

05B : Construire et inspecter un programme C

420-3A6-ST : Programmation système

Dany Gagnon

Cégep de Saint-Jean-sur-Richelieu

Lundi 28 septembre 2026

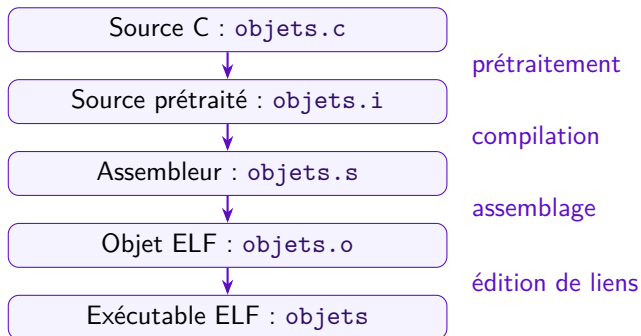
Le problème à résoudre

Comment passer de plusieurs sources à un programme que l'on peut reconstruire, tester et inspecter ?

À la fin de la séance

- Relier sources, objets, en-tête et exécutable.
- Lire des octets en hexadécimal et expliquer leur interprétation.
- Exprimer les dépendances dans Make.
- Repérer une erreur logique avec des cas de test.
- Distinguer en-tête, sections et segments d'un ELF.

Cycle de vie : du C à l'exécutable



Arrêter GCC à chaque étape

Dans `exemples/construction/elf` :

```
gcc -E objets.c -o objets.i
gcc -S objets.i -o objets.s
gcc -c objets.s -o objets.o
gcc objets.o -o objets
./objets
```

- `-E` développe les inclusions et les macros.
- `-S` produit du texte assembleur ; `-c` s'arrête au fichier objet.
- Sans ces options d'arrêt, GCC peut enchaîner les étapes jusqu'à la liaison.

Un bit, un octet, une écriture hexadécimale

- Un **bit** vaut 0 ou 1.
- Sur nos machines, un **octet** contient 8 bits : 256 valeurs, de 0 à 255.
- Un chiffre hexadécimal représente 4 bits ; deux chiffres représentent un octet.

Binaire	Hexadécimal	Décimal
0000 0000	00	0
0100 0001	41	65
1111 1111	ff	255

0x41 est une notation pour la valeur 65, pas une autre donnée.

Les mêmes octets peuvent représenter du texte ou un nombre

```
printf 'AB\n' | od -An -tx1  
# 41 42 0a
```

- En ASCII, 41 représente A ; 42, B ; 0a, le saut de ligne.
- Interprété comme entier non signé, l'octet 41 vaut 65.
- Le texte 65 contient deux octets ASCII : 36 35.

Le sens vient du format et du programme qui lit les octets. Un fichier ELF contient à la fois du code, des données et des métadonnées.

Boutisme : dans quel ordre ranger plusieurs octets ?

La valeur entière sur 32 bits `0xDEADBEEF` occupe quatre octets.

Adresses croissantes	A	A+1	A+2	A+3
Gros boutisme	DE	AD	BE	EF
Petit boutisme	EF	BE	AD	DE

Sur x86-64, les entiers sont stockés en petit boutisme. On inverse l'ordre des **octets**, pas les bits de chaque octet. Une chaîne de caractères conserve l'ordre de ses caractères.

Code machine et assembleur décrivent les mêmes instructions

```
int incrementer(int valeur) { return valeur + 1; }
```

Extrait de sortie sur x86-64 avec GCC :

```
83 c0 01    add eax,0x1
5d          pop rbp
c3          ret
```

- À gauche : les octets qui codent les instructions.
- À droite : leur représentation en assembleur, syntaxe Intel.
- Ici, `add` ajoute 1 ; `pop` restaure un registre ; `ret` revient à l'appelant.

La sortie dépend de l'architecture et des options. Il n'y a pas forcément une instruction machine par ligne C.

Observer sans retrouver exactement le source

```
gcc -c instruction.c -o instruction.o  
objdump -d -M intel instruction.o  
nm instruction.o
```

- Le désassemblage traduit les octets d'instructions en texte assembleur.
- Les commentaires ne sont pas conservés dans ce code machine.
- Les optimisations peuvent supprimer ou réorganiser des calculs.
- `-g` ajoute des informations de débogage ; `nm` affiche les symboles disponibles.

Une modification, deux étapes de construction

```
main.c ----> main.o ----\  
                                > somme  
calcul.c --> calcul.o --/
```

La compilation produit un objet par source. L'édition de liens rassemble les objets et résout les noms de fonctions.

Le fichier d'en-tête expose le contrat

```
/* calcul.h */  
#ifndef CALCUL_H  
#define CALCUL_H  
int somme_jusqua(int limite);  
#endif
```

Les deux sources incluent cet en-tête. La déclaration permet de vérifier les appels ; la définition de la fonction reste dans `calcul.c`.

Construire une fois à la main

```
gcc -Wall -Wextra -Wpedantic -g -c main.c  
gcc -Wall -Wextra -Wpedantic -g -c calcul.c  
gcc main.o calcul.o -o somme  
./somme
```

Si la liaison annonce un symbole absent, chercher quel objet fournit sa définition.

Make décrit les dépendances

```
somme: main.o calcul.o
    gcc main.o calcul.o -o somme

main.o: main.c calcul.h
    gcc -Wall -Wextra -g -c main.c

calcul.o: calcul.c calcul.h
    gcc -Wall -Wextra -g -c calcul.c
```

Une recette commence par une tabulation. Une modification de l'en-tête doit reconstruire les deux objets, puis l'exécutable.

Observer ce qui est réellement reconstruit

```
make  
make  
touch calcul.c  
make  
touch calcul.h  
make
```

Prévoir les commandes lancées à chaque étape. Si un fichier attendu n'est pas reconstruit, relire ses dépendances.

Les options aident à trouver une cause

- `-Wall -Wextra -Wpedantic` signale des usages suspects.
- Le premier message du compilateur aide à retrouver la ligne fautive.
- Un test compare un résultat observé à une valeur prévue.
- Une compilation réussie ne prouve pas que le calcul est juste.

Un cas normal et deux frontières

Le programme doit calculer la somme de 1 jusqu'à la limite incluse.

Limite	Résultat prévu	Ce que le test vérifie
0	0	aucun terme à ajouter
1	1	premier terme inclus
5	15	dernier terme inclus

Une compilation sans avertissement ne garantit pas que la condition de boucle est correcte.

ELF : un format commun à plusieurs fichiers

ELF signifie *Executable and Linkable Format*. Sur notre Linux, il sert notamment aux fichiers objets, aux exécutables et aux bibliothèques partagées.

Type ELF	Exemple	Usage
REL	objets.o	objet à relier
EXEC	exécutable non PIE	programme à charger
DYN	bibliothèque .so ou PIE	objet chargeable à une base variable

L'extension ne détermine pas le format. Un fichier .o peut être ELF sans être un programme que l'on lance directement.

Le fichier exécutable possède une structure

```
make -C exemples/construction/elf  
file exemples/construction/elf/objets  
readelf -h exemples/construction/elf/objets
```

- Relever la classe, l'architecture et le type.
- ELF contient des en-têtes et des tables; le contenu ne se limite pas aux instructions de `main`.

Reconnaître ELF dans les premiers octets

Dans le dossier de la démonstration :

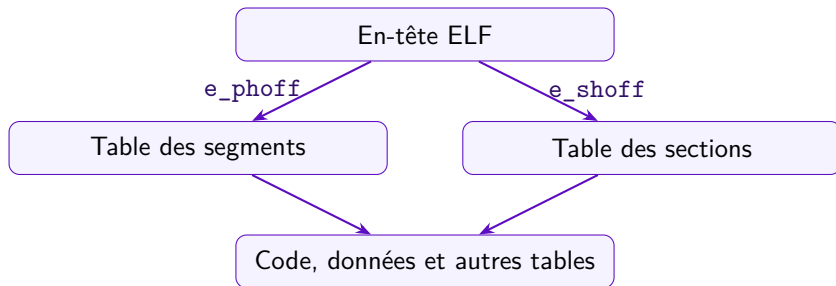
```
od -An -tx1 -N16 objets
```

Début attendu pour notre ELF64 x86-64, petit-boutiste :

```
7f 45 4c 46 02 01 01 ...
```

Décalage depuis le début	Valeur	Sens
0 à 3	7f 45 4c 46	signature : 7f puis ASCII ELF
4	02	classe ELF64 (01 pour ELF32)
5	01	petit boutisme (02 pour gros boutisme)
6	01	version de l'encodage ELF

L'en-tête donne des repères pour lire le fichier



Ce schéma montre les références entre les structures, pas leur ordre exact dans le fichier. `readelf -h` affiche notamment leurs positions et le point d'entrée.

Une table de sections se lit par colonnes

`readelf -SW objets`

- **Nr** donne l'indice ; **Name**, le nom ; **Type**, le type de section.
- **Size** est la taille en octets, affichée en hexadécimal.
- **Off** repère les octets dans le fichier ; **Address**, leur adresse virtuelle prévue lorsqu'elle s'applique.
- **Flg** indique notamment allocation, écriture et exécution.

Relier les déclarations C aux sections observées

```
int compteur = 7;
int visites;
const char message[] = "Observation ELF";
```

Dans le programme fourni, on s'attend à trouver :

- `compteur` dans `.data`;
- `visites`, initialisé à zéro par les règles du C, dans `.bss`;
- `message` dans `.rodata`; les instructions de `main` dans `.text`.

```
readelf -sW objets
objdump -s -j .rodata objets
```

Pourquoi .bss prend de la mémoire sans stocker tous les zéros

- Une globale sans initialiseur, comme `visites`, commence à zéro.
- Une section de type `NOBITS` décrit une taille, sans occuper cette taille en octets dans le fichier.
- Pour un segment `LOAD`, `MemSiz` peut dépasser `FileSiz`; la partie supplémentaire est mise à zéro au chargement.

Une grande zone globale initialisée à zéro peut donc augmenter la mémoire nécessaire sans ajouter autant d'octets au fichier exécutable.

Relier une section à un objet concret

Section courante	Ce qu'on peut y retrouver
<code>.text</code>	instructions
<code>.rodata</code>	données en lecture seule
<code>.data</code>	données statiques initialisées
<code>.bss</code>	stockage statique initialisé à zéro
<code>.symtab</code>	table de symboles lorsqu'elle est présente

L'ordre de ces sections n'est pas fixé. Ces noms ne sont pas forcément les premières lignes de la table.

Les premières sections peuvent décrire le chargement

- `.interp` contient le chemin de l'interpréteur dynamique.
- Les sections `.note.*` portent des métadonnées : ABI, identifiant de construction, propriétés.
- `.gnu.hash` aide à rechercher des symboles dynamiques.
- `.dynsym` décrit les symboles dynamiques ; `.dynstr` contient les chaînes auxquelles les tables dynamiques font référence.
- L'entrée 0 de la table est une entrée nulle ; annoncer si on la compte.

Exemple : relever les cinq premières sections nommées

Extrait de `readelf -SW objets`, limité à trois colonnes :

[Nr]	Name	Type
[0]		NULL
[1]	.note.gnu.build-id	NOTE
[2]	.interp	PROGBITS
[3]	.gnu.hash	GNU_HASH
[4]	.dynsym	DYNSYM
[5]	.dynstr	STRTAB

Ici : identifiant de construction, chemin de l'interpréteur, table de recherche des symboles, symboles dynamiques, puis chaînes de caractères.

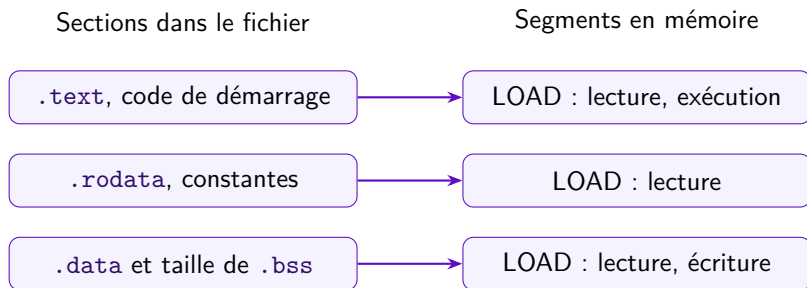
Conserver l'ordre observé et indiquer les indices retenus. Cet ordre peut changer d'un exécutable à l'autre.

Sections et segments répondent à deux questions

```
readelf -SW objets  
readelf -lW objets
```

- Les sections organisent le contenu pour la construction et l'inspection.
- Les segments décrivent ce que le chargeur doit notamment projeter en mémoire.
- Un segment peut regrouper plusieurs sections.

Du fichier à la mémoire : les segments LOAD



Exemple simplifié : lire la correspondance réelle dans `readelf -lW`, sous *Section to Segment mapping*. L'alignement se fait par pages.

Une position dans le fichier n'est pas une adresse mémoire

Supposons un segment dont `Offset = 0x1000` et `VirtAddr = 0x401000`. Un octet placé à l'offset `0x1020` correspond, dans cette zone, à :

$$0x401000 + (0x1020 - 0x1000) = 0x401020$$

- L'offset sert à retrouver l'octet dans le fichier.
- L'adresse virtuelle sert au processus pendant son exécution.
- Pour un PIE, ajouter aussi la base choisie au chargement.

Trouver les bibliothèques demandées

```
readelf -dW objets  
readelf -dW objets | grep NEEDED
```

Les entrées **NEEDED** donnent les dépendances directes demandées par cet exécutable. Elles ne donnent pas toute la liste des dépendances transitives.

Exemple : lire une dépendance et présenter son observation

Dans la sortie de `readelf -dW objets` :

```
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]
```

Le nom demandé est `libc.so.6`, la bibliothèque C. Ce n'est pas un chemin complet vers un fichier installé.

- Noter l'outil utilisé : `readelf`.
- Pour les sections : indice, nom exact et rôle en une phrase.
- Pour les bibliothèques : relever toutes les lignes `NEEDED`.
- Le chemin de `.interp` désigne le chargeur dynamique.

Le point d'entrée n'est généralement pas main

Pour un exécutable lié dynamiquement sous Linux :

- ❶ Le noyau lit les en-têtes et prépare les segments du programme.
- ❷ L'interpréteur indiqué par `INTERP` charge les bibliothèques et prépare la liaison dynamique.
- ❸ Le contrôle rejoint le code de démarrage de l'exécutable, souvent nommé `_start`.
- ❹ Ce code et la bibliothèque C préparent l'appel à `main`.

Comparer `Entry point address` dans `readelf -h` avec les symboles `_start` et `main` dans `nm -n objets`.

Retirer des symboles ne retire pas le code

```
cp objets objets-sans-symboles
strip objets-sans-symboles
nm objets-sans-symboles
readelf -lW objets-sans-symboles
./objets-sans-symboles
```

- La table de symboles ordinaire et les informations de débogage peuvent être retirées.
- Le programme conserve les données nécessaires à son chargement et à son exécution.
- Des symboles dynamiques peuvent encore être visibles avec `nm -D`.

À vous : décrire un exécutable

- Compiler le petit programme fourni dans le dossier ELF.
- Expliquer ses sept premiers octets, puis confirmer avec `readelf -h`.
- Relever les cinq premières sections nommées et justifier leur rôle ; exclure ici l'entrée nulle d'indice 0.
- Identifier l'architecture et toutes les dépendances `NEEDED` ; noter l'outil et les commandes utilisés.
- Relier `visites` à une section et celle-ci à un segment `LOAD`.
- Comparer le résultat et les symboles avant et après `strip`.