

Programmation C avancée

Concepts & Outils
pour le développement



Sondage

#QDLE#S#ABCD#45#

- Quel est, d'après vous, votre niveau en programmation C :
 - A. Au top
 - B. Moyen
 - C. Médiocre
 - D. Très mauvais

LOC !

#QDLE#S#ABCD#30#

- Quel est votre vitesse de frappe au clavier :
 - A. Super Rapide
 - B. Rapide
 - C. Moyenne
 - D. Tortue

Cobaye

- Prenons un volontaire désigné d'office pour faire une performance sur la logiciel typespeed...

Petit Calcul

- à raison de 4 caractères par secondes, 6 heures par jour, 5 jours par semaine, 49 semaines par an pendant 40 ans, combien tapperez-vous de caractères...

$$\dots \cdot 4 * 60 * 60 * 6 * 5 * 49 * 40 = ?$$

questions subsidiaires: combien de tendinites/torticollis/mals de dos...?

Petit Calcul

- 846 720 000
- en supposons qu'une ligne de code contienne 40 caractères, cela fait
- 21 168 000 lignes de codes

LOC (line of code)

- Le nombre de ligne de code est une métrique permettant de mesurer la taille d'un programme.
- Il existe plusieurs sous mesures:
 - Lignes de code
 - Lignes de code logiques (instructions)
- Permet d'estimer la complexité d'un programme, la capacité de développement, la maintenabilité...

LOC (line of code)

- quelques exemples en millions de LOC:

- FreeBSD: 8.8
- Linux kernel 2.6.35: 13.5
- Windows XP: 45
- Mac OSX (10.4): 86
- Debian 5.0: 324

LOC (line of code)

- En travaillant durement tout seul pendant 40 ans, vous pourriez à peine faire mieux qu'un noyau... qui sera obsolète. Car le calcul ne tient pas compte du temps de debug/re-écriture/...
- Comment faire?

LOC (line of code)

- En travaillant durement tout seul pendant 40 ans, vous pourriez à peine faire mieux qu'un noyau... qui sera obsolète. Car le calcul ne tient pas compte du temps de debug/re-écriture/...
- Comment faire?
 - améliorer sa productivité

LOC (line of code)

- En travaillant durement tout seul pendant 40 ans, vous pourriez à peine faire mieux qu'un noyau... qui sera obsolète. Car le calcul ne tient pas compte du temps de debug/re-écriture/...
- Comment faire?
 - améliorer sa productivité
 - **travailler avec les autres**

Introduction

- Sur l'informatique et la mémoire...

```
1001000110100110110010000010010011001111  
01101100000100001111000011110100110010110  
0000110100111101001001111100111000001101  
10111110011000001110000110110011001011100  
001111001111010111001011001011000001110  
100110111100000111001111000011001011100  
0011101011100000111010011011110000011110  
01110111110101100000111010011011111001  
001100001111001101101000001001001100000  
110100011011111000011001011000001111001  
110111111010110000011001011101110101010  
110111111001100000110100111101001000010
```

- Un ordinateur n'est qu'une machine qui transforme et 0 en 1 et inversement
- Tout est dans l'interprétation des suites de 0 et de 1

Introduction

#QDLE#Q#ABCD*#60#

- Par exemple, la suite :

01000100010001010100000101000100

- Que représente cette suite ?

A. L'entier sur 4 oct: 1145389380

B. Les quatre entiers sur 1 oct: 68, 69, 65, 68

C. Ou, si l'on pose qu'on associe le symbole A au nombre 65, les quatre lettres DEAD

D. autre



Objectifs du module

- Maitrise des outils de compilation en C
- Bonne compréhension des mécanismes de gestion de la mémoire
- Maitrise des outils de développement :
 - Gestion des sources
 - Tests
 - Intégration continue
 - ...
- Écriture de programmes : sains, maintenables, robustes, évolutifs



Objectifs...

#QDLE#S#AB#30#

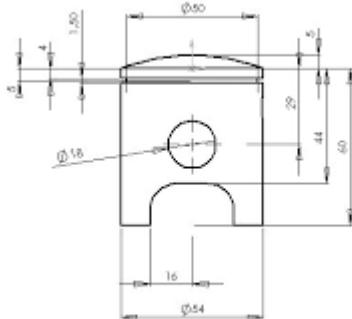
- Est-ce que les objectifs du module vous paraissent clairs ?
 - A. Oui
 - B. Non

Plan

- Des sources à la mémoire
- Allocations et analyses
- Convention de code, Documentation
- Gestion des sources, dépôt et compilation auto
- L'intégration continue
- Les tests : types, utilisation et framework
- Couverture & intégration continue (suite)
- Performance : localité et analyse
- Pointeurs de fonction, chargement dynamique

Des sources à la mémoire

- Un fichier source est un texte exprimé dans un langage de programmation.
- Un binaire est un fichier qui contient des instructions machines.
- Un exécutable est un binaire ayant un point de début d'exécution.



SOURCE
Hello.c



binaire
Hello.o



exécutable
Hello

Script shell

#QDLE#Q#A*BC#30#

```
#!/bin/bash

echo « hello »
echo « world »
echo $USER
```

```
#!/usr/bin/python

import sys

print sys.argv
```

- Un script est

- un fichier source ?
- un fichier binaire ?
- un fichier binaire exécutable ?

Script shell

```
#!/bin/bash

echo « hello »
echo « world »
echo $USER
```

```
#!/usr/bin/python

import sys

print sys.argv
```

- Un script est

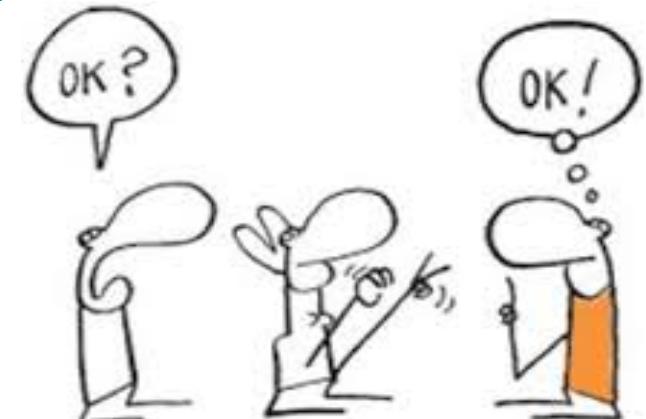
- un fichier source ?
- un fichier binaire ?
- un fichier binaire exécutable ?

Un script est un fichier source qui est interprété.

Dans le cas d'un script shell, l'interpréteur est le shell. Dans celui de python c'est le programme python.

La programmation

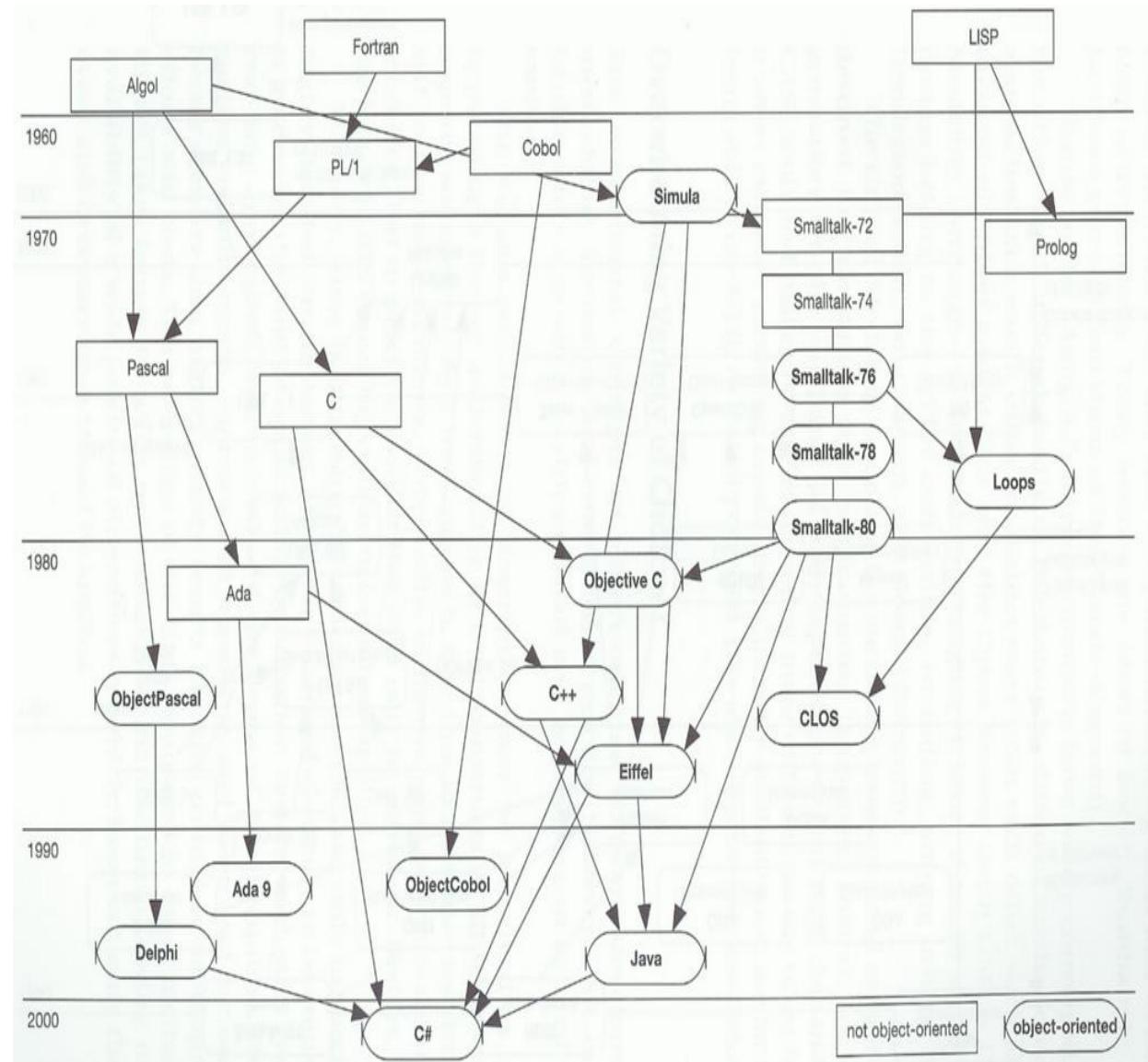
- Un langage de programmation nous aide à structurer la mémoire et son interprétation.
- Il se place entre le programmeur, dont la vision est très haut niveau et la machine dont la « vision » est très bas niveau.
- Il existe de nombreux langages : typés, non typés, compilés, interprétés, impératifs, fonctionnels,....
- La langage C est un langage impératif typé et compilé.



Combien existe-t-il de langage de programmation ?

#QDLE#Q#ABC*D#25#

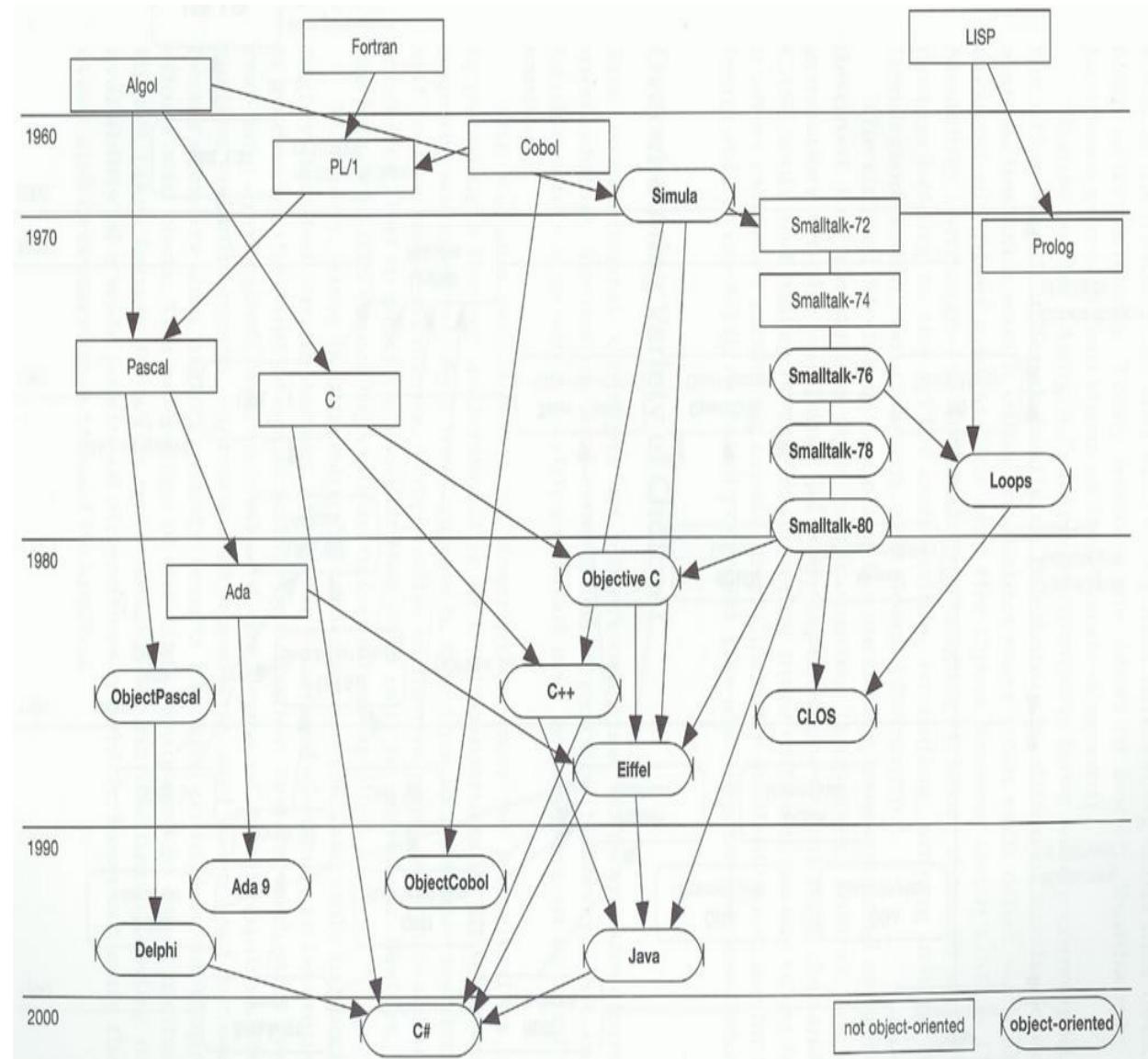
- A. plus de 2000
- B. plus de 1000
- C. plus de 500
- D. moins de 500



Combien existe-t-il de langage de programmation ?

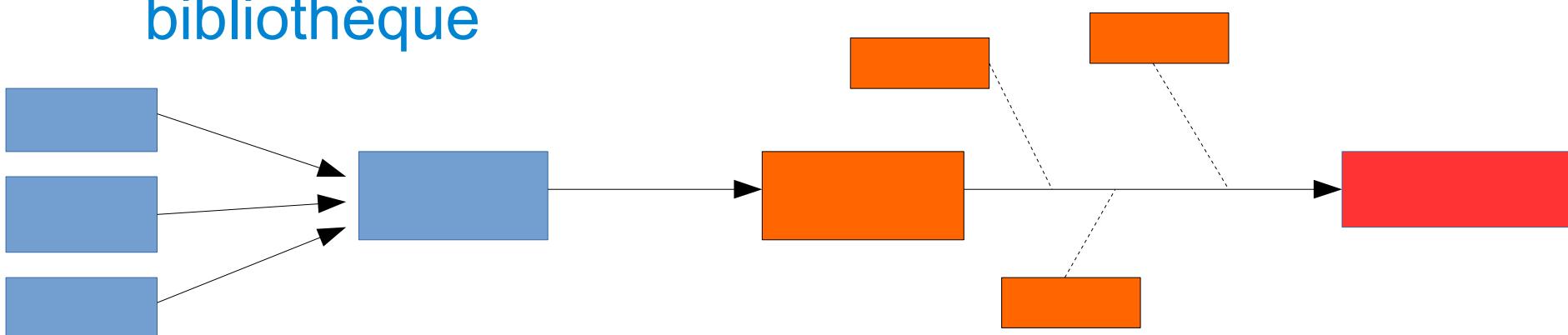
- A. plus de 2000
- B. plus de 1000
- C. plus de 500
- D. moins de 500

wikipedia
référence
à ce jour
698 langages



Fichier source et compilation

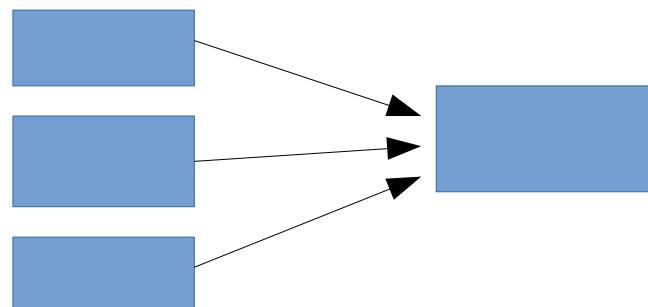
- Le compilateur est un programme qui lit des sources et produit un binaire possiblement exécutable
- La compilation comporte trois étapes :
 - Le pré-processing : sources => source
 - La compilation : source => binaire
 - L'édition de lien : binaires => exécutable, bibliothèque



La pré-compilation



- La pré-compilation prend un ou plusieurs fichiers sources en entrée et produit un unique fichier source exempt de macro :
 - Tous les « #include » sont remplacés par leurs contenus
 - Les macro (#define, #ifdef,...) sont interprétées et remplacer par leurs évaluations
- Le résultat est un unique fichier source C sans dépendance



La pré-compilation : exemple

```
#define N 10
```

```
int main(){
```

```
    int i,j=0;
```

```
    for(i=0;i<N;++i)
```

```
        j+=i;
```

```
    return j;
```

```
}
```

```
# 1 "exemple.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello.c"
```

```
int main(){
    int i,j=0;
    for(i=0;i<10;++i)
        j+=i;
    return j;
}
```

```
gcc -E exemple.c
```

La pré-compilation : exemple 2

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

#define MESSAGE "hello\n"

int main(){
    printf(MESSAGE);
    return EXIT_SUCCESS;
}
```

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
.....
extern int printf (const char *__restrict __format, ...);
.....
# 3 "hello.c" 2
```

```
gcc -E hello.c
```



La pré-compilation

- Quelques mots sur les macros :

substitution :

```
#define NAME VALUE  
#define NAME(arg) VALUE
```

mise en chaîne :

```
#define s(x) #x
```

Retrait :

```
#undef
```

- Branchement conditionnel :

```
#ifdef, #ifndef, #endif, #else, #elif, #defined
```

Macros : substitution

```
#define N 10  
  
int main(){  
    int i=N ;  
}
```

```
gcc -E toto.c
```

```
int main(){  
    int i=N ;  
}
```

```
gcc -E -DN=10 toto.c
```

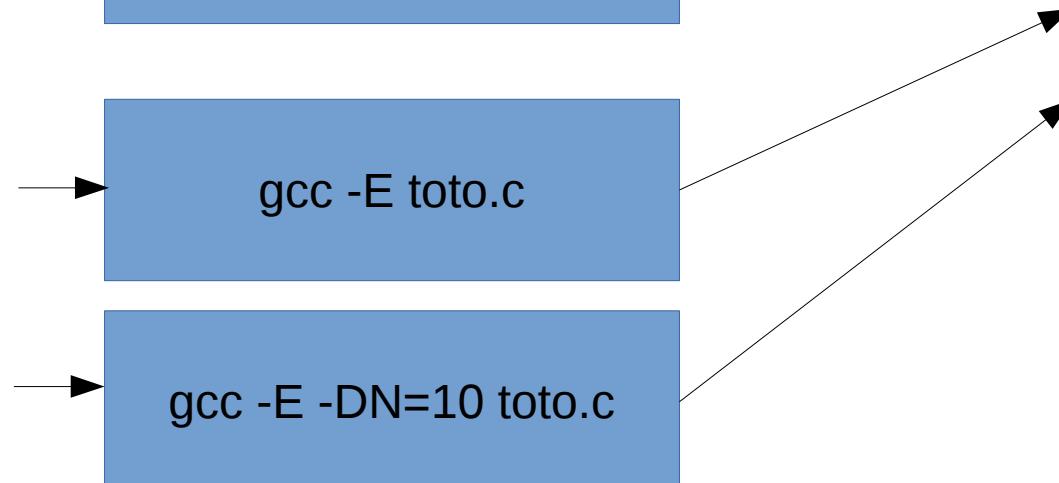
```
int main(){  
    int i=10 ;  
}
```

```
#ifndef N  
#define N 10  
#endif  
  
int main(){  
    int i=N ;  
}
```

```
gcc -E toto.c
```

```
gcc -E -DN=10 toto.c
```

```
int main(){  
    int i=10 ;  
}
```



macros : concaténation et mise en chaîne

```
#define str(x) struct point_##x { \
    x value ; \
}
```

```
str(int) ;  
str(float) ;
```



```
struct point_int {  
    int value ;  
};  
  
struct point_float{  
    float value ;  
};
```

```
#define msg(x) if (x) { \  
    printf(#x) ; \  
}  
  
int main(){  
    int i=0 ;  
    msg(i+1) ;  
}
```



```
int main(){  
    int i=0 ;  
    if (i+1) { printf("i+1") ; } ;  
}
```

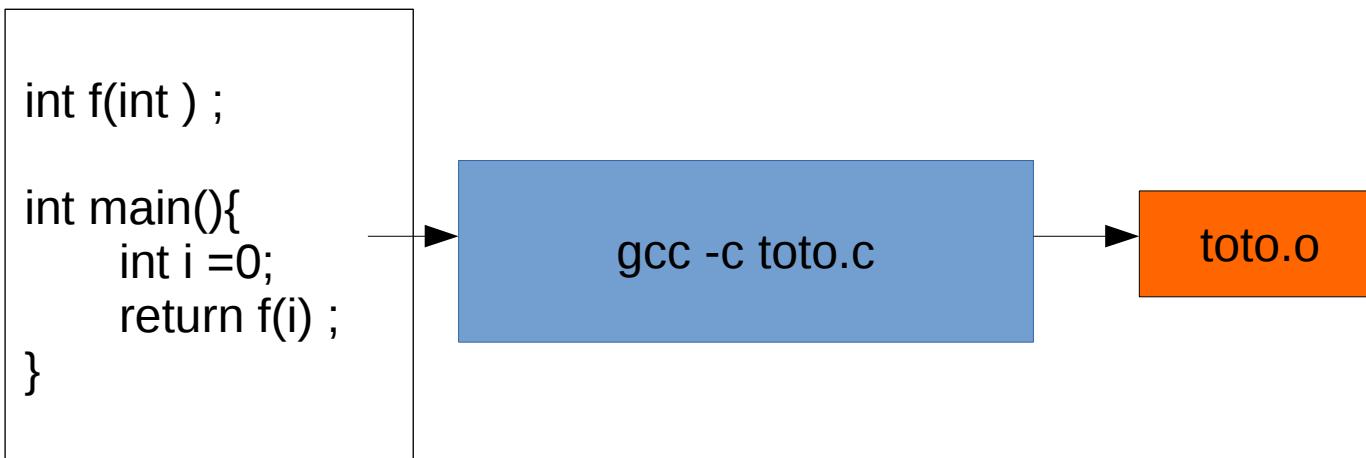
La compilation

- Traduction d'un fichier source en un fichier binaire.



- Les instructions sont transformées en code machine, regroupé par fonctions.
- Les fonctions non implémentées sont indiquées comme « symboles à résoudre ». Seul le nom est indiqué.
- Les variables globales externes sont déclarées
- Pour compiler, toute fonction doit être soit implémentée, soit déclarée.

Compilation : objdump



```
toto.o: file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 I df *ABS* 0000000000000000 toto.c
0000000000000000 I d .text 0000000000000000 .text
0000000000000000 I d .data 0000000000000000 .data
0000000000000000 I d .bss 0000000000000000 .bss
0000000000000000 I d .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 I d .eh_frame 0000000000000000 .eh_frame
0000000000000000 I d .comment 0000000000000000 .comment
0000000000000000 g F .text 0000000000000001b main
0000000000000000 *UND* 0000000000000000 f
```

The output of the `objdump -t toto.o` command is shown in a large blue box. It displays the symbol table for the `toto.o` file, which is an ELF64 executable. The table includes entries for the `main` function and an undefined symbol `f`.

Compilation

```
extern float x ;  
int f(int );  
int i ;  
static int j ;  
int g(){  
    static int k=0;  
    return k++;  
}  
  
int main(){  
    int i =0;  
    return f(i);  
}
```

- A la fin de cette étape, on dispose d'un unique fichier dit « objet » qui contient :
 - Un ensemble de variables globales
 - Un ensemble de variables statiques
 - Un ensemble de fonction : nom + instructions
 - Un ensemble de symboles « manquants »

gcc -c a.c



a.o

nm a.o

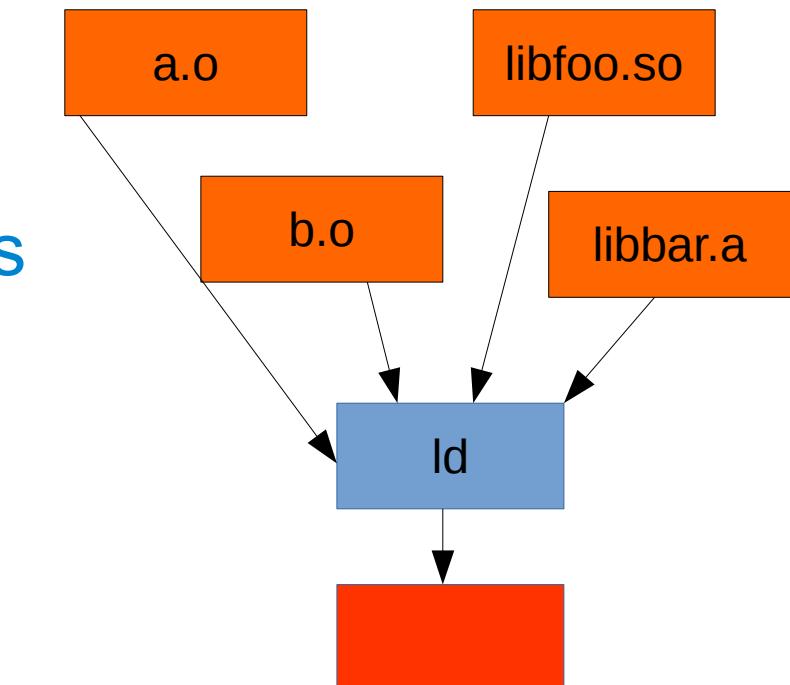
U f
0000000000000000 T g
0000000000000004 C i
0000000000000000 b j
0000000000000004 b k.1748
0000000000000015 T main

U : The symbol is undefined.
T : The symbol is in the text (code) section
C : The symbol is common. Common symbols are uninitialized data.
b : The symbol is in the uninitialized data section (known as BSS)

L'édition de liens



- L'édition de lien consiste à interconnecter différents « objets » au sein d'une bibliothèque ou bien d'un binaire exécutable.
- En entrée, l'éditeur de lien (`ld`) prend :
 - des fichiers objets
 - des bibliothèques statiques
 - des bibliothèques dynamiques
- En sortie on obtient :
 - une bibliothèque **dynamique**
 - un binaire exécutable



L'édition de liens



- L'éditeur de lien énumère l'ensemble des symboles fournis et manquants
- Lorsqu'un symbole est manquant dans un objet mais fournit dans un autre, alors les deux sont liés et le symbole est **résolu**
- Si un symbole manquant n'est fournit par aucun autre objet alors il est dit manquant :

```
gcc a.o -o a
```

a.o: In function `main':
a.c:(.text+0xa): undefined reference to `f'
collect2: error: ld returned 1 exit status



L'édition de liens

#QDLE#Q#AB*C#30#



```
> cat a.c
int main() {
    return f();
}
> gcc -c a.c
> nm a.o :
U f
00000000 T main
```

```
gcc -c b.c
nm b.o :
00000004 C f
```

gcc a.o b.o

```
0000000000601038 B __bss_start
0000000000601038 b completed.7259
0000000000601028 D __data_start
0000000000601028 W data_start
0000000000400430 t deregister_tm_clones
00000000004004b0 t __do_global_dtors_aux
0000000000600e18 t __do_global_dtors_aux_fini_array_entry
0000000000601030 D __dso_handle
0000000000600e28 d _DYNAMIC
0000000000601038 D _edata
0000000000601040 B _end
000000000060103c B f
0000000000400584 T _fini
00000000004004d0 t frame_dummy
0000000000600e10 t __frame_dummy_init_array_entry
00000000004006b8 r __FRAME_END__
0000000000601000 d __GLOBAL_OFFSET_TABLE__
w __gmon_start__
00000000004003a8 T _init
0000000000600e18 t __init_array_end
0000000000600e10 t __init_array_start
0000000000400590 R __IO_stdin_used
```

→ segfault, pourquoi ?

- A. parce que f n'est pas initialisée dans b.c
- B. parce que f est une fonction dans a.c mais une variable globale dans b.c
- C. parce que f est une variable globale dans a.c mais une fonction dans b.c

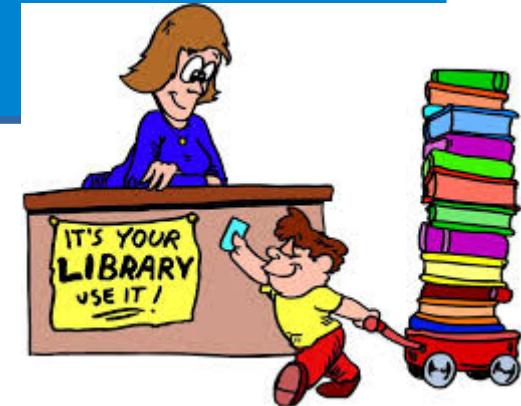
L'édition de liens

- Aucune cohérence de type des symboles n'est effectuée lors de l'édition
 - > d'où l'importance de fichier d'entête s'assurant au moment de la compilation de cette cohérence
- Si un symbole est présent en plusieurs versions, une erreur est signalée :

```
gcc a.o b.o c.o

c.o: In function `pow':
c.c:(.text+0x0): multiple definition of `pow'
b.o:/tmp/b.c:1: first defined here
collect2: error: ld returned 1 exit status
```

Les bibliothèques statiques



- Les bibliothèques statiques sont un regroupement d'objets
- Elles peuvent être créées avec l'outil « ar »

```
ar rcs ma_bibilo.a b.o c.o d.o

nm ma_biblio.a
b.o:
0000000000000000 T b

c.o:
0000000000000000 T c

d.o:
          U c
0000000000000000 T d
```

- Lors de l'utilisation, si un objet d'un bibliothèque apporte un symbole manquant, alors il est intégralement inclus (l'objet par la bibliothèque).

```
nm a.out | grep -v _
00000000040051b T c
000000000601038 b completed.7259
00000000040050b T d
0000000004004f6 T main
```

Les bibliothèques dynamiques

- Une bibliothèque dynamique est binaire qui regroupe un ensemble de symboles.
- Lors de l'édition de liens, si un symbole manquant est fourni par un biblio. dyn. alors un lien est créé vers cette bibliothèque :

```
gcc -fPIC -shared -o libtoto.so b.c c.c d.c
```

```
gcc -c a.c
```

```
gcc a.o -ltoto -L.
```

```
nm a.out | grep -v _  
000000000601040 b completed.7259  
U d  
000000000400696 T main
```

```
LD_LIBRARY_PATH=. ldd a.out
```

```
linux-vdso.so.1 => (0x00007fff8e522000)  
libtoto.so => ./libtoto.so (0x00007f63a7ea2000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f63a7ac4000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f63a80a6000)
```

