Institut
Mines-Télécom

Chaîne de compilation

Genèse et autopsie des exécutables

Alexis Polti

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr

tl;dr

- **Ce qu'on va apprendre :**
  - écrire du C propre
  - ce que fait un compilateur
  - ce que fait un éditeur de lien
  - comment sont construits les exécutables, à l'octet près
- **Au passage, on va dégommer beaucoup de mythes urbains comme :**
  - "un char fait 8 bits"
  - "GCC est un compilateur"
  - …

Avant tout

- **Installer votre chaîne de cross-compilation**
  - GCC ARM Embedded, maintenue par ARM.
    - https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads
    - L'installer où vous voulez (`/opt` par exemple).
    - Pensez à mettre à jour votre `$PATH` !

- **Taille des entiers**
  - Sur combien de bits sont codés les types suivants ?
    - `char`
    - `short`
    - `int`
    - `long`
    - `long long`

- **Taille des entiers**
  - Sur combien de bits sont codés les types suivants ?
    - `char` : au moins 8
    - `short` : au moins 16
    - `int` : au moins 16
    - `long` : au moins 32
    - `long long` : au moins 64
  - En C99, on dispose dans `stdint.h` de :
    - `int8_t / uint8_t`
    - `int16_t / uint16_t`
    - `int32_t / uint32_t`
    - `int64_t / uint64_t`

- **Taille des entiers**
  - Qu'est-ce qu'un octet / byte ?
  - Qu'affiche le programme suivant ?

```c
#include <stdio.h>
#include <stdint.h>

int main()
{
  uint32_t a= 0x44332211;
  uint8_t *p = (uint8_t *) &a;
  printf("premier octet de a = %x\n", *p);

  return 0;
}
```

2ème commandement de l'UE

- `int` est le type "naturel" du processeur, celui dont les manipulations sont les plus rapides. **On l'emploiera quand on n'a pas spécifiquement besoin d'un autre type**. Exemples : indice de boucles, fd, …

- **Quand on a besoin de connaître la taille de stockage d'une variable, on utilisera les type exacts de C99** : `int8_t`, `uint8_t`, etc. Rien d'autre !

3ème commandement de l'UE

- On utilisera au moins le standard C99 avec les extensions GNU de GCC :

```
gcc -std=gnu99
```

- **Inclusions réciproques**
  - si deux headers se référencent mutuellement, comment les écrire ?
  - exemple :

```
/* a.h */
#include "b.h"
```

```
/* b.h */
#include "a.h"
```

- **Inclusions réciproques**
  - si deux headers se référencent mutuellement, comment les écrire ?
  - exemple :

```
/* a.h */
#ifndef A_H
#define A_H

#include "b.h"

#endif

/* b.h */
#ifndef B_H
#define B_H

#include "a.h"

#endif
```

4ème commandement de l'UE

- Les headers seront toujours protégés contre les inclusions cycliques.

- **Les goto existent !**
  - comme en assembleur, on peut définir des labels et sauter à l'un de ces labels

```
{
  …
start:
  …
  if (…)
    goto end;
  goto start;
  …
end:
  …
}
```

  - attention : à consommer avec modération !
  - http://cs.sjsu.edu/~mak/CS185C/KnuthStructuredProgrammingGoTo.pdf

- **Les goto existent !**
  - cas d'utilisation légitimes :

```
void foo()
{
  if (!try_A())
    goto exit;
  if (!try_B())
    goto cleanupA;
  if (!try_C())
    goto cleanupB;

  // everything succeeded
  return;

 cleanupB:
  undoB();
 cleanupA:
  undoA();
 exit:
  return;
}
```

- gestion des erreurs

```
void foo()
{
  while (...) {
    while (...) {
      if (...)
        goto end;
      // loop action
    }
  }
 end:
  // end action
  ...
}
```

- sortie de boucles imbriquées

- **Type de main en C**
  - quel est le type de `main` ?
  - pourquoi est-ce (très-vraiment-très-très) important ?

- **Type modifier : `const`**
  - `const` : la valeur en question est constante
    - `const int a:?`
    - `const int * a:?`
    - `int const * a:?`
    - `int * const a:?`
    - `const int * const a:?`
  - comment retenir facilement le sens ?

5ème commandement de l'UE

- Tout ce qui est constant sera déclaré `const`.

- **Type modifier `volatile`**
  - `volatile` :
    - la variable peut être modifiée par autre chose que le flot normal de code
      - autre thread
      - handler d'interruption
      - quoi d'autre ?

- **Accès à des périphériques mappés en mémoire**
  - déréférencer une adresse précise
  - comment faire, en C ?

Compléments de C

- **Accès à des périphériques mappés en mémoire**
  - déréférencer une adresse précise
  - comment faire, en C ?

```
// Contrôleur de GPIO en 0xA300020
volatile uint32_t * const gpio_config = (uint32_t *) 0xA300020;

// Utilisable ainsi :
*gpio_config = 20;
uint32_t value = *gpio_config;

// Autre possibilité
#define GPIO_REG (*(volatile uint32_t *) 0xA300020)

GPIO_REG = 20;
value = GPIO_REG;
```

6ème commandement de l'UE

- pour accéder à un registre mappé en mémoire, on utilisera cette construction :

```
#define REG (*(volatile uint32_t *)0xff00ff00)
```

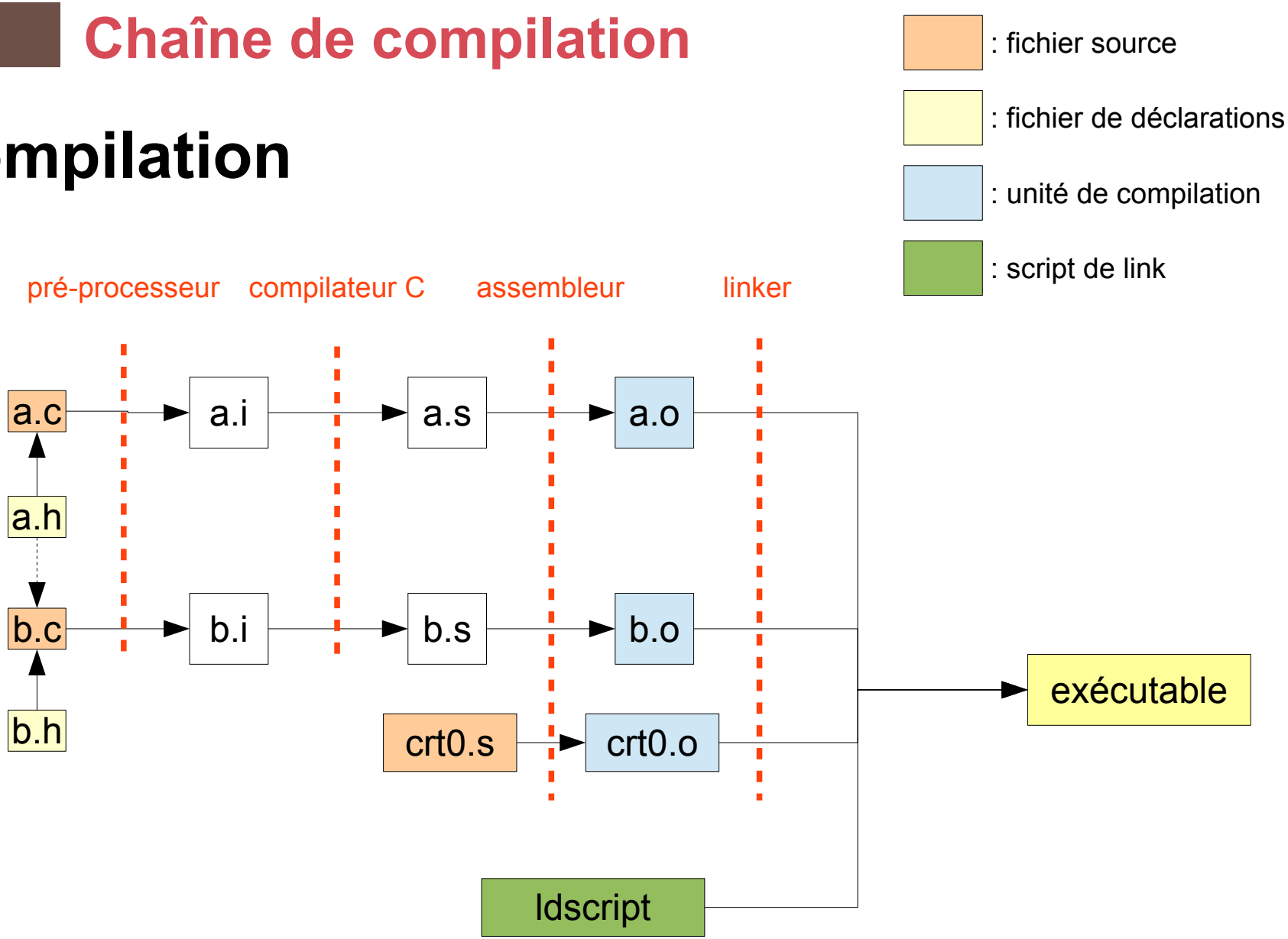
- dans d'autres UE, on verra d'autres constructions tout aussi élégantes, notamment pour des sets de registres.

Où en est-on ?

- **On a vu**
  - quelques rappels importants de C
- **On va voir maintenant**
  - ce que fait exactement une chaîne de compilation
  - ce que fait un compilateur
  - écrire du C propre
  - à quoi sert un éditeur de lien
- **On verra plus tard**
  - comment sont architecturés les exécutables
  - comment piloter l'éditeur de lien pour produire l'exécutable qu'on veut.

Chaîne de compilation

- **Compilation**



Chaîne de compilation : GCC

- **Exemple : GCC**
  - GCC : GNU Compiler Collection
    - compilateurs
    - pré-processeurs
    - driver (`gcc -v` / `gcc -###`)
  - binutils
    - éditeur de lien (linker) : `ld` / `gold` / `collect2`
    - assembleur : `as`
    - `objcopy`, `objdump`, `gprof`, `strip`, `readelf`, `nm`, `size`
  - GNU Debugger : `gdb`
  - Bibliothèque C
    - glibc, eglibc
    - newlib, tinylibc, dietlibc
    - klibc

Chaîne de compilation : GCC

- **Pré-processeur : `cpp`**
  - processeur texte
  - interprète les directives de compilation
    - `#include`
    - `#define`
    - `#ifdef`
    - ...
  - produit un fichier C pré-processé (`.i`, `.ii` pour le C++)
  - on peut voir le fichier résultant avec `gcc -E`
  - `-Dname / -Dname=value` : définit des macros
  - `-Uname` : annule la définition de macros

Chaîne de compilation : GCC

- Pré-processeur : fichiers `.h`
  - user include files : `#include "file.h"`
  - system include files : `#include <file.h>`

  - `-iquote dir` :
    - ajoute `dir` à la liste où sont cherchés les headers utilisateurs

  - `-isystem dir` :
    - ajoute `dir` à la liste où sont cherchés les headers systèmes

  - `-I dir` :
    - ajoute `dir` à la liste où sont cherchés les headers

  - `-nostdinc` :
    - limite la recherche des headers à « `.` » et aux répertoires spécifiés par `-I` et `-iquote`

Chaîne de compilation : GCC

- **Compilation**
  - production d'un fichier assembleur : `gcc -S`
  - production d'un objet : `gcc -c`

  - cross compilation : le compilateur produit du code pour une cible différente de la machine où il s'exécute

compilation du compilateur	exécution du compilateur	exécution du code	nom
x86	x86	x86	natif
x86	x86	ARM	cross
x86	SH	x86	crossback
x86	SH	SH	crossed native
x86	SH	PPC	canadian cross

  - un cross-compilateur se comporte comme un compilateur natif
  - certaines cibles ont des options particulières :

    `gcc --target-help` pour les connaître

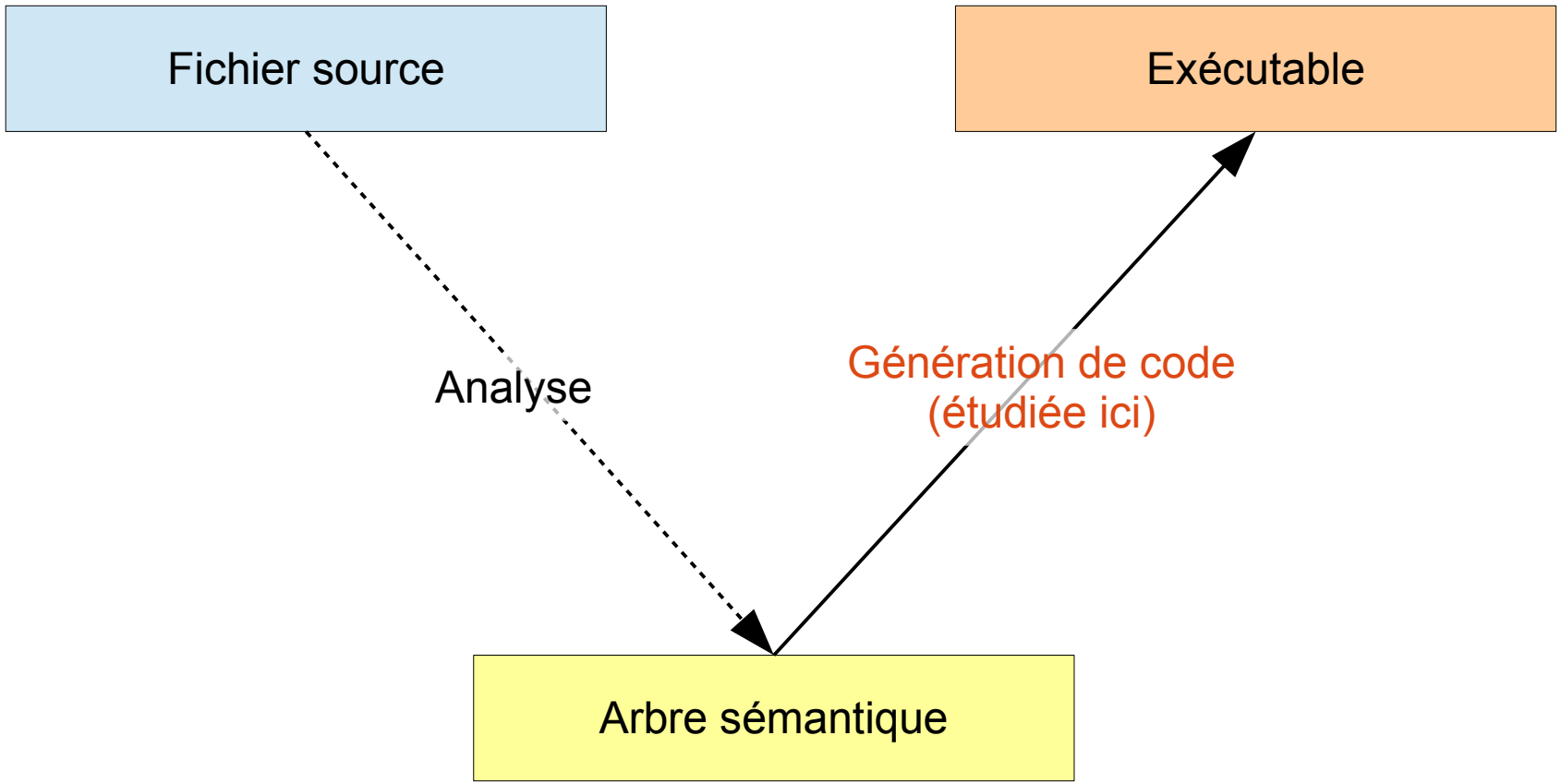
Où en est-on ?

- **On a vu**
  - quelques rappels importants de C
  - les chaînes de compilation
- **On va voir maintenant**
  - ce que fait un compilateur
  - écrire du C propre
  - à quoi sert un éditeur de lien
- **On verra plus tard**
  - comment sont architecturés les exécutables
  - comment piloter l'éditeur de lien pour produire l'exécutable qu'on veut.

Bases de compilation

- **Code**
  - nous utiliserons la définition suivante de **code** :
    suite d'instructions destinées à un ordinateur
  - exemples de code :
    - *code C* : programme en langage C
    - *code machine* : suite de mots binaires directement exécutables par un processeur
    - *bytecode* : suite de mots exécutables par une machine virtuelle
    - *pseudocode* : suite d'instructions à effectuer, facilement compréhensibles par des humains

- **Processus de compilation**



Bases de compilation

- **Étapes**
  - La définition d'un langage intermédiaire facilite la réutilisation :
    - M parties frontales et N générateurs de code permettent d'écrire M + N + 1 fragments plutôt que M N
  - Le générateur de code peut produire
    - soit directement du code machine
    - soit de l'assembleur
      - possibilité d'optimisations par l'assembleur : calcul de déplacement, etc.

Bases de compilation

- **Frontal (front-end)**
  - La partie frontale d'un compilateur
    - transforme le code en un arbre syntaxique
    - construit les associations sémantiques
    - vérifie la syntaxe et la sémantique du code
  - Elle peut également
    - opérer des transformations sur l'arbre (optimisations, simplifications)
    - générer des informations de haut-niveau (nombre de lignes de code, présence de code mort, …)

Bases de compilation

- **Exemple : GCC**
  - GCC (GNU Compiler Collection) utilise un langage intermédiaire : gimple
  - GCC génère un fichier assembleur temporaire
  - L'optimisation se fait à chaque niveau
  - Il est facile de rajouter :
    - un nouveau langage
    - une nouvelle cible
  - GCC peut être configuré en n'importe quelle configuration (natif, cross, cross back, crossed native, canadian cross)

- **Génération de code**
  - But : permettre à chaque sous-programme présent dans l'arbre d'être appelé
  - Moyens :
    - pour chaque instruction, générer du code effectuant les bonnes opérations
    - générer du code pour l'entrée (prologue) et la sortie (épilogue) du sous-programme

Bases de compilation

- **Code machine**
  - Le code machine
    - est simple : x^y n'existe pas
    - a peu d'arguments : `f(u, v, w, x, y, z)` est impossible
    - est peu structuré : `for(i=0; i<10; i++)` n'est pas représentable simplement
  - Il faut transformer l'arbre en instructions élémentaires

- **Constructions ternaires**
  - classiquement, chaque opération est transformée en une suite de constructions généralement ternaires, car cela :
    - correspond aux possibilités usuelles d'un microprocesseur
    - permet d'optimiser indépendamment chaque instruction
    - permet d'unifier les sous-expressions communes
  - certains jeux d'instructions ARM disposent de quelques opérations plus que ternaires (MLA, STM, LDM).

Bases de compilation

- **Constructions usuelles**
  - sur les architectures load / store :
    - une instruction ne peut manipuler que des registres
    - l'accès à la mémoire se fait par les instructions LDR et STR
  - pour accéder à une variable, il faut donc :
    - d'abord stocker son adresse (connue à l'édition de lien) dans un registre
    - puis accéder à la mémoire (LDR ou STR)

Bases de compilation

- **Constructions usuelles (ARM)**
  - avec GCC,un symbole représente
    - en C : *la valeur* d'une variable
    - en assembleur ou linker script : *l'adresse* de cette variable
  - ainsi, avec a entier 32 bits, `a = 3;` pourrait devenir :

```
0:  ldr r3, [pc, #4]    ; r3 ← &a
4:  mov r2, #3          ; r2 ← 3
8:  str r2, [r3]        ; 3 → mem[&a]
c:  .word a             ; adresse de a
```

```
0:  ldr r3, =a          ; r3 ← &a
4:  mov r2, #3          ; r2 ← 3
8:  str r2, [r3]        ; 3 → mem[&a]
```

Bases de compilation

- **Constructions usuelles (ARM)**

```
int32_t a, b;
a = b;
```

pourrait devenir :

```
0:    ldr r3, =a          ; r3 ← &a
4:    ldr r2, =b          ; r2 ← &b
8:    ldr r2, [r2]        ; r2 ← b
c:    str r2, [r3]        ; b → mem[&a]
```

Bases de compilation

- **Constructions usuelles (ARM)**

```
int32_t a, b, c;
a = a + b*c;
```

pourrait devenir :

```
 0:   ldr   r3, =a           ; r3 ← &a
 4:   ldr   r1, =b           ; r1 ← &b
 8:   ldr   r2, =c           ; r2 ← &c
 c:   ldr   r0, [r1]         ; r0 ← b
10:   ldr   r2, [r2]         ; r2 ← c
14:   ldr   r1, [r3]         ; r1 ← a
18:   mla   r2, r0, r2, r1   ; r2 ← b*c + a
1c:   str   r2, [r3]         ; r2 → mem[&a]
```

Bases de compilation

- **Constructions usuelles (ARM)**

```
*p++ = 3;
```

pourrait devenir :

```
 0: ldr   r2, =p          ; r2 ← &p
 4: ldr   r3, [r2]        ; r3 ← p
 8: mov   r1, #3          ; r1 ← 3
 c: str   r1, [r3], #4    ; 3 → mem[p] puis r3 ← p+4
10: str   r3, [r2]        ; p+4 → mem[&p]
```

Bases de compilation

- **Constructions usuelles (ARM)**
  - Sauts simples :

En C :

```
a:
   ...
   goto a;
```

devient :

```
a:
   ...
   b a
```

En C :

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

devient :

```
   ...
   bl f
   ...
```

Bases de compilation

- **Constructions usuelles**
  - Les boucles sont généralement ré-écrites selon la structure suivante :

    - 1 : initialisation des paramètres de la boucle
    - 2 : saut par-dessus l'étape 3 si le test est en fin de boucle
    - 3 : test de sortie, saut après 6 si positif
    - 4 : corps de la boucle
    - 5 : exécution de la partie finale de la boucle
    - 6 : saut inconditionnel en 3
    - 7 : suite du programme

Bases de compilation

- **Constructions usuelles**
  - Exemple : `for (i=0;i<10;i++) {...}`

```
1 : i = 0
2 :
3 : if NOT(i < 10) goto 7
4 : ...
5 : i = i + 1
6 : goto 3
7 :
```

Bases de compilation

- **Constructions usuelles**
  - Exemple : `while(c) {...}`

```
1 :
2 :
3 : if NOT(c) goto 7
4 : ...
5 :
6 : goto 3
7 :
```

Bases de compilation

- **Constructions usuelles**
  - Exemple : `do {...} while(c)`
    - 1 :
    - 2 : `goto 4`
    - 3 : `if NOT(c) goto 7`
    - 4 : `...`
    - 5 :
    - 6 : `goto 3`
    - 7 :

Chouette, des exercices !

- **À vous de travailler :**
  - exercice 1 : facile
  - exercice 2 : moyen

Exercices de compilation

- **Exercice 1**
  - Traduire en assembleur ARM le code suivant :

```
uint32_t a; // global variable
...
for (uint8_t i=0; i<=a; i++)
   g();
```

  - Même question si `i` est un `unsigned int`. Conclusion ?

  - Indice : pour voir ce que produit GCC pour ARM :

```
uint32_t a; // global variable

__attribute__((naked)) void f() {
    for (uint8_t i=0; i<=a; i++)
        g();
}
```

Puis : `arm-none-eabi-gcc -Os -S t.c`

Exercices de compilation

- **Exercice 2**
  - Traduire en assembleur ARM le code suivant :

```
// Global variables
uint32_t *a;
uint32_t *b;
uint32_t *c;
…
*a += *c;
*b += *c;
```

Exercices de compilation

- **Exercice 2 (suite)**
  - Comparez avec ce que produit GCC, ainsi :

```
// Global variables
uint32_t *a;
uint32_t *b;
uint32_t *c;

__attribute__((naked)) void f() {
    *a += *c;
    *b += *c;
}
```

Puis : `arm-none-eabi-gcc -O2 -S t.c`

  - Pourquoi GCC charge-t-il deux fois le contenu de `*c` au lieu d'une seule ?

Où en est-on ?

- **On a vu**
  - quelques rappels importants de C
  - les chaînes de compilation
  - ce que fait un compilateur
- **On va voir maintenant**
  - comment écrire du C propre
  - à quoi sert un éditeur de lien
- **On verra plus tard**
  - comment sont architecturés les exécutables
  - comment piloter l'éditeur de lien pour produire l'exécutable qu'on veut.

- **Déclaration vs. instanciation des variables**
  - une variable possède plusieurs caractéristiques :
    - un nom (appelé symbole)
    - un type
    - et, si elle réside en mémoire, une adresse
  - **déclaration** :
    - association nom ↔ type
    - exemple : `extern int a;`
    - empêche l'initialisation : ~~`extern int a=2;`~~
  - **instanciation** / **définition** :
    - déclaration + allocation de la mémoire pour stocker la variable
    - exemple : `int a;`
    - exemple : `int a=3;`

- **Déclaration et instanciation des variables**
  - le compilateur peut générer du code utilisant une variable même si son adresse n'est pas encore connue
    - il n'a besoin que d'en connaître le type → la déclaration
    - il lui assigne une adresse temporaire (nulle)
    - cette adresse sera rectifiée lors de l'édition de lien
  - conséquence :
    - une variable ne doit être **instanciée qu'une seule fois**, dans un fichier source **.c**
      - exception : symboles `weak`, qu'on verra après
    - une variable **exportée** doit **en plus** :
      - **être déclarée** dans le header .h correspondant au fichier .c où elle est instanciée
      - ce header sera **inclus par tous les .c utilisant cette variable**

- **Déclaration et instanciation des variables**
  - Exemple :

```c
/* a.h */
extern int a;
```

```c
/* a.c */
#include "a.h"
int a=3;

void foo() {
   a=a+1;
}
```

```c
/* b.c */
#include "a.h"

void bar() {
   a=23;
}
```

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

void baz() {
   g(a);
}
```

- **Déclaration et instanciation des variables**
  - ATTENTION PIÈGE :
    - avec le linker de GCC sous Linux, des variables de même noms instanciées dans des unités de compilation différentes sont considérées comme une seule et même instance, même si elles n'ont pas le même type !!!
    - une erreur n'est émise **que** si plus d'une instance est initialisée

- **Déclaration et instanciation des variables**
  - ATTENTION PIÈGE : que donne le code (sale) suivant ?

```c
/* a.c */
#include <stdint.h>
#include <stdio.h>

void foo();
uint32_t a=0x2f0;

int main() {
  foo();
  printf("a=0x%x\n", a);
  return 0;
}
```

```c
/* b.c */
#include <stdint.h>
#include <stdio.h>

uint8_t a;

void foo() {
  a = a + 0x50;
  printf("a=0x%x\n", a);
}
```

NE JAMAIS ÉCRIRE ÇA !!!

Compléments de C

- **Déclaration vs. définition des fonctions**
  - une fonction possède plusieurs caractéristiques :
    - un nom
    - un type de retour
    - une liste de paramètres (type)
    - un corps (la définition)
  - **déclaration** :
    - association nom ↔ (type de retour, types des paramètres)
    - exemple : `int foo(int, char);`
  - **définition** :
    - déclaration + code du corps de la fonction
    - exemple : `int foo(int a, char b) { return a+b; }`

Déclaration et définition des fonctions

- le compilateur peut générer du code utilisant une fonction même si sa définition et/ou son adresse ne sont pas encore connues
  - il n'a besoin que de savoir
    - quels arguments lui passer
    - comment récupérer la valeur de retour
    - bref, connaître seulement sa déclaration et les conventions d'appel (ABI)
  - l'adresse à laquelle sauter pour exécuter la définition de la fonction est prise temporairement nulle
  - et sera rectifiée à l'édition de lien

- conséquence :
  - une fonction ne doit être **définie qu'une seule fois**, dans un fichier source **.c**

  - une fonction **exportée** doit **en plus** :
    - **être déclarée** dans le header .h correspondant au fichier où elle est instanciée
    - ce header sera **inclus par tous les .c utilisant cette fonction**

- **Déclaration et instanciation des fonctions**
  - Exemple :

```
/* a.h */
int foo(char, char);
int bar(char a, char b);
```

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

int foo(char a, char b)
{
  return a+b;
}

int bar(char c, char d)
{
  return c+d;
}
```

```
/* b.c */
#include "a.h"

void baz()
{
    ...
    x = foo(a, b) + bar(e, f);
    ...
}
```

- **Déclaration et instanciation des fonctions**
  - on souhaite parfois définir un objet par défaut que l'utilisateur a la possibilité de remplacer s'il le souhaite
    - exemple : handler d'interruption par défaut
  - mais on ne peut pas avoir deux définitions conflictuelles
    - pour les variables ça fait des choses monstrueuses
    - pour les fonctions c'est interdit et ça déclenche une erreur
  - solution : symboles `weak`
    - exemple : `__attribute__((weak)) int a = 3;`
    - exemple : `int __attribute__((weak)) foo(int x);`

- **Visibilité des objets**
  - en C, les objets (fonctions et variables) globaux sont **exportés** par défaut
  - pour les rendre privés à un fichier : `static`
  - exemples :
    - `static int a;`
    - `static void foo() { … }`

- **Fonctions `inline`**
  - pour optimiser la vitesse d'exécution
    - évite les prologues / épilogues / appels
    - permet des optimisations entre appelant et appelé (arguments constants, …)
  - pas toujours possible
  - en C99 :
    - le plus simple : `static inline void foo() {…}`
    - nécessite au moins `-O1` et dépend de la taille de la fonction
    - forcer un inline : `__attribute__((always_inline))`
    - empêcher un inline : `__attribute__ ((noinline))`
  - Les fonctions `static inline` seront dans cette UE les **seules** fonctions à pouvoir être définies dans des headers.

7ème commandement de l'UE

- Les objets globaux ne seront définis / instanciés
  - qu'une seule fois
    - seule exception : symboles weak
  - dans des fichiers sources (.c) uniquement
    - seule exception : fonctions `static inline`

- Un objet global privé :
  - sera défini `static`
  - et ne sera donc **pas** déclaré dans un header (`.h`)

- Un objet global exporté :
  - sera déclaré dans le header `.h` associé au fichier source (.c) où il est instancié
  - qui sera inclus par **tous** les `.c` utilisant cet objet

7ème commandement de l'UE (suite)

- Conséquence, un header ne comporte **que** :
  - des déclarations d'objets globaux exportés
  - des fonctions `static inline`
  - des déclarations de types
  - des macros et `enum`

- **RIEN D'AUTRE / AUCUNE INSTANCIATION**

- **Variables locales**
  - `auto` : sont allouées / désallouées automatiquement (généralement sur la pile)
  - `static` : sont allouées et initialisées au lancement du programme et durent pendant toute la durée du programme
  - par défaut, les variables locales sont de type `auto`

Chaîne de compilation / GCC

- **Exemple : GCC**
  - `gcc -Ox` : optimisations
    - `-Os` : taille
    - `-O0` : aucune optimisation
    - `-O1 ... -O999` : de plus en plus d'optimisations
    - attention, à partir de `-O4` inclus, ça devient expérimental…
    - `-Og` : permet un débug plus simple
  - `gcc -g` pour inclure des infos de débug
    - influence sur la vitesse d'exécution du programme ?

8ème commandement de l'UE

- Il n'y a **jamais** de bonnes raisons de compiler en `-O0`
- On compilera toujours en
  - `-O1` ou `-Og` : code simple à lire
  - `-O2` : standard, mais code plus compliqué à suivre

Chaîne de compilation / GCC

- **Les warnings**
  - *warning* : quand le compilateur ne peut pas décider de lui même si un morceau de code est valide ou non
    - **pas là pour faire joli**
    - implique obligatoirement une intervention humaine
    - la plupart des projets sérieux obligent une compilation sans warnings
  - flags :
    - `gcc -Wall` : active tous les warnings facilement corrigeables
    - `gcc -Wextra` : active des warnings supplémentaires
    - `gcc -Werror` : traite les warnings comme des erreurs

9ème commandement de l'UE

- Toutes les compilations se feront avec `-Wall -Wextra -Werror`

Où en est-on ?

- **On a vu**
  - quelques rappels importants de C
  - les chaînes de compilation
  - comment écrire du C propre
- **On va voir maintenant**
  - ce que fait un éditeur de lien
- **On verra plus tard**
  - comment sont architecturés les exécutables
  - comment piloter l'éditeur de lien pour produire l'exécutable qu'on veut.

Édition de lien

- **Principe**
  - les fichiers objets
    - peuvent contenir des références vers des symboles externes : les adresses de ces symboles sont temporairement nulles
    - ont un code logé temporairement à l'adresse 0
  - le linker se chargera de :
    - allouer à tous les symboles une adresse finale
    - remplacer les références externes (adresses temporaires nulles) par les adresses finales
  - pour cela, il va directement patcher le code, en s'aidant d'informations indiquant comment procéder
    - ces informations sont appelées "informations de relocation"
    - les symboles disposant de ces informations de relocation sont dit "relogeables"

Édition de lien

- **Exemple**

```c
#include <stdio.h>

const char *mesg = "Hello World!";

int main() {
  printf(mesg);
  return 0;
}
```

```
// objdump -x

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000c R_ARM_CALL        printf
0000001c R_ARM_ABS32       mesg
```

Avant link

```
00000000 <main>:
   0:   e92d4008    push    {r3, lr}
   4:   e59f3010    ldr     r3, [pc, #16] ; 1c <main+0x1c>
   8:   e5930000    ldr     r0, [r3]
   c:   ebfffffe    bl      0 <printf>
  10:   e3a00000    mov     r0, #0
  14:   e8bd4008    pop     {r3, lr}
  18:   e12fff1e    bx      lr
  1c:   00000000    .word   0x00000000
```

Après link

```
0000821c <main>:
821c: e92d4008 push    {r3, lr}
8220: e59f3010 ldr     r3, [pc, #16] ; 8238 <main+0x1c>
8224: e5930000 ldr     r0, [r3]
8228: eb000075 bl      8404 <printf>
822c: e3a00000 mov     r0, #0
8230: e8bd4008 pop     {r3, lr}
8234: e12fff1e bx      lr
8238: 0001a624 .word   0x0001a624
```

Édition de lien

- **Avec GCC**
  - `gcc -l` pour lier des bibliothèques
  - `gcc -L` spécifie le chemin de recherche des bibliothèques
    - l'ordre a de l'importance ! Une bibliothèque complète les objets précédents.
    - il est parfois nécessaire d'utiliser plusieurs fois `-l`.
  - `gcc -nostartfiles` pour ne pas lier les fichiers de démarrage
  - `gcc -nodefaultlibs` pour ne pas lier les bibliothèques standard
  - `gcc -nostdlib` pour combiner les deux
  - dans ces cas là, on veut quand même souvent `libgcc` :
    - `gcc -lgcc`
    - `gcc -nostdlib <files>... `gcc -print-libgcc-file-name``
  - `gcc –Wl,` pour passer des options au linker

Édition de lien

- **Linker GCC : outil natif**
  - on peut invoquer directement `ld`
    - permet de simplifier le passage d'options
    - exemple : `--start-group / --end-group` pour références circulaires
  - fichiers d'entrée :
    - `*.o`
    - `*.a`
    - `ldscript`
  - par défaut, le script de link et les `crt` sont implicites
  - le script de link spécifie comment assembler les différents fichiers objets et résoudre les adresses pour produire le fichier final.

Édition de lien

- **Linker GCC : optimisations**
  - L'éditeur de liens peut faire des optimisations a posteriori :
    - Le fichier objet stocke le GIMPLE des fonctions compilées.
    - L'éditeur de lien a alors une vision d'ensemble du programme, comme si tout avait été compilé d'un coup, et peut optimiser le programme globalement.
    - Voir https://gcc.gnu.org/onlinedocs/gccint/LTO-Overview.html pour plus de détails.
  - `gcc -flto`

Où en est-on ?

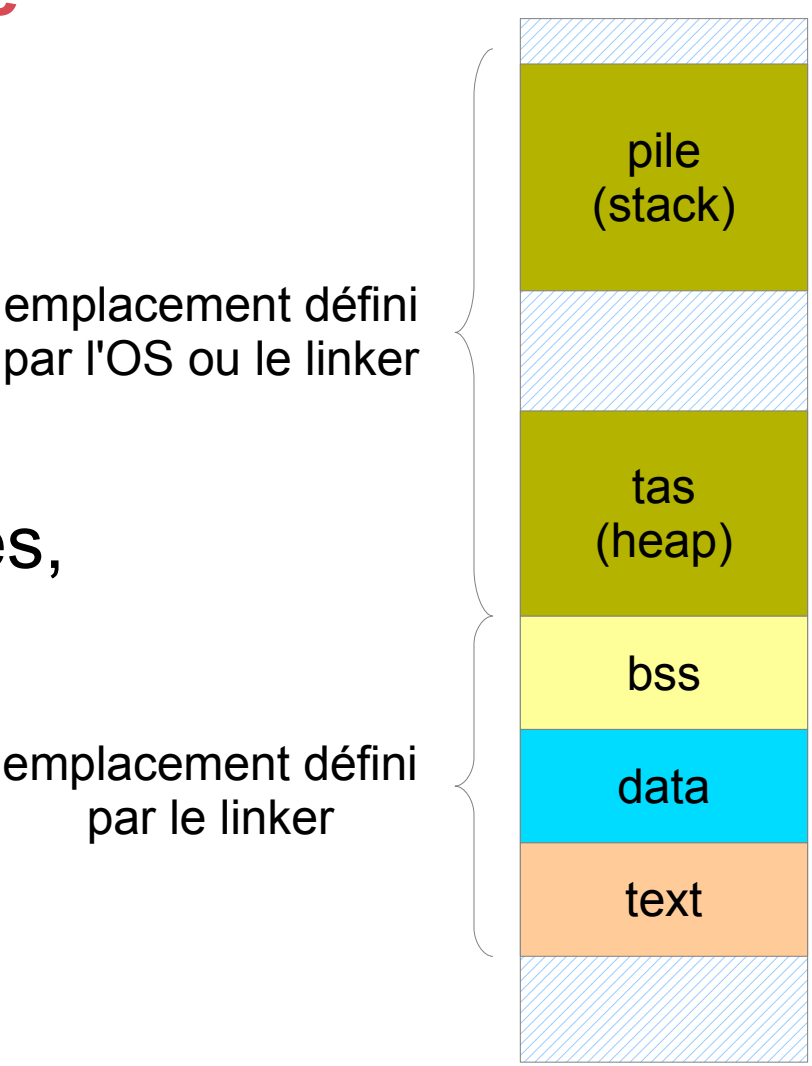
- **On sait maintenant :**
  - écrire du C propre
  - ce que fait un compilateur
  - à quoi sert un éditeur de lien
- **On va voir**
  - comment sont architecturés les exécutables
  - comment piloter l'éditeur de lien pour produire l'exécutable qu'on veut.

Anatomie d'un exécutable

- **Composantes d'un exécutable**
  - le code
  - les données
  - nécessité de les séparer :
    - recopies ROM – RAM
    - instances multiples de programmes
    - instances multiples de bibliothèques
    - ...
  - découpage en segments / sections

Anatomie d'un exécutable

- **Sections : aperçu**
  - text : le code
  - data : les données initialisées
  - rodata : les constantes
  - bss : les données non initialisées, ou initialisées à zéro
  - Très rarement :
    - stack : pile
    - heap : allocation dynamique (malloc)



Anatomie d'un exécutable

- **Format des exécutables**
  - les exécutables sont destinés à être exécutés
    - soit directement
    - soit à travers un loader
  - les loaders ont besoin d'informations auxiliaires pour charger un programme en mémoire
  - les images mémoire (directement exécutables) n'en ont pas besoin.
  - selon leur destination, les exécutables n'ont pas à fournir les même informations → différents formats !

Anatomie d'un exécutable

- **Tant qu'on y est...**
  - la compilation fait généralement intervenir plusieurs types de fichiers :
    - des objets,
    - des bibliothèques,
    - …
  - les objets :
    - contiennent du code
    - ainsi que beaucoup d'informations auxiliaires (sections, infos de débug, infos de relocation, …)
  - Il serait bien d'avoir un même format pour les exécutables et les objets

Anatomie d'un exécutable

- **Format d'exécutable : image mémoire binaire**
  - = dump direct du contenu de la flash
  - seul "format" pouvant assurer un boot
  - aucune information auxiliaire
  - facilement flashable
  - difficile à désassembler / examiner
  - variantes disponibles :
    - S-REC
    - Intel HEX

Anatomie d'un exécutable

- **Format d'exécutable : a.out**
  - plusieurs variantes
  - organise les choses en segments :
    - exec : infos sur les autres segments
    - text : le code / constantes
    - data : variables initialisées
    - text relocations
    - data relocations
    - symbol table
    - string table
  - abandonné pour COFF puis ELF / PE
  - premier format à laisser une page vide au début de la mémoire virtuelle : pour quoi faire ?

Anatomie d'un exécutable

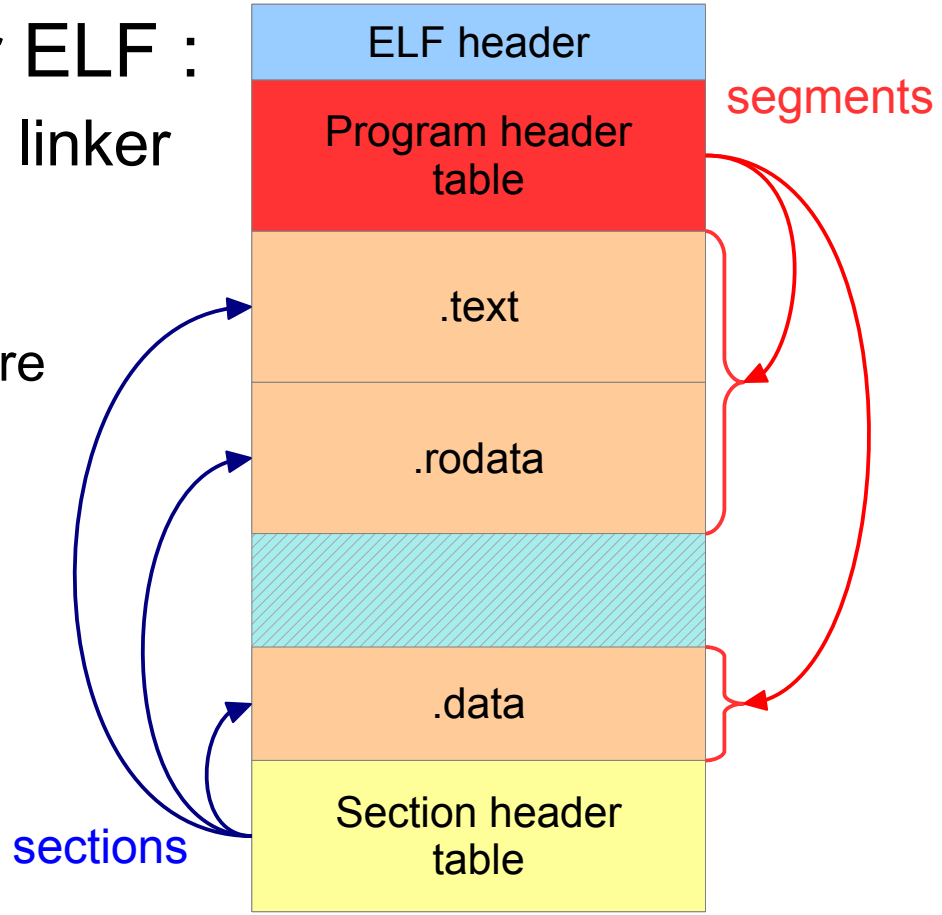
- **Format d'exécutable : COFF**
  - base de PE
  - partage le code en sections
  - améliorations de a.out
    - informations de débug plus complètes (mais pas suffisantes pour C++)
    - adresses virtuelles relatives : les adresses sont des déplacements par rapport à une adresse de base globale au fichier.
  - a été remplacé par PE (Windows) et ELF (reste du monde)

Anatomie d'un exécutable

- **Format d'exécutable : ELF**
  - Executable and Linkable Format
  - quatre types d'objets :
    - objets (`*.o`) :
      - créés par l'assembleur
      - contiennent des symboles non résolus / du code relogeable temporairement stocké à l'adresse 0
      - doivent passer par le linker avant de pouvoir être exécutés.
    - exécutables :
      - ont tous leurs symboles résolus (sauf bib. dynamiques)
      - ont toutes les relocations faites
      - destinés à un loader ELF
    - bibliothèques partagées (`*.so`)
      - contiennent des informations sur les symboles (pour le linker)
      - et du code et des informations d'exécution (pour le loader)
    - core file :
      - core dump
  - format universellement adopté dans le monde Unix et dans l'embarqué (seule exception : Windows...)

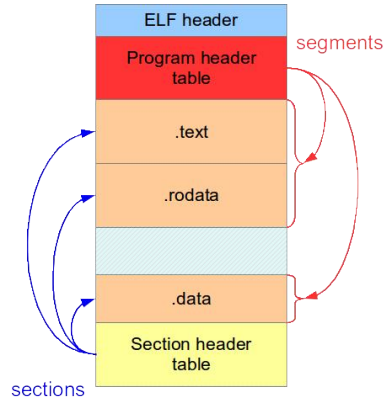
Anatomie d'un exécutable

- **Format d'exécutable : ELF**
  - deux vues d'un même fichier ELF :
    - liste de sections : destinées au linker
    - liste de segments :
      - pour le loader ELF de Linux
      - destinés à être mappés en mémoire
      - composés de sections



	Program header	Section header
objets relogeables	optionnel	obligatoire
executables	obligatoire	optionnel
bibliothèques partagées	obligatoire	obligatoire

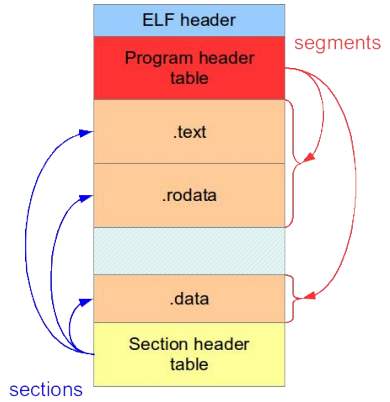
Anatomie d'un exécutable



- **Format d'exécutable : ELF**
  - header ELF

```
typedef struct{
    unsigned char magic[4]; // magic number "\0x7fELF..."
    char class;             // address size, 1=32 bits, 2=64 bits
    char byteorder;         // 1 = little-endian, 2 = big-endian
    char hversion;          // header version, always 1
    char pad[9];
    Elf32_Half e_type       // file type : 1 = relocatable, 2 = executable,
                            //             3 = shared object, 4 = core file, ...
    Elf32_Half e_machine;   // 2 = SPARC, 3 = x86, 4 = 68k, 8 = MIPS, etc...
    Elf32_Word e_version;   // always 1
    Elf32_Addr e_entry;     // entry point if avalaible
    Elf32_Off e_phoff;      // file position of program header or 0
    Elf32_Off e_shoff;      // file position of section header or 0
    Elf32_Word e_flags;     // architecture specific flags, usually 0
    Elf32_Half e_ehsize;    // size of this ELF header
    Elf32_Half e_phentsize; // size of an entry in program header
    Elf32_Half e_phnum;     // number of entries in program header or 0
    Elf32_Half e_shentsize; // size of an entry in section header
    Elf32_Half e_shnum;     // number of entries in section header or 0
    Elf32_Half e_shstrndx;  // section number that contains section name strings
}Elf32_Ehdr;
```

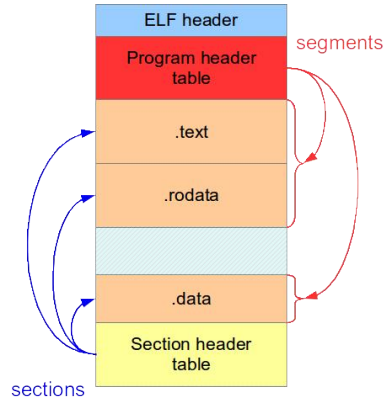
Anatomie d'un exécutable



- **ELF : les sections**
  - Section headers

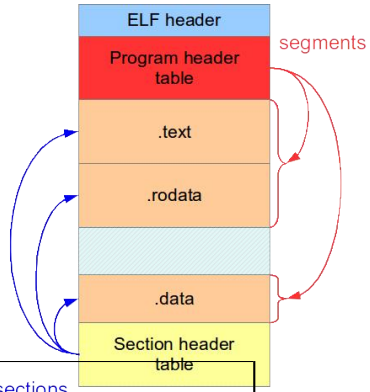
```
typedef struct{
    Elf32_Word sh_name;      // name, index into the string table
    Elf32_Word sh_type;      // section type (PROGBITS, NOBITS, SYMTAB, ...)
    Elf32_Word sh_flags;     // flag bits (ALLOC, WRITE, EXECINSTR)
    Elf32_Word sh_addr;      // base memory address(VMA), if loadable, or zero
    Elf32_off  sh_offset;    // file position of beginning of section
    Elf32_Word sh_size;      // size in bytes
    Elf32_Word sh_link;      // section number with related info or zero
    Elf32_Word sh_info;      // more section-specific info
    Elf32_Word sh_align;     // alignment granularity if section is moved
    Elf32_Word sh_entsize; // size of entries if section is an array
} Elf32_Shdr;
```

Anatomie d'un exécutable



- **ELF : les sections**
  - type :
    - `PROGBITS` : section contenant du code, des data, des infos de debug et qui occupe de la place dans le fichier ELF
    - `NOBITS` : idem, mais sans occuper de place (typiquement, bss)
    - `SYMTAB / DYNSYM` : table des symboles
    - `STRTAB` : table des noms de symboles
    - `REL / RELA` : informations de relocation
    - `DYNAMIC / HASH` : informations pour link dynamique
  - flags
    - `WRITE` : les données sont modifiables lors de l'exécution
    - `ALLOC` : la section occupe de la mémoire lors de l'exécution
    - `EXECINSTR` : la section contient du code exécutable

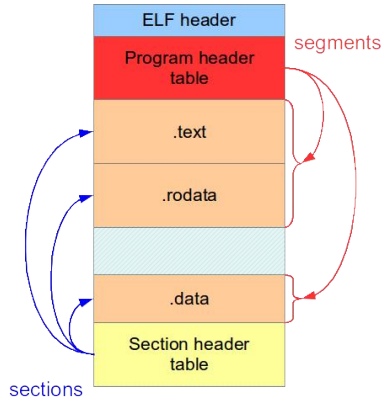
Anatomie d'un exécutable



ELF : les sections

Nom	Type	Flags	Usage
.text	PROGBITS	A, EX	Code de l'exécutable
.rodata .rodata1 .sdata2	PROGBITS	A	Données non modifiables
.data .data1 .sdata	PROGBITS	A, W	Données modifiables initialisées non nulles
.bss .sbss	NOBITS	A, W	Données non initialisées (ou initialisées à zéro). Le système initialise cette zone au lancement de l'exécution avec des zéros. Peut aussi servir à la pile et au tas
.heap		A, W	Le tas.
.stack			La pile, généralement après .heap. Peut être combiné avec .heap pour former une section .bss_stack
.init .ini	PROGBITS	A, EX	Initialisation du processus. Le système exécute cette section avant d'appeler main. Utilisé par la libC pour initialiser des variables globales.
.fini	PROGBITS	A, EX	Terminaison du processus. Le système exécute cette section à la fin du processus si celui-ci s'est correctement terminé.
.ctors	PROGBITS	A, W	Pointeurs vers des fonctions à appeler au démarrage du programme
.dtors	PROGBITS	A, W	Pointeurs vers des fonctions à appeler à la fin du programme
.got / .got2	PROGBITS	A, W	Global Offset Table : permet de référencer les variables globales des bibliothèques dynamiques
.plt	PROGBITS	A, EX	Procedure Linkage Table : permet de référencer les fonctions des bibliothèques dynamiques (lazy binding)
rela.text rela.data rela.rodata	REL	none (sauf bib dyn)	Informations de relocations pour le code, les données et les constantes

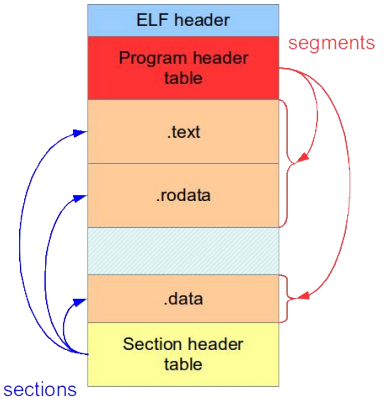
Anatomie d'un exécutable



ELF : les sections

```c
#include <stdint.h>
#include <stdio.h>

int32_t x = 34;
int32_t y;
const char mesg[] = "Hello World!";

int main() {
    static uint8_t z;
    uint16_t t;

    y = 12;
    z = z + 1;
    t = y+z;

    printf(mesg);
    printf("x = %d, y = %d, z = %d, t = %d\n",
           x, y, z, t);

    return 0;
}
```

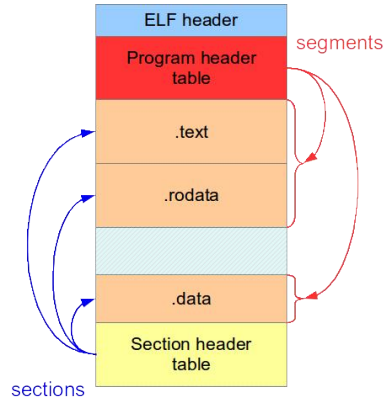
Anatomie d'un exécutable



- ELF : les sections

```
alexis@plop> arm-none-eabi-gcc -c t.c

alexis@plop> arm-none-eabi-objdump -f t.o

t.o:     file format elf32-littlearm
architecture: armv4t, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
```

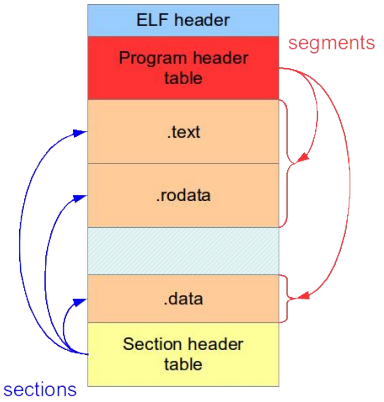
Anatomie d'un exécutable



- **ELF : les sections**

```
alexis@plop> arm-none-eabi-objdump -h t.o

t.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000b8  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  000000ec  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000001  00000000  00000000  000000f0  2**0
                  ALLOC
  3 .rodata       00000032  00000000  00000000  000000f0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000001e  00000000  00000000  00000130  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002a  00000000  00000000  0000014e  2**0
                  CONTENTS, READONLY
```

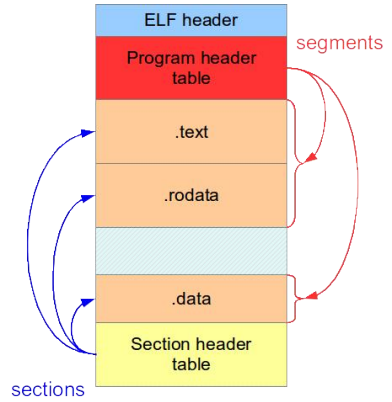
Anatomie d'un exécutable



- **Où vont les variables ?**

			.text	.rodata	.data	.bss	stack
globale	initialisée				X		
	non initialisée					X	
	const		X	X			
locale	static	initialisée			X		
		non initialisée				X	
	non static	initialisée					X
		non initialisée					X
	const		X	X			
valeur immédiate			X	X			

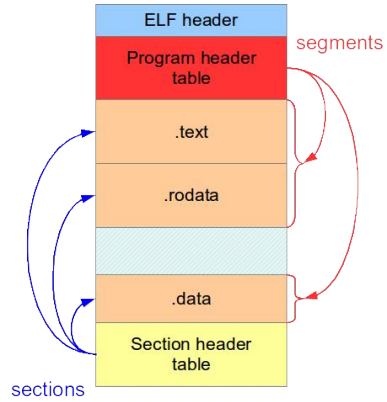
Anatomie d'un exécutable



- **Que fait le programme suivant ?**

```
#include <stdio.h>

char *p = "Jello World!\n";

int main()
{
  p[0] = 'H';
  printf("%s", p);

  return 0;
}
```

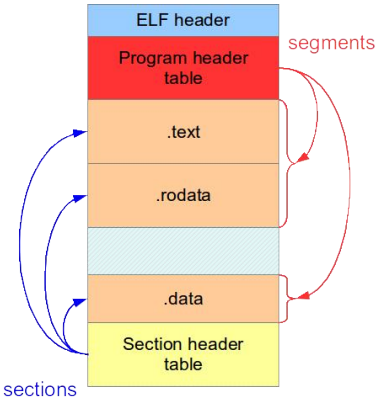
Anatomie d'un exécutable



- **ELF : les sections**

```
alexis@plop> arm-none-eabi-objdump -t t.o

t.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*      00000000 t.c
00000000 l    d  .text      00000000 .text
00000000 l    d  .data      00000000 .data
00000000 l    d  .bss       00000000 .bss
00000000 l    d  .rodata    00000000 .rodata
00000000 l       .bss       00000001 z.5152
00000000 l    d  .comment   00000000 .comment
00000000 g    O  .data      00000004 x
00000004      O  *COM*      00000004 y
00000000 g    O  .rodata    0000000d mesg
00000000 g    F  .text      000000b8 main
00000000         *UND*      00000000 printf
```

Anatomie d'un exécutable



ELF : les sections

```
alexis@plop> arm-none-eabi-objdump -t t.o

t.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*      00000000 t.c
00000000 l    d  .text      00000000 .text
00000000 l    d  .data      00000000 .data
00000000 l    d  .bss       00000000 .bss
00000000 l    d  .rodata    00000000 .rodata
00000000 l       .bss       00000001 z.5152
00000000 l    d  .comment   00000000 .comment
00000000 g    O  .data      00000004 x
00000004      O  *COM*      00000004 y
00000000 g    O  .rodata    0000000d mesg
00000000 g    F  .text      000000b8 main
00000000         *UND*      00000000 printf
```

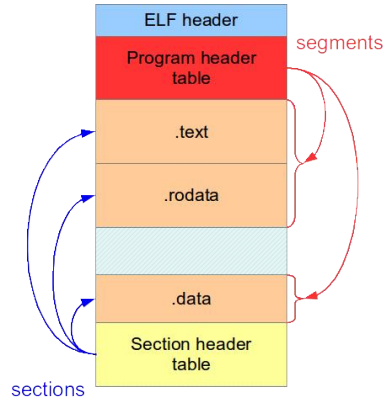
adresse

taille

local / global

section

Anatomie d'un exécutable

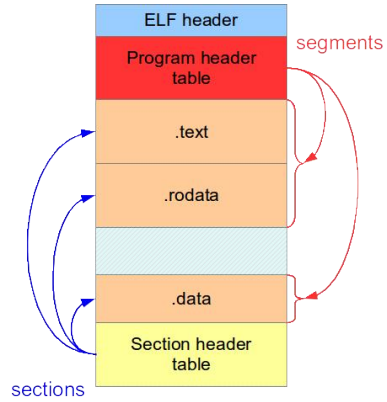


ELF : les sections

- `*UND*` : un symbole pour l'instant non défini, qui sera résolu plus tard - du moins on l'espère.

- `*COM*` : des symboles dont on ne sait pas encore s'ils seront placés dans `bss` ou dans `data` après link. Seront placés dans le `bss` après link si aucune autre unité ne les définit en les initialisant à une valeur non nulle.

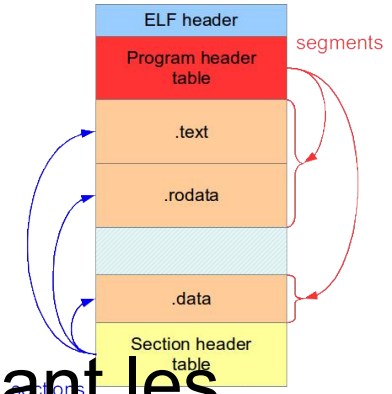
Anatomie d'un exécutable



- **ELF : les sections**
  - est-ce cohérent ?

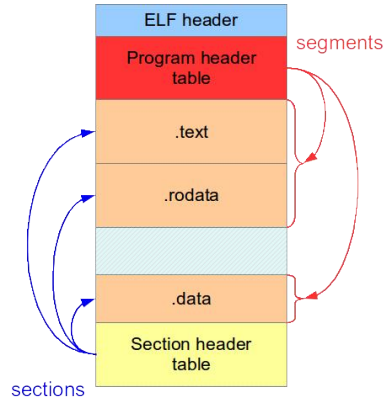
```
alexis@plop> arm-none-eabi-objdump -h t.o

t.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000b8  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  000000ec  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000001  00000000  00000000  000000f0  2**0
                  ALLOC
  3 .rodata       00000032  00000000  00000000  000000f0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000001e  00000000  00000000  00000130  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 00000030  00000000  00000000  0000014e  2**0
                  CONTENTS, READONLY
```

Anatomie d'un exécutable



- **ELF : les segments**
  - un segment est composé de sections partageant les mêmes attributs d'exécution (protection mémoire)
  - les segments sont
    - mappés en mémoire par le loader
    - ou bien transformés en images mémoire par objcopy
  - généralement peu de segments
    - un read-only exécutable pour le code
    - un read-only non exécutable pour les constantes
    - un read/write non exécutable pour les données normales
  - de façon à mapper le processus en mémoire le plus vite possible

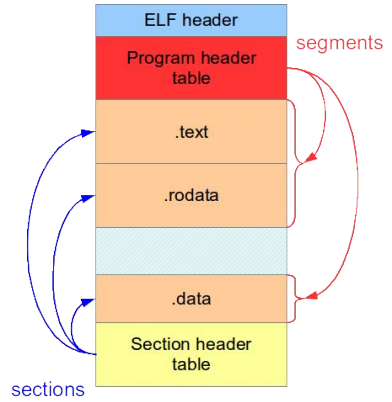
Anatomie d'un exécutable



- **ELF : les segments**
  - Program headers

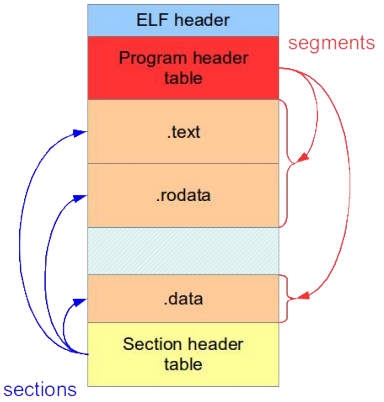
```
typedef struct{
    Elf32_Word type;        // loadable code or data, dynamic linking info, etc.
    Elf32_off  offset;      // file offset of segment
    Elf32_Addr virtaddr;    // virtual address to map segment (VMA)
    Elf32_Addr physaddr;    // physical address (LMA)
    Elf32_Word filesize;    // size of segment in file
    Elf32_Word memsize;     // size of segment in memory (bigger if contains bss)
    Elf32_Word flags;       // Read, Write, Execute bits
    Elf32_Word align;       // required alignment, invariably hardware page size
} Elf32_Phdr;
```

Anatomie d'un exécutable



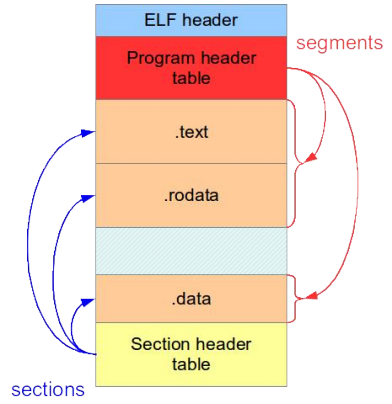
- **ELF : les segments**
  - type :
    - `PT_LOAD` : le segment est mappable en mémoire
    - `PT_DYNAMIC` : le segment est dynamiquement partageable
    - `PT_INTERP` : spécifie un interpréteur / loader
  - LMA / adresse physique :
    - adresse de stockage avant recopie
  - VMA / adresse virtuelle :
    - adresse en mémoire après recopie par crt0
  - rappel : rien à voir avec les adresse physiques / virtuelles des MMU

ELF : les segments



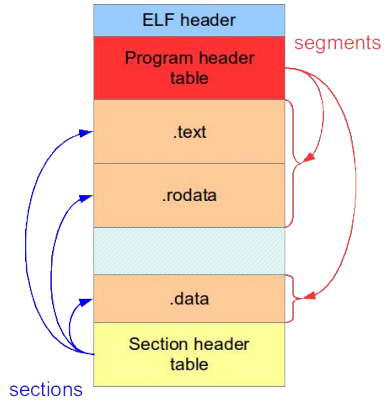

```
alexis@plop> arm-none-eabi-gcc t.o stubs.c -o t
alexis@plop> arm-none-eabi-readelf -l t

Elf file type is EXEC (Executable file)
Entry point 0x80e8
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  EXIDX          0x012488 0x00012488 0x00012488 0x00008 0x00008 R   0x4
  LOAD           0x000000 0x00000000 0x00000000 0x12494 0x12494 R E 0x10000
  LOAD           0x012494 0x00022494 0x00022494 0x009c0 0x00a20 RW  0x10000

 Section to Segment mapping:
  Segment Sections...
   00     .ARM.exidx
   01     .init .text .fini .rodata .ARM.exidx .eh_frame
   02     .init_array .fini_array .data .bss
```

Anatomie d'un exécutable



- **Outils ELF**
  - `objdump` : examen du contenu d'objets
    - `objdump -f` : infos générales
    - `objdump -h` : les section
    - `objdump -p` : les segments
    - `objdump -t` : la table des symboles
    - `objdump -x` : tous les headers
    - `objdump -d` : désassemble les sections de code
    - `objdump -S` : désassemble les sections de code en mixant avec le source C (s'il contient des informations de débug)

Anatomie d'un exécutable



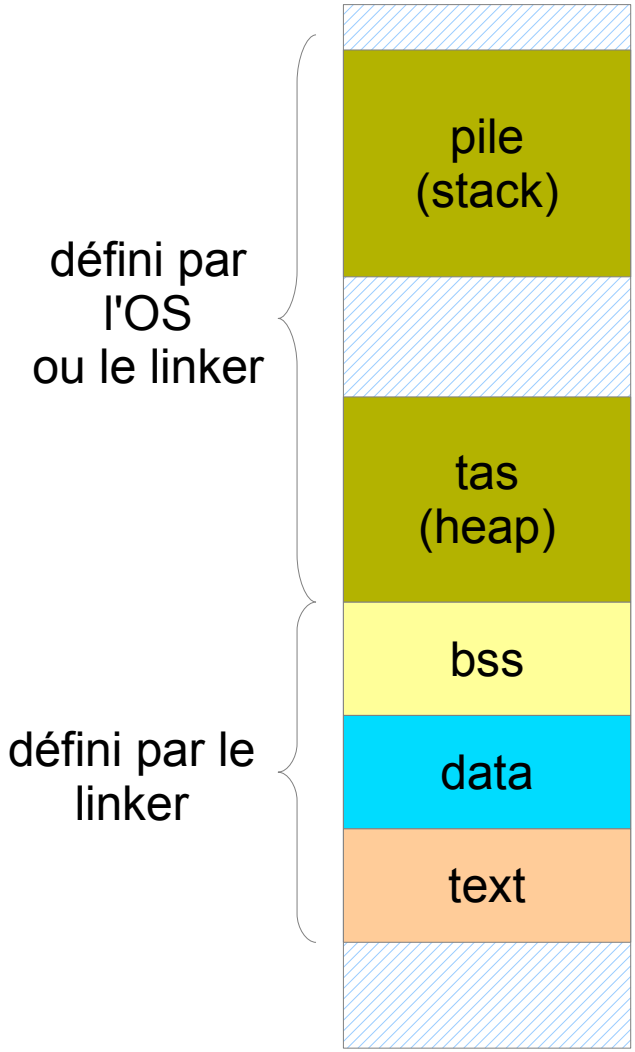
- **Outils ELF**
  - `readelf` : idem, mais que pour ELF
    - `readelf -h` : infos générales
    - `readelf -S` : les section
    - `readelf -l` : les segments
    - `readelf -s` : la table des symboles
    - `readelf -d` : les sections dynamiques
  - complémentaire d'`objdump`

Où en est-on ?

- **On sait maintenant :**
  - écrire du C propre
  - ce que fait un compilateur
  - à quoi sert un éditeur de lien
  - comment sont architecturés les exécutables
- **On va voir**
  - comment piloter l'éditeur de lien pour produire l'exécutable qu'on veut.

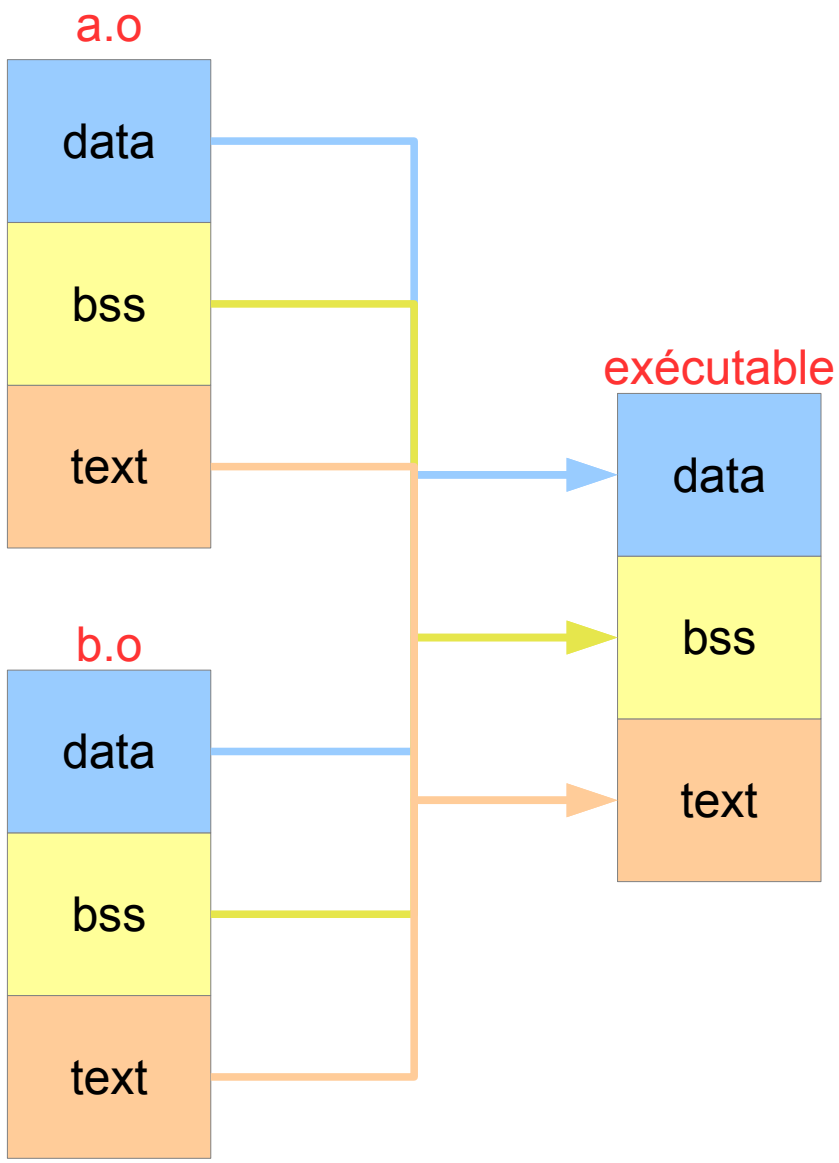
Linkers et linker scripts

- **Quel mapping pour les exécutables ?**
  - le plan d'adressage est défini par l'OS ainsi que le linker
  - quand il n'y a pas d'OS, c'est à l'exécutable lui-même de créer sa pile / le tas (si nécessaire)

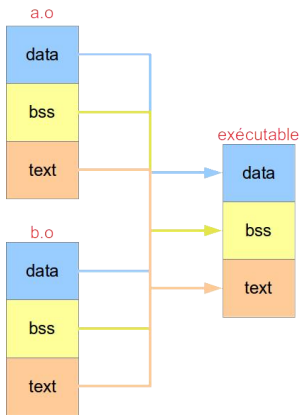


Linkers et linker scripts

- **Link**
  - le script de link définit :
    - comment sont rassemblées les sections
    - le point d'entrée
    - d'éventuels symboles additionnels
    - …
  - Les segments sont créés automatiquement après avoir rassemblé les sections.



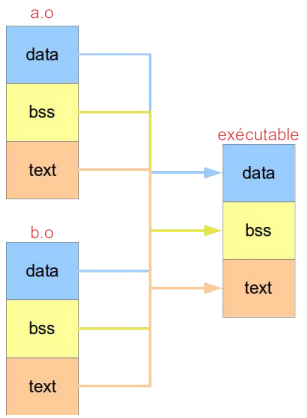
Linkers et linker scripts



- **MEMORY**
  - permet de décrire le layout de la mémoire. Optionnel.
  - chaque entrée est appelée « région »
  - exemple :

```
MEMORY
        {
          rom (rx) : ORIGIN = 0, LENGTH = 256K
          ram (wx) : ORIGIN = 0x40000, LENGTH = 4M
        }
```

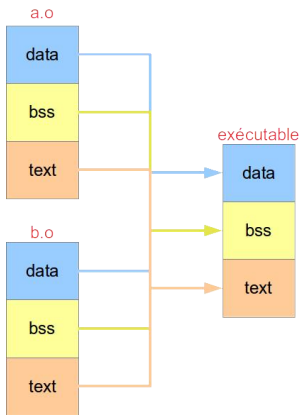
Linkers et linker scripts



- **SECTION**
  - permet de décrire comment fusionner les sections des objets d'entrée :

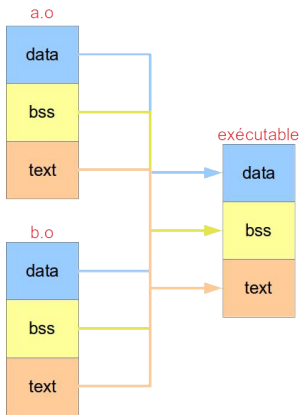
```
section [VMA_address] : [AT(LMA_address)[ALIGN(section_align)]
        {
          output-section-command
          output-section-command
          ...
        } [>VMA_region] [AT>LMA_region]
```

Linkers et linker scripts



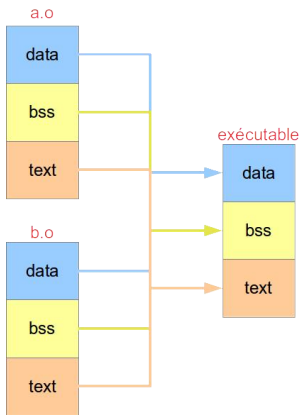

```
// LDSCRIPT

SECTIONS
 {
   .text : {
        *(.text)
    } > rom

   .rodata : AT (ADDR(.text) + SIZEOF(.text)){
        *(.rodata)
    }

   .data 0x2000 : {
        *(.data)
    } AT> rom

   .bss 0x3000 : {
        *(.bss) *(COMMON)
    }
 }
```

Linkers et linker scripts



- **Commandes :**
  - `ENTRY(symbol)` :
    - définit "`symbol`" comme étant le point d'entrée du programme
    - usuellement : `_start`
  - symboles et affectations :
    - crée un symbole et lui donne une valeur
    - exemple : `etext = 0x1000;`
  - les symboles sont accessibles depuis le C, mais :
    - ils ne contiennent pas de valeur
    - ils ont seulement une adresse !!!

Linkers et linker scripts

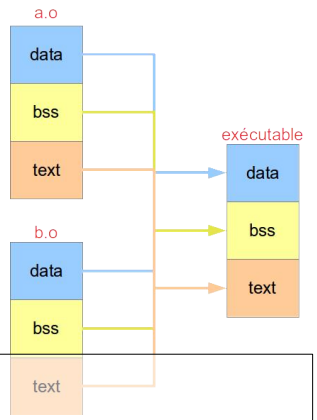


- **Exemple :**

```
// LDSCRIPT
start_of_RAM    = 0x1000;
start_of_TEXT   = .text;
sizeof_of_TEXT = sizeof (.text);
```

```
// init.c
extern uint8_t start_of_RAM, end_of_TEXT,
               sizeof_of_TEXT;

memcpy (&start_of_RAM, &start_of_TEXT, &sizeof_of_TEXT);
```

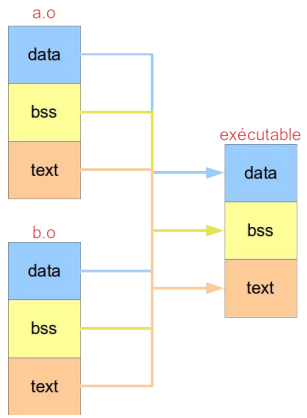
Linkers et linker scripts



-
  - location counter
  - modification :
    - à l'intérieur d'une section : offset par rapport au début de la section
    - à l'extérieur d'une section : adresse (VMA) absolue
  - affecté à un symbole : symbole = VMA(.)
  - ne peut jamais revenir en arrière

```
SECTIONS
      {
            . = 0x100;
            .text: {
                *(.text)
                . = 0x200;
            }
            . = 0x500
            .data: {
                *(.data)
                . += 0x600;
            }
      }
```

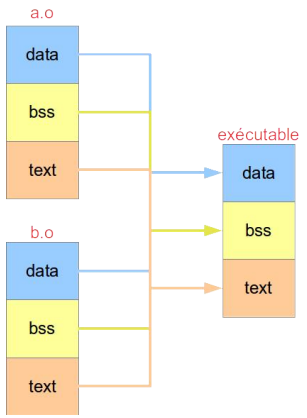
Linkers et linker scripts




```
// LDSCRIPT

MEMORY {
  rom (rx) : ORIGIN = 0x00000000, LENGTH = 256K
  ram (!rx): ORIGIN = 0x00200000, LENGTH = 1M
}

SECTIONS {
   .text : {
         *(.text)
         _etext = . ;
    } > rom

   .data : AT (ADDR(.text)+SIZEOF(.text)) {
         _data = . ;
         *(.data)
         _edata = . ;
    } > ram

   .bss : {
         _bstart = . ;
         *(.bss) *(COMMON)
         _bend = . ;
    } > ram
}
```

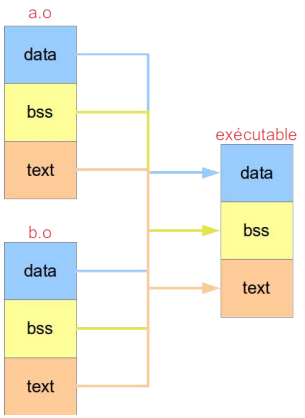
```
// init.c

extern char _etext, _data,
            _edata, _bstart, _bend;

uint8_t *src = &_etext;

uint8_t *dst = &_data;


// ROM has placed .data at end of .text.
// Copy .data to RAM
while (dst < &_edata)
    *dst++ = *src++;


// Zero out bss
for (dst = &_bstart; dst < &_bend; dst++)
    *dst = 0;
```

Linkers et linker scripts



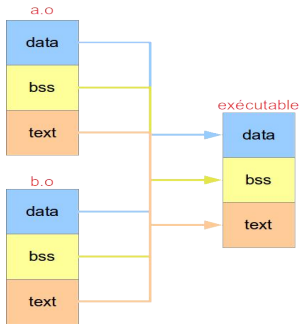
- **Autres commandes pratiques:**
  - `ORIGIN(region)` :
    - renvoie l'adresse de début d'une région
  - `LENGTH(region)` :
    - renvoie la longueur d'une région
  - `ADDR(section)` :
    - renvoie la VMA d'une section
  - `LOADADDR(section)` :
    - renvoie la LMA d'une section
  - `SIZEOF(section)` :
    - renvoie la taille d'une section

Derniers détails



- **crt0.s**
  - en charge de préparer l'espace d'exécution
    - prépare la pile et positionne le pointeur de pile
    - met le bss à zéro
    - recopie les données de la ROM vers la RAM
    - en utilisant les symboles exportés par le linker
  - si on est sûr que data et/ou bss sont vides, ce n'est pas la peine de les initialiser
  - symboles spéciaux : `etext`, `edata`, `end`

Derniers détails

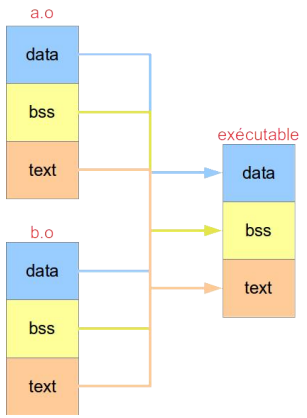


- **Conversion de formats**
  - il est préférable de garder le plus de détails sur un exécutable, le plus longtemps possible : travailler en ELF
  - mais un fichier ELF n'est pas bootable tel quel (pas mappable en mémoire sans un loader)
  - il faut d'abord le transformer en une image mémoire avant de le flasher comme code de boot dans une ROM

  - `objcopy -O binary bootloader.elf bootloader.bin`

  - manipule les sections selon leur LMA lors de la génération d'une image binaire
  - peut modifier des sections au passage :
    - strip
    - relocations

Derniers détails



- **Conversion de formats**
  - inversement, il est possible de transformer un fichier binaire quelconque en une section ELF (`.data` par défaut) :

```
objcopy -I binary -O elf64-little -B i386 input_file output_file
```

  - `objcopy -B` crée aussi trois symboles :
    - `_binary_input_file_start`
    - `_binary_input_file_end`
    - `_binary_input_file_size`

Cooool, encore des exercices !

- **À vous de travailler :**
  - exercice 1 : facile
  - exercice 2 : moyen
  - exercice 3 : compliqué

Exercices

- **Exercice 1**
  - Slide 96 : pour y, l'adresse est "4", ce qui est très bizarre !

    En vous aidant du manuel d'objdump et en expérimentant avec, pour y, des types de tailles différentes (plus et moins grandes), corrigez le slide. On compilera pour ARM.

    N'oubliez pas de fournir le code des expérimentations que vous avez faites.

Exercices

- **Exercice 2 :**

  1) Sur un PC Linux x86_64 actuel et par adresses croissantes, dans quel ordre sont stockées les sections suivantes : `text`, `data*`, `rodata*`, `bss`, pile et tas ?

  2) Dans quel sens croît la pile ?

  Remarque : ce n'est pas le résultat qui m'intéresse (je le connais déjà), mais la façon dont vous y êtes arrivé. Justifiez-donc en détail vos réponses.

Exercices

- **Exercice 3 :**

  1) Compilez sans édition de lien ce code-ci (http://bit.ly/2ApXoDl) pour ARM avec une chaîne récente, et avec les optimisations suivantes : `Os`, `O0`, `O1` et `O2`.
  Pour chaque niveau d'optimisation, justifiez la taille des sections de données que vous obtenez.

  2) Remplacez `const char mesg[]` par `static const char mesg[]`. Expliquez les différences dans les sections de données par rapport à la question précédente (elles dépendent ici aussi des optimisations).

  3) Remplacez `const char mesg[]` par `const char *mesg`. puis par `const char * const mesg`. Expliquez les différences dans le code généré et les sections de données par rapport à la question 2.

Exercices

- **Exercice 3 : indications**
  - Cet exercice est destiné à vous faire manipuler `gcc` et `objdump`. Il a l'air simple, mais c'est un véritable jeu de piste, complexe, et qui prend du temps. Vous devrez avancer pas à pas, compiler, examiner à la loupe les différentes sections, tester, recommencer. N'hésitez pas à poser des questions au fur et à mesure par mail.

  - Si vous devez faire une édition de lien pour produire un exécutable, il vous faudra fournir certaines fonctions (une version bidon suffira) : vous pouvez soit les écrire vous même, soit les trouver ici : http://bit.ly/2Bv1TMr

Exercices

- Pour ceux qui sont perdus :
  - Commencez par `-O0`, et examinez le contenu de .rodata (`objdump -s`). Pourquoi contient-elle deux fois la même chaîne ? D'ailleurs est-ce bien la même chaîne ?
  - Regardez le code généré (`objdump` toujours, à vous de trouver la bonne option) : quelle chaîne est utilisée ? À quelle fonction est-elle passée ? Pourquoi n'est-ce pas la même fonction que celle spécifiée dans le code C ? Quel est l'intérêt ? (le `man` de cette fonction pourra vous aider). La suite va permettre de comprendre à quoi sert l'autre chaîne.
  - Compilez en -O1. Regardez les sections de données et leur contenu. Que remarquez-vous ? Une recherche sur google du nom de la section de données nouvellement apparue vous indiquera son utilité. Avec une édition de lien, vous trouverez quelle chaîne est véritablement utilisée. À vous de faire la suite.

    Attention : stackoverflow n'est **pas** votre ami !

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr