a2 2e 00 00 lea     0x2ea2(%rip),%rsi    # 4010 <my_data>
116e:    48 8d 3d 9c 0e 00 00 lea     0xe9c(%rip),%rdi    # 2011 <_IO_stdin_used+0x11>
1175:    b8 00 00 00 00       mov     $0x0,%eax
117a:    e8 d1 fe ff ff       callq   1050 <printf@plt>
117f:    b8 00 00 00 00       mov     $0x0,%eax
1184:    c9                   leaveq  %eax
1185:    c3                   retq
1186:    66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
118d:    00 00 00
```



Description approfondie des fichiers ELF (PDF)

Dissection d'un fichier ELF (png)



Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec la participation de

- Hugo Descoubes (PRAG ENSICAEN)