Chapitre 1

Chaîne de Compilation

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Points-clés


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

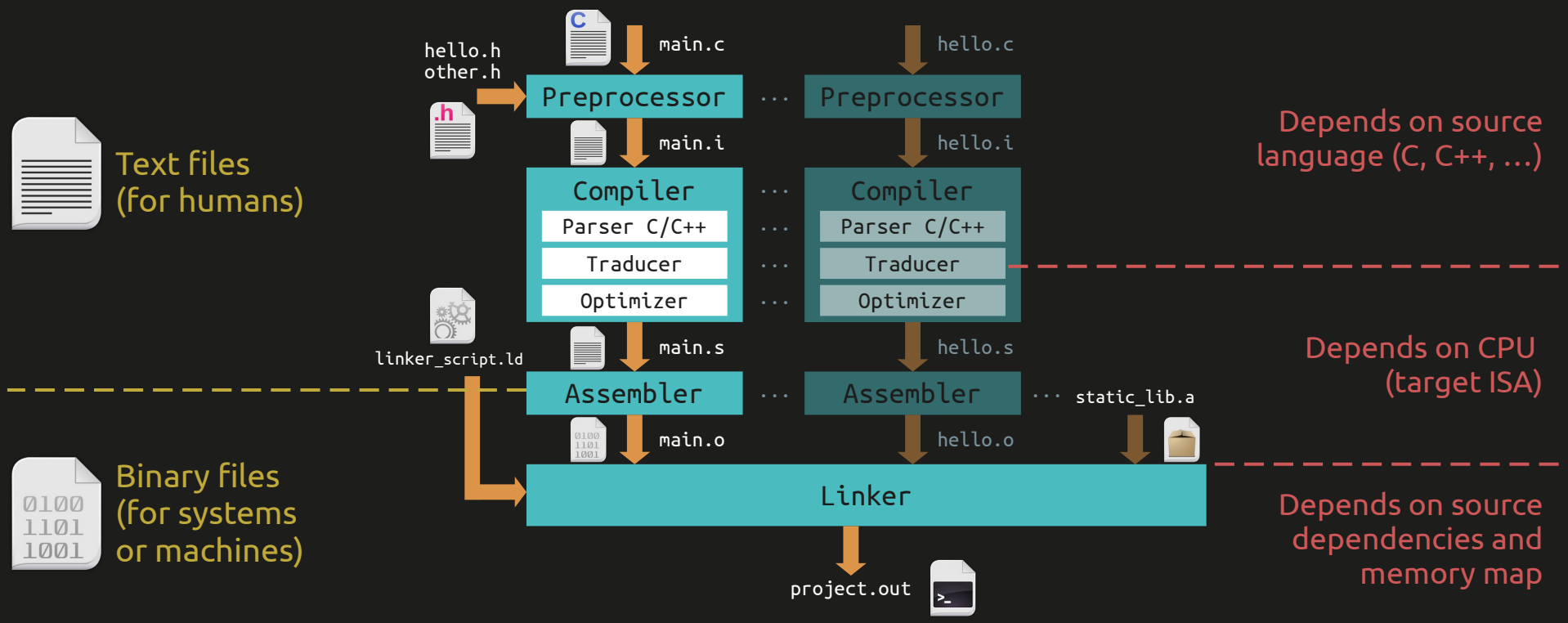

Analyser les fichiers issus des différents étages de la toolchain

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



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

INTRODUCTION

Langage machine vs langage humain

Langage portable vs langage spécifique

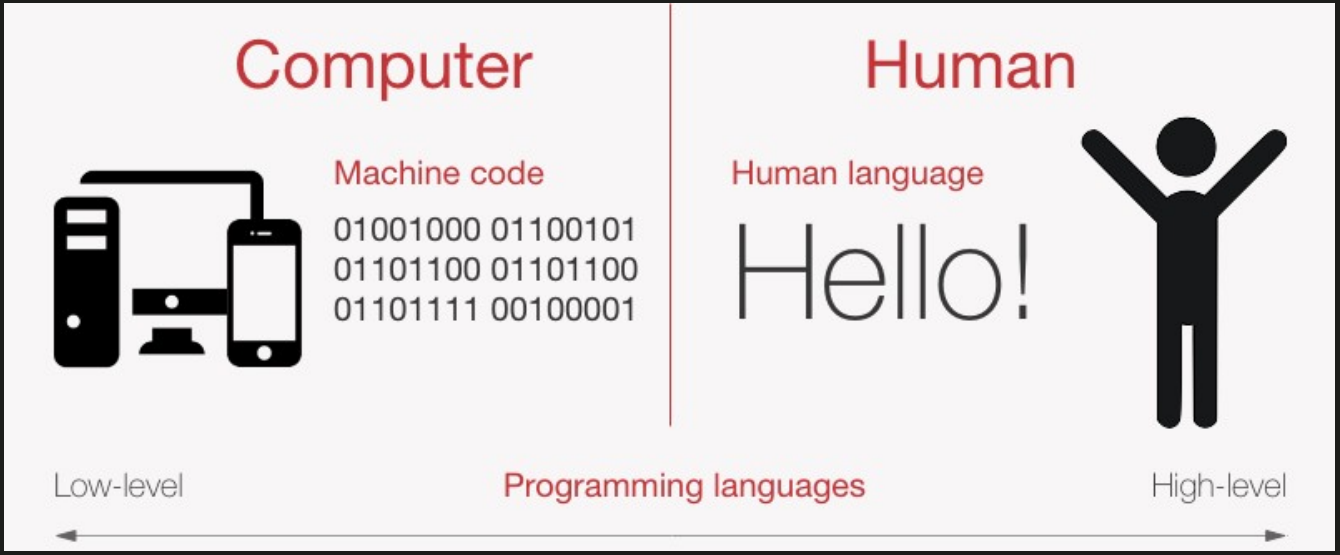
ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Langage humain vs. Langage machine

Le **langage machine** désigne l'ensemble des instructions directement exécutables par un processeur, qui sont des fonctions réalisées avec des circuits électroniques.

Il s'oppose au **langage humain** (au format texte) qui est certes plus facile à rédiger, comprendre et maintenir mais qui n'est directement compréhensible par le processeur.



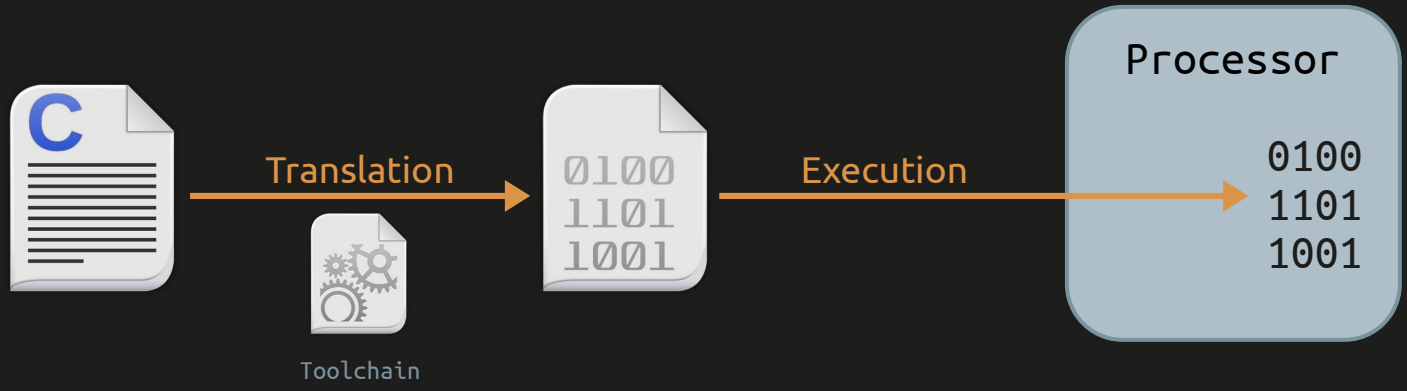
Pour transformer un langage humain en langage machine, il existe deux approches.

La **traduction**

Chaque instruction du programme en langage humain est convertie en instructions en langage machine.
L'ensemble des nouvelles instructions forme un programme en langage machine, qui sera **exécuté plus tard**.

- nécessité de créer un nouveau fichier avant l'exécution, et de le recréer si le programme d'origine a été modifié.

Langages compilés : C, C++, Rust, Java, …



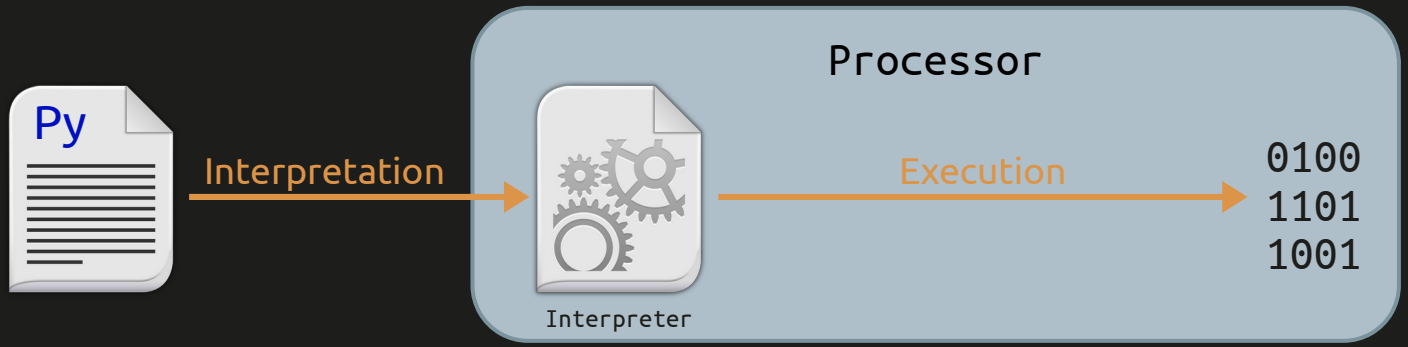
Pour transformer un langage humain en langage machine, il existe deux approches.

L'**interprétation**

Un interpréteur (programme rédigé en langage machine) va, **pendant l'exécution** du programme d'origine, lire chaque instruction en langage humain et la convertir directement en suite d'instructions machine.

- L'interpréteur s'exécute en même temps que le programme = plus lent à l'exécution.

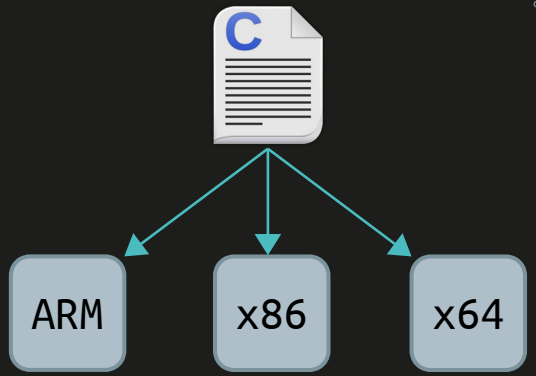
Langages interprétés : JavaScript, Python, PHP, Matlab, …



Langage portable vs. Langage spécifique

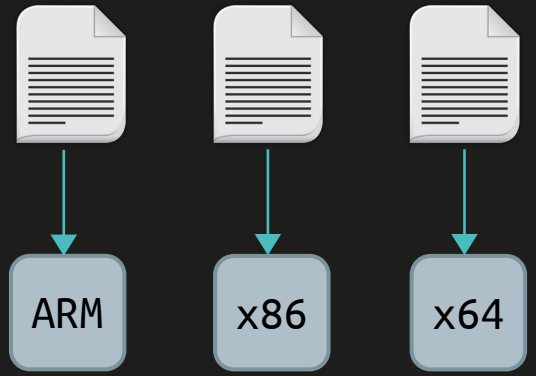
Le langage C est qualifié de **portable**.

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.



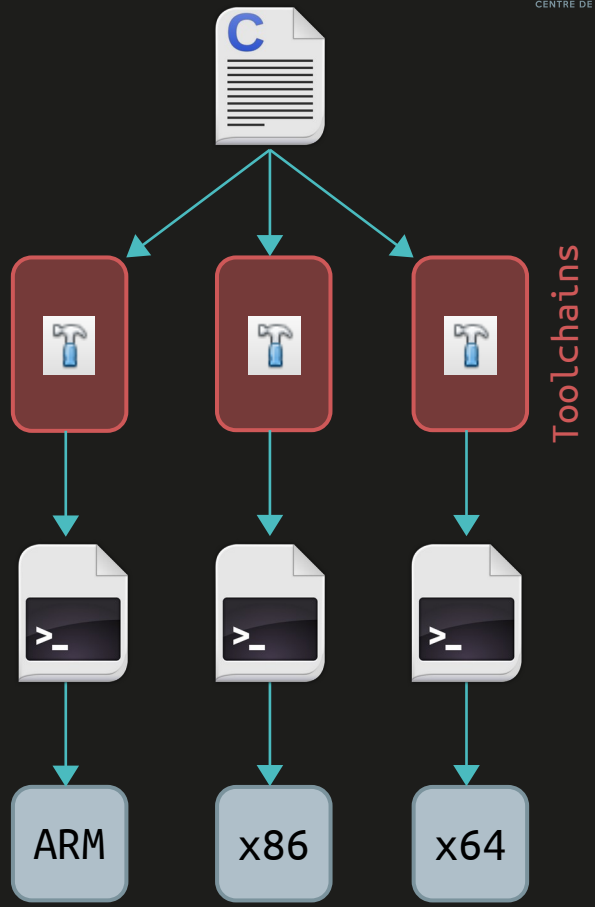
Toutefois, la diapo précédente est imprécise.

“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 traduits 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.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

LA TOOLCHAIN GCC

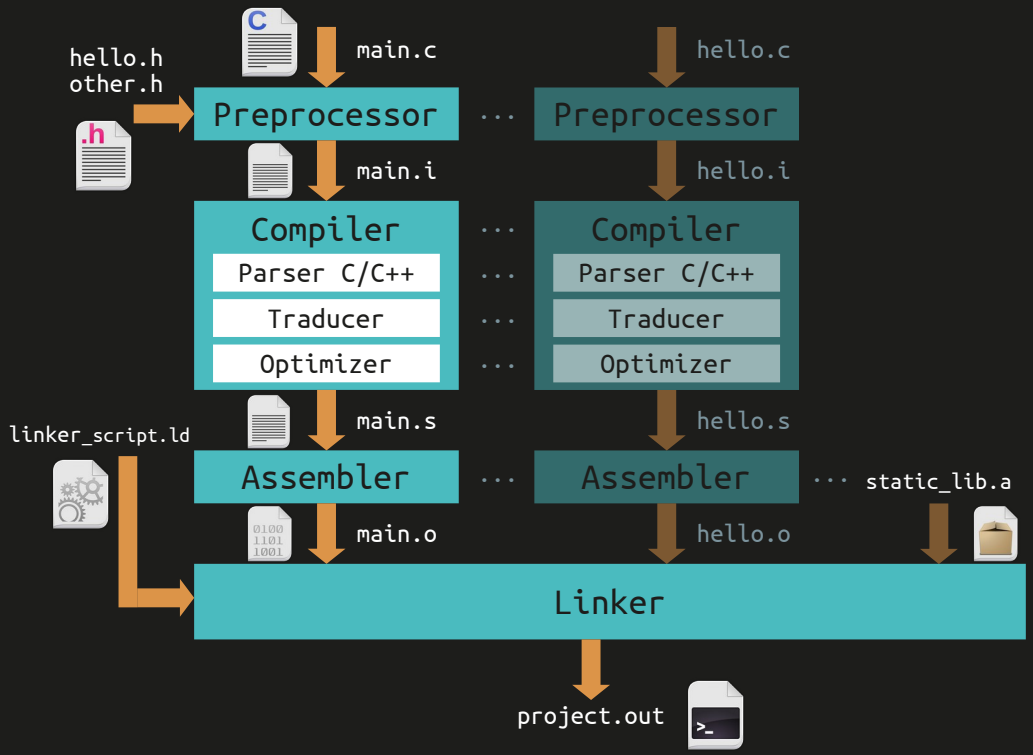
Du fichier source C vers le fichier exécutable

Les pages de cette partie sont à connaître par cœur.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

LA TOOLCHAIN GCC

Vue d'ensemble



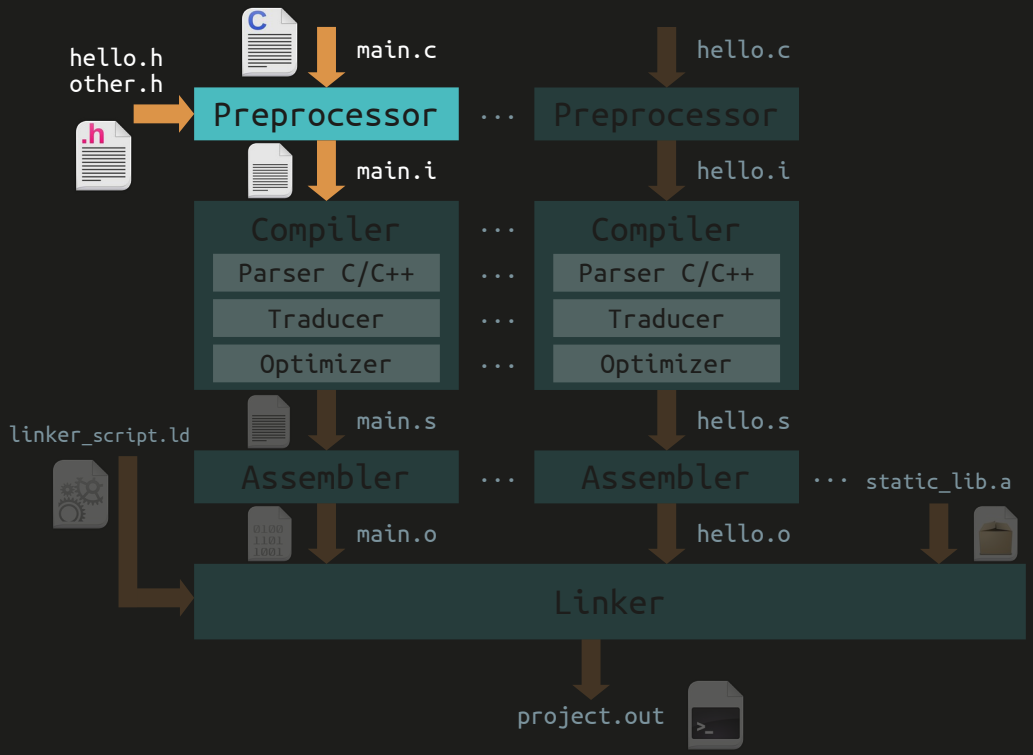
La toolchain est composée de 4 étages, pouvant produire un fichier de sortie.

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 (débute 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`

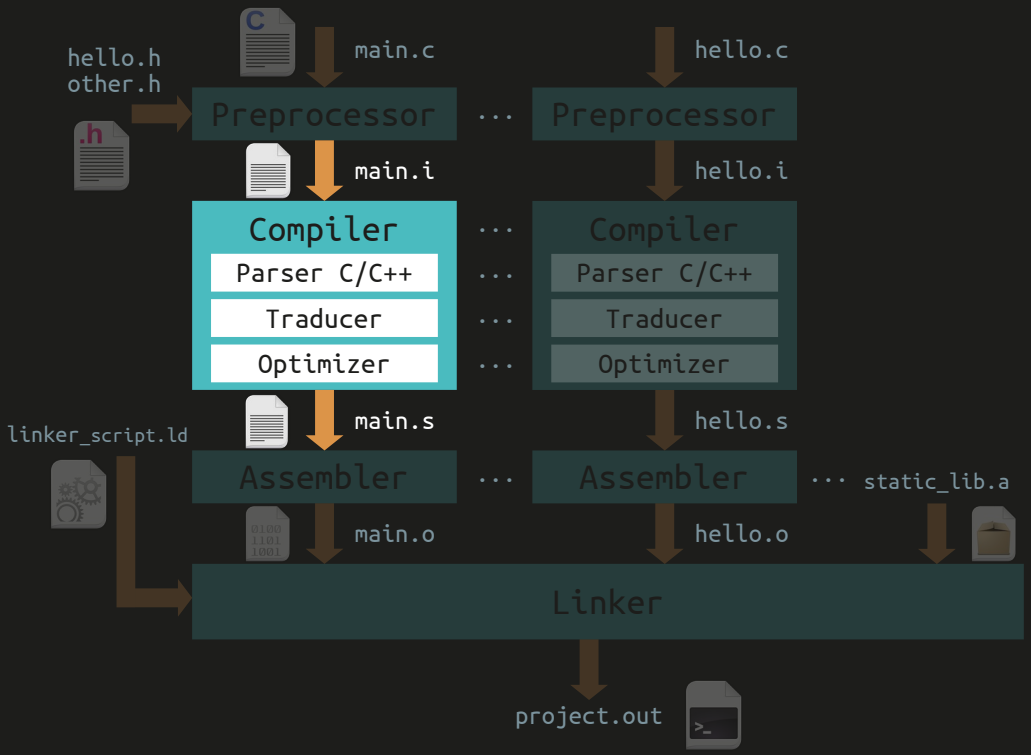
test.c

```
#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.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 :

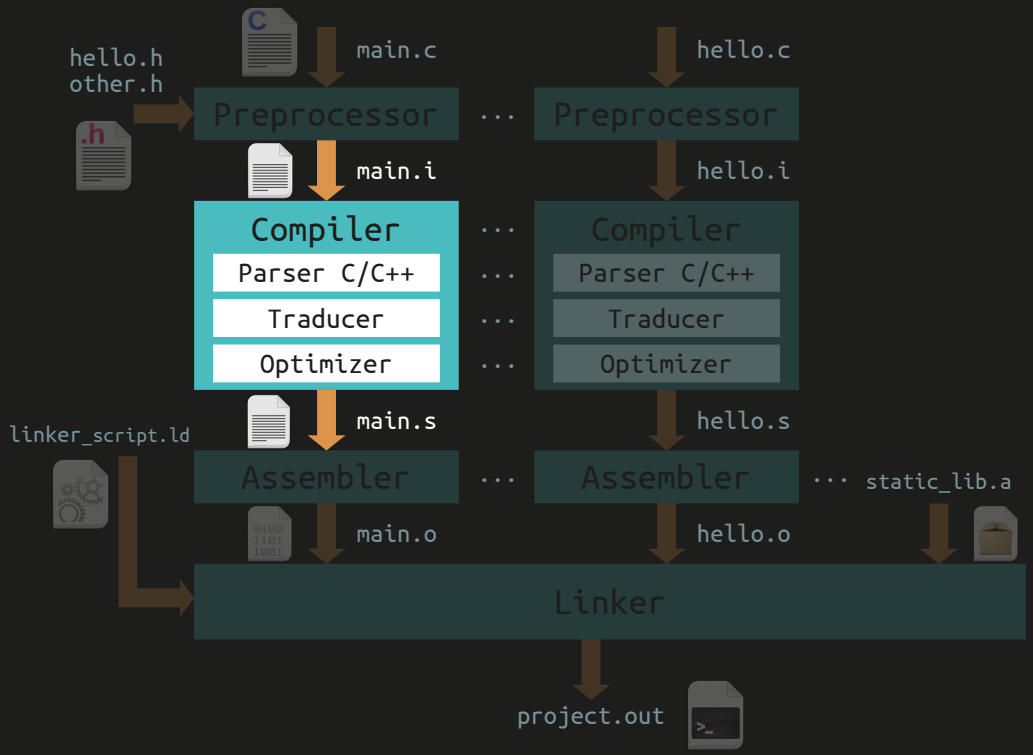
Parser : vérifie le respect des règles du langage.

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 ficher 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

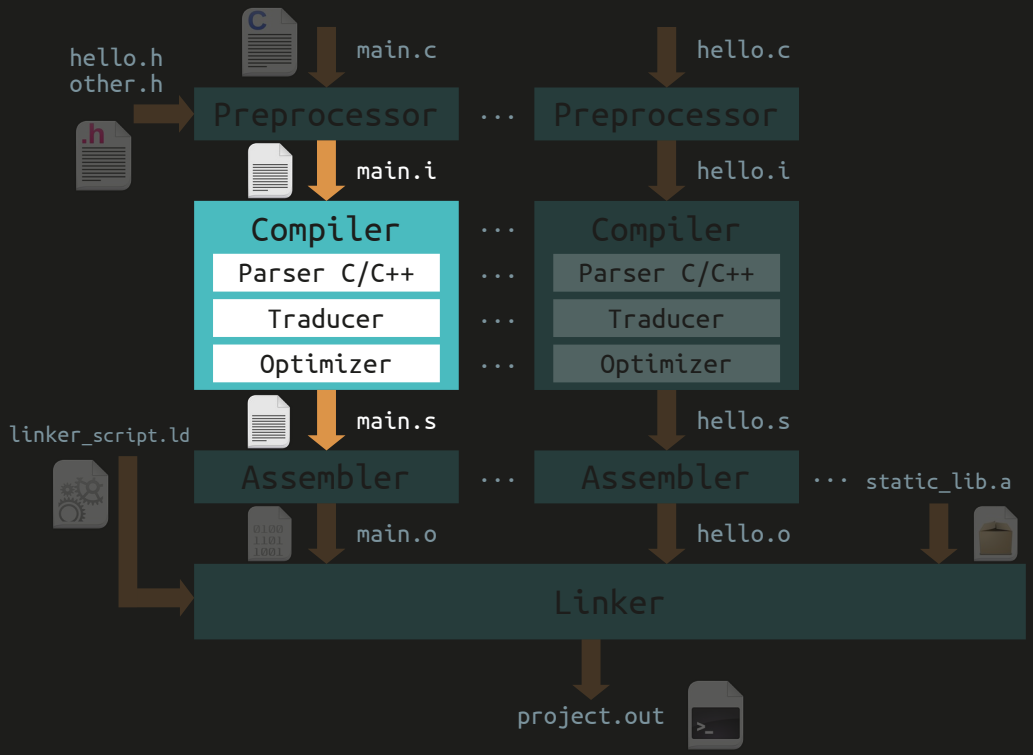
Les phrases font-elles sens ?

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

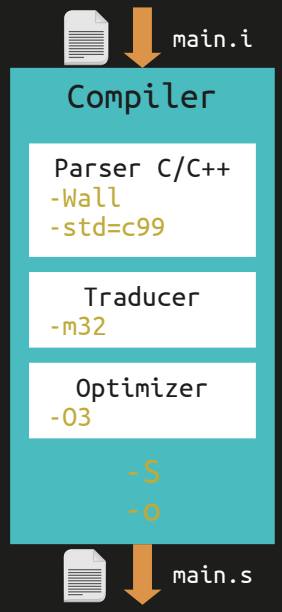
Phase optionnelle. Spécifique à la cible.

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

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

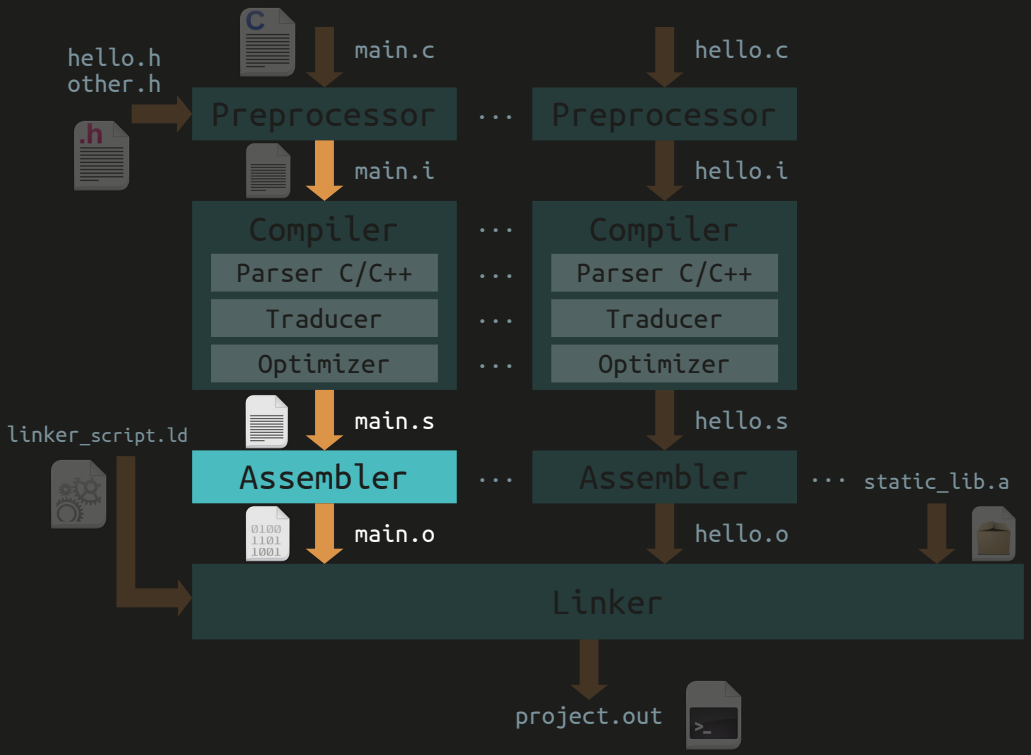
Étage de compilation

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.

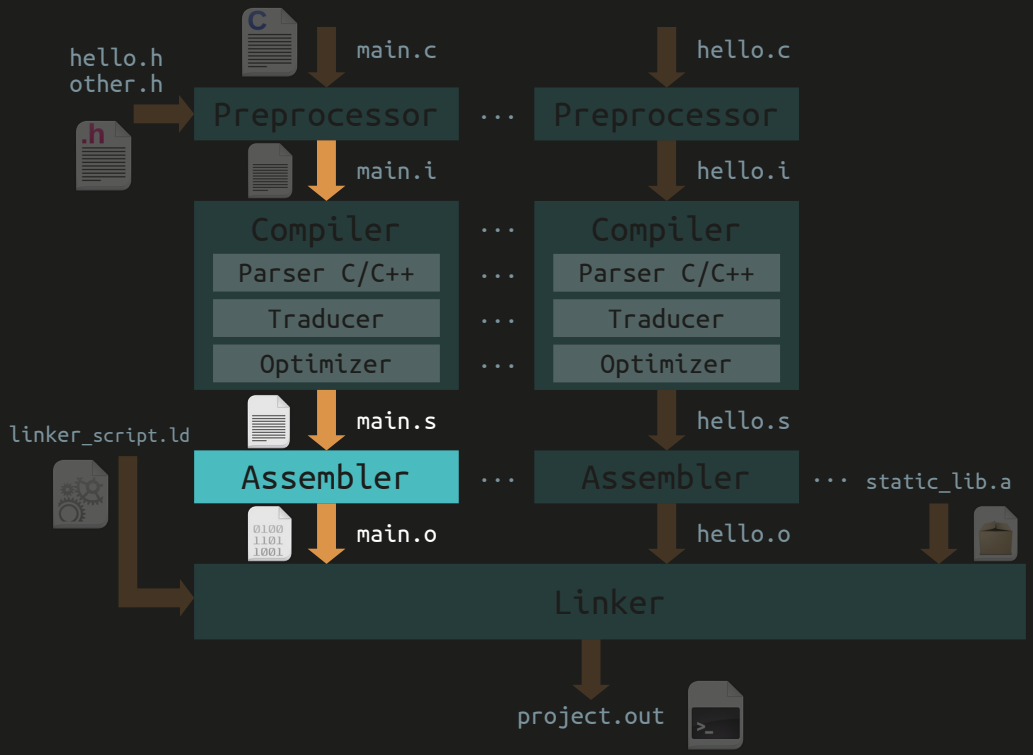
Il utilise des références symboliques : les variables et fonctions externes ont un nom, mais leur adresse est encore inconnue pour le moment.

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 assembleur



L'assembleur fonctionne en deux passes.

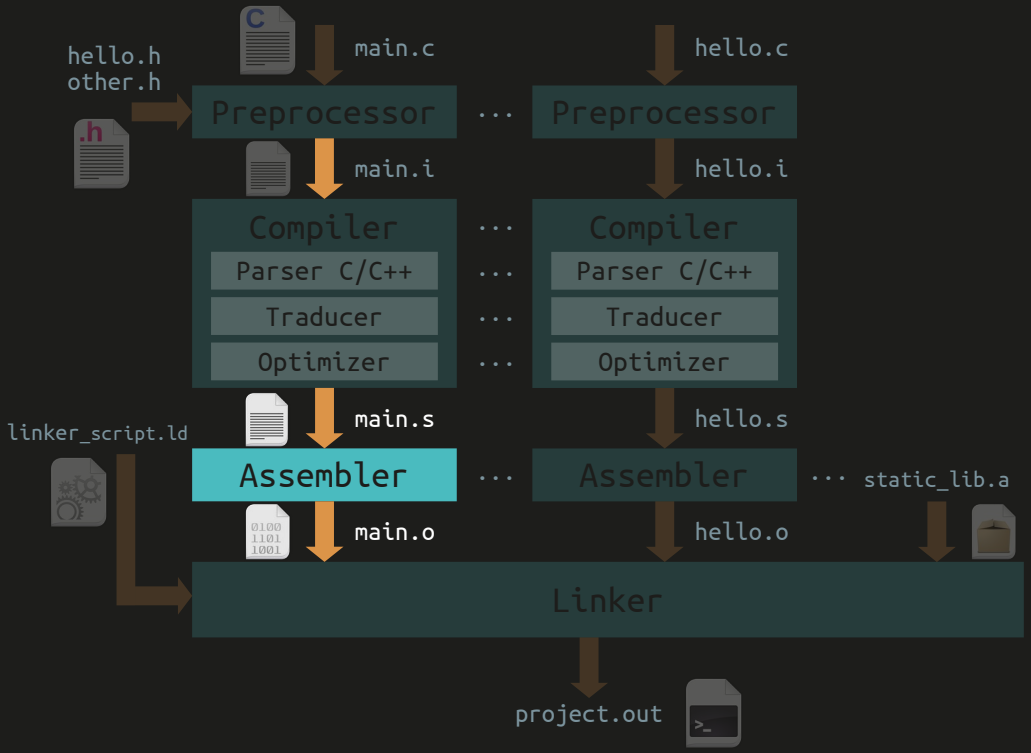
Première passe

Toutes les instructions assembleur sont parcourues pour identifier les différents symboles utilisés (label, nom de constante, …).

Avec ceci, la **table des symboles** est construite.

Celle-ci contient tous les symboles et les valeurs (ou adresses) qui leurs sont associées.

Étage assembleur



L'assembleur fonctionne en deux passes.

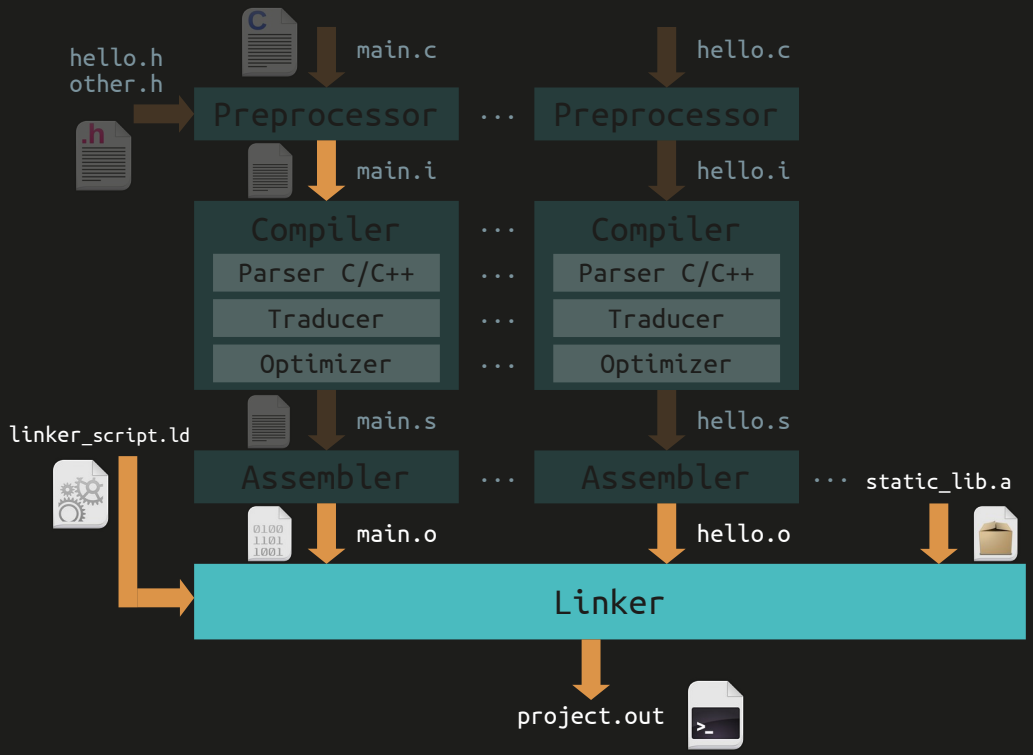
Deuxième passe

Les instructions asm sont traduites en binaire une par une. Si un symbole est rencontré, il est remplacé par sa valeur (listée dans la table des symboles).

Cette passe détecte les erreurs (instruction inconnue, symbole inconnu ou défini plusieurs fois, nombre d'opérandes incorrect, …)

Ainsi le fichier objet est généré et des informations sont fournies pour l'étage suivant (référence à des fonctions externes par exemple).

Étage d'édition des liens



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

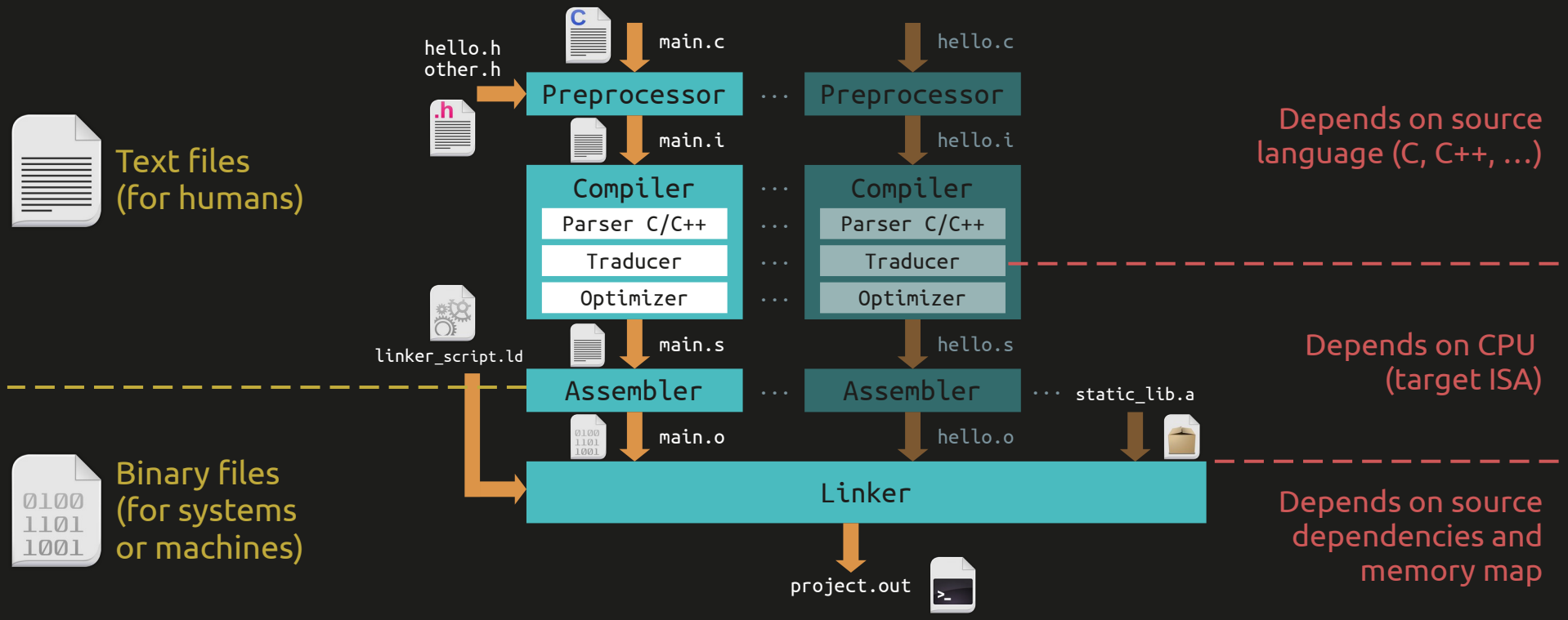
Plus précisément, le *linker* lie les fichiers ensemble : *object files*, *static libraries* (ensemble de fonctions pré-compilées), et *dynamic libraries* (chargées à l'exécution du programme).

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.

Vue d'ensemble



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 compulation avec CMake !

Comment fonctionne Make ? → La commande `make`

Le répertoire courant doit contenir un fichier nommé **`Makefile`** . Celui-ci contient les règles de compilation.

Depuis la console, il suffit de taper la commande « `make` » pour exécuter la première règle, ou bien « `make la_regle` » pour exécuter une règle spécifique.

Le fichier Makefile est dans le répertoire courant

make : la première règle du fichier est appelée. *Dans cet exemple elle sert à générer l'exécutable bin/tp9.*

make clean : la règle "clean" est explicitement appelée. *Dans cet exemple elle sert à supprimer les fichiers obj et exe.*

```
dboudier:tp9$ ls
bin  doc  inc  Makefile  obj  pic  README.md  src

dboudier:tp9$ make
gcc src/main.c -I./inc -c -o obj/main.o
gcc src/BMP.c -I./inc -c -o obj/BMP.o
gcc obj/main.o obj/BMP.o -o bin/tp9

dboudier:tp9$ make clean
rm obj/*
rm bin/*
```

Comment fonctionne Make ? → Le fichier `Makefile`

Un Makefile est composé de règles, chacune étant constituée d'une cible, de pré-requis et d'une commande.

Cible / Target → `mon_exec: fichier.c` ← Pré-requis / Prerequisites

Commande / Command → `gcc fichier.c -o mon_exec`

Quand une règle est évaluée, elle compare d'abord les pré-requis et la cible.

Si au moins un pré-requis est plus récente que la cible (ou si la cible n'existe pas), alors la commande est exécutée.
C'est la cas du premier make ci-contre.

Si la cible est plus récente, alors il ne se passe rien.
Cas du deuxième make de l'exemple.

```
dboudier:howtomake$ ls -lt
total 8                                          "mon_exec" inexistant
-rw-rw-r-- 1 dboudier dboudier 50 mai   21 12:11 fichier.c
-rw-rw-r-- 1 dboudier dboudier 47 mai   21 12:11 Makefile
dboudier:howtomake$ make
gcc fichier.c -o mon_exec  <- Commande exécutée

dboudier:howtomake$ ls -lt
total 28                        "mon_exec" plus récent que "fichier.c"
-rwxrwxr-x 1 dboudier dboudier 16464 mai   21 12:12 mon_exec
-rw-rw-r-- 1 dboudier dboudier    50 mai   21 12:11 fichier.c
-rw-rw-r-- 1 dboudier dboudier    47 mai   21 12:11 Makefile
dboudier:howtomake$ make
make: 'mon_exec' is up to date.  <- Commande NON exécutée
```

Comment fonctionne Make ? → Le fichier `Makefile`

Dans les pré-requis d'une règle, on peut préciser la cible d'autres règles. Ainsi on peut effectuer le processus de compilation en plusieurs étapes. Ceci amène l'avantage de ne recompiler que les fichiers nécessaires et non l'intégralité des fichiers sources.

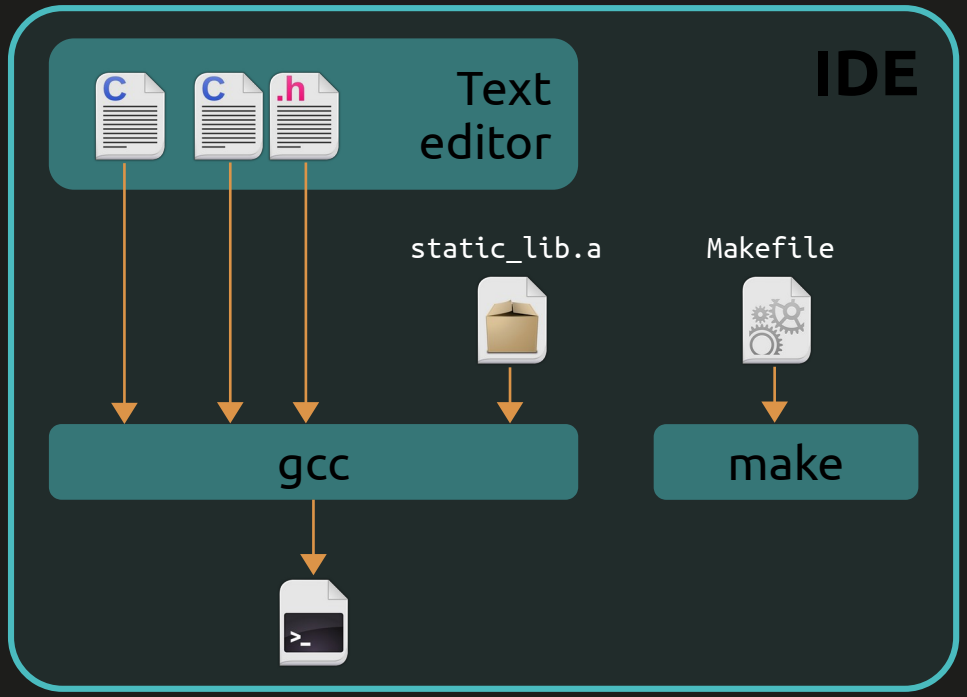
```
# Édition des liens (création de l'exécutable, plusieurs .o -> 1 exe)
bin/tp9: obj/main.o obj/BMP.o
    gcc obj/main.o obj/BMP.o -o bin/tp9

# Compilation de chaque fichier (1 .c -> 1 .o)
obj/main.o: src/main.c inc/BMP.h inc/Image.h
    gcc src/main.c -I./inc -c -o obj/main.o

obj/BMP.o: src/BMP.c inc/BMP.h inc/Image.h
    gcc src/BMP.c -I./inc -c -o obj/BMP.o
```

Quand cette première règle sera appelée, elle devra d'abord évaluer les règles correspondant à ses pré-requis.

```
dboudier:tp9$ make
gcc src/main.c -I./inc -c -o obj/main.o
gcc src/BMP.c -I./inc -c -o obj/BMP.o
gcc obj/main.o obj/BMP.o -o bin/tp9
```

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 (make, Cmake).

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

FICHIERS BINAIRES

Analyse du format ELF

(*Executable and Linkable Format*)

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS

CENTRE DE RECHERCHE

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 binaires : impossible de les lires avec un éditeur de texte, seule la machine peut les comprendre.

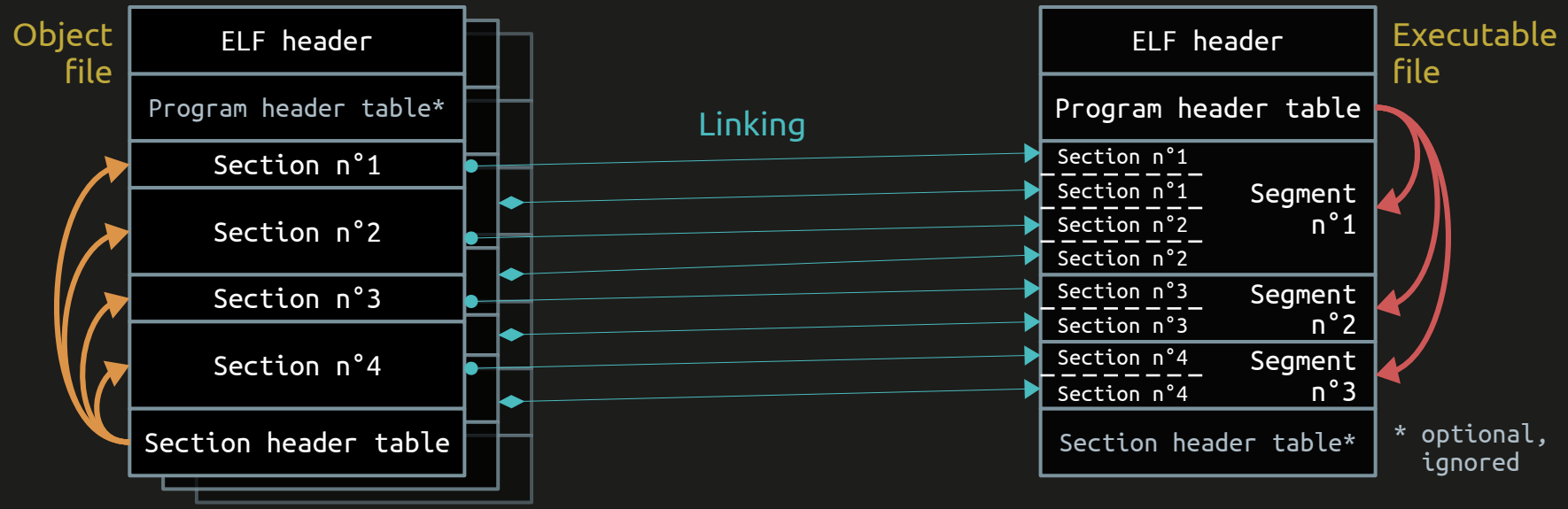
Tous ces fichiers binaires sont au format ELF (*Executable and Linkable Format*).

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.

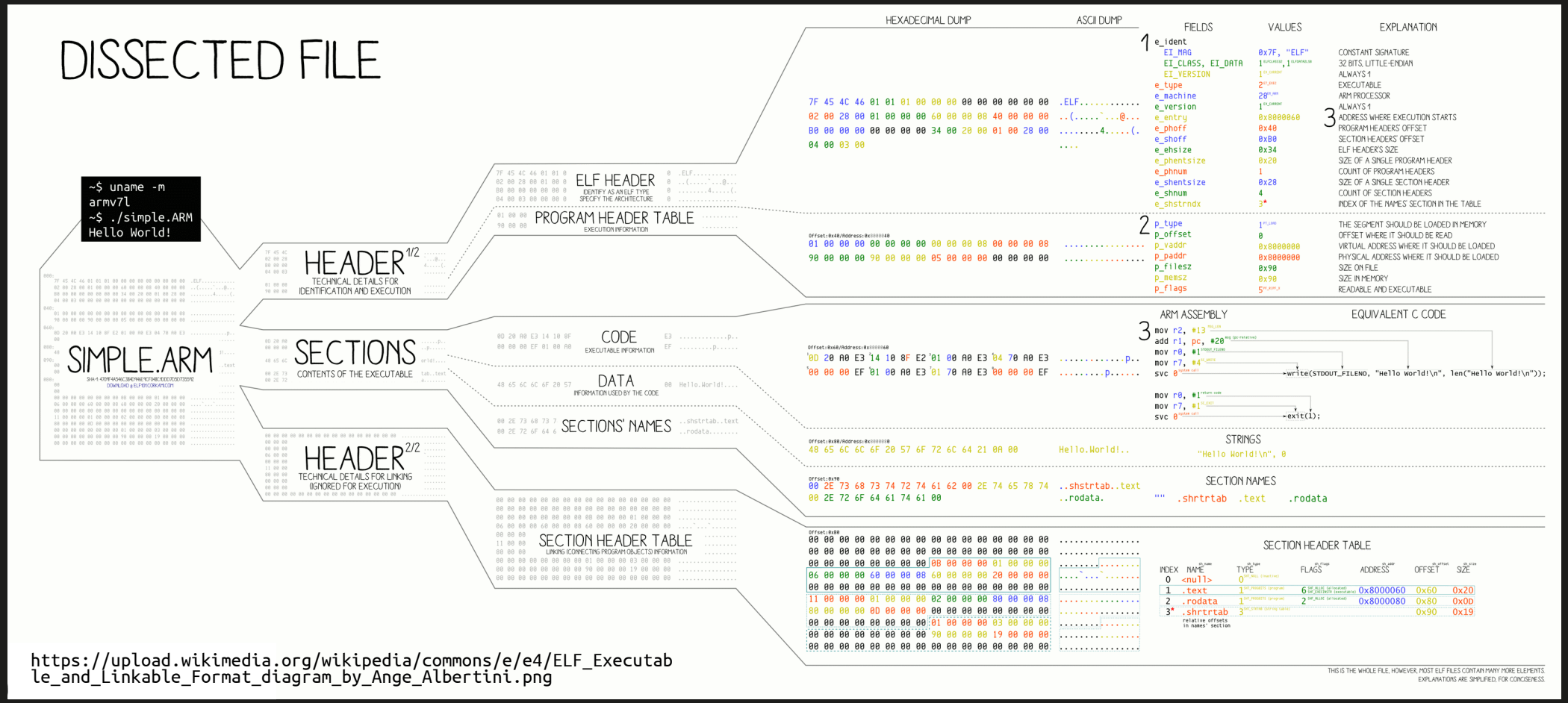


Un fichier objet est décomposé en **sections**, qui sont ensuite listées dans la *Section header table*.

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.



https://upload.wikimedia.org/wikipedia/commons/e/e4/ELF_Executable_and_Linkable_Format_diagram_by_Ange_Albertini.png

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 f
  Start of section headers:          320 (bytes into
  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: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
dress:                               0x8049000
ram headers:                         52 (bytes into file)
ion headers:                         4388 (bytes into file)
                                     0x0
eader:                               52 (bytes)
am headers:                          32 (bytes)
gram headers:                        3
n headers:                           40 (bytes)
tion headers:                        6
 string table index:                 5
```

```
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
```

Fichiers objet : table des sections

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

```c
/**
* @file objdump-minimal.c
*/

#include <stdio.h>

char my_data[] = "Bonjour le monde\n";

/**
* program entry point
*/
int main(void)
{
    char* my_rodata = "Hello World\n";

    printf("%s%s", my_data, my_rodata);

    return 0;
}
```

```
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.

```c
/**
* @file objdump-minimal.c
*/

#include <stdio.h>

char my_data[] = "Bonjour le monde\n";

/**
* program entry point
*/
int main(void)
{
    char* my_rodata = "Hello World\n";

    printf("%s%s", my_data, my_rodata);

    return 0;
}
```

```
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 *disassembly* :

```
dboudier:toolchain$ objdump -s objdump-mini

objdump-minimal.o:     file format elf64-x8

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
  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.

```c
/**
* @file objdump-minimal.c
*/

#include <stdio.h>

char my_data[] = "Bonjour le monde\n";

/**
* program entry point
*/
int main(void)
{
    char* my_rodata = "Hello World\n";

    printf("%s%s", my_data, my_rodata);

    return 0;
}
```

```
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
```

Adresse dans la section

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

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.

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).

```
0000000000001149 <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
    1185:       c3                      retq
    1186:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    118d:       00 00 00
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Description approfondie des fichiers ELF (PDF)

Dissection d'un fichier ELF (png)

CONTACT

Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)