Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

Objectifs du module

- Maîtrise des outils de compilation en C
- Bonne compréhension des mécanismes de gestion de la mémoire
- Maîtrise des outils de développement :
  - Gestion des sources
  - Tests
  - Intégration continue
  - …

- Écriture de programmes : sains, maintenables, robustes, évolutifs

Evaluation du module

- Épreuve de deux heures sur l'environnement machine :
  - Pas d'accès internet
  - Documents et supports interdits
  - Du code à analyser avec un fichier à remplir :
  - La note repose uniquement sur les réponses du fichier texte.

```
======================
== Binaire: 6 points ==
======================
L'exercice se base sur les fichiers presents dans le
répertoire "buggy".


De quelles bibliothèques dépend le programme exe? (1)
#DEBUT A1
#FIN A1


Quelles sont les fonctions externes (qui seront donc
apportées par les bibliothèques) dont dépend
le programme? (1)
#DEBUT A2
#FIN A2
```

Objectifs...

#QDLE#S#AB#30#

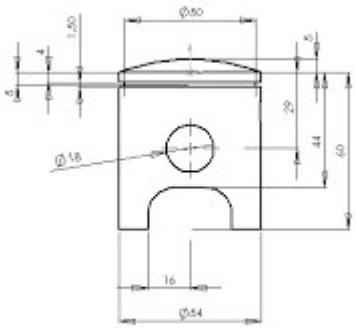
- Est-ce que les objectifs du module vous paraissent clairs ?
  - A. Oui
  - B. Non

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
```

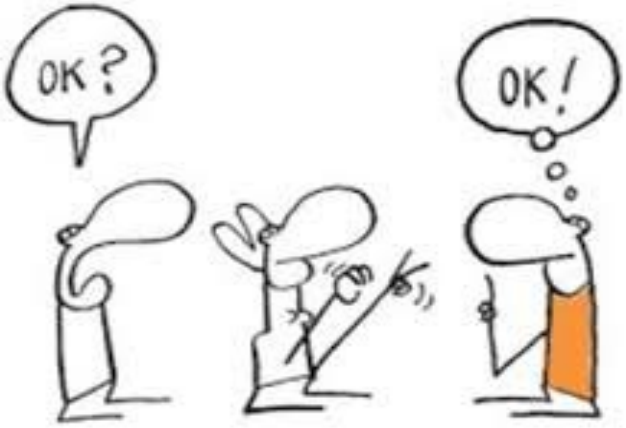
- Un script est
  - un fichier source ?
  - un fichier binaire ?
  - un fichier binaire exécutable ?

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.

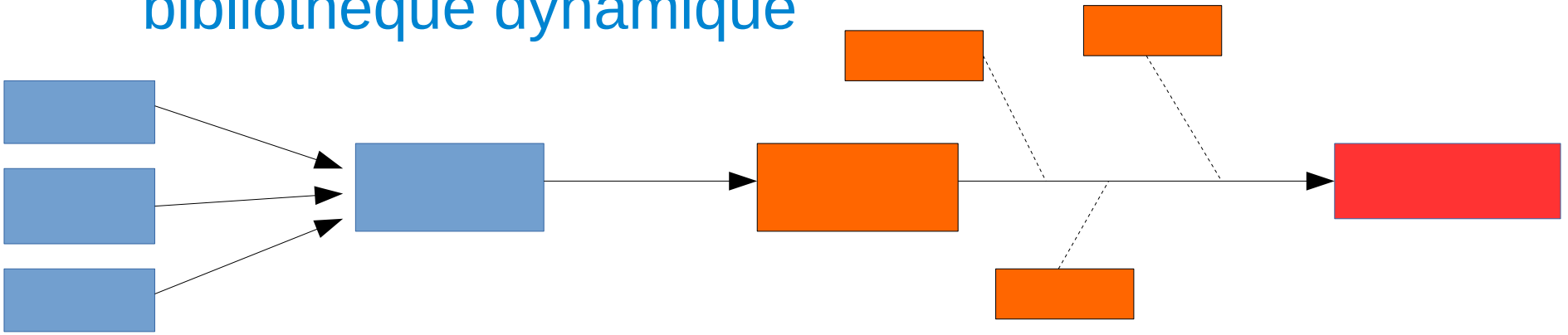
La programmation

- Un langage de programmation nous aide à structurer la mémoire et l'interpréter.
- 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.



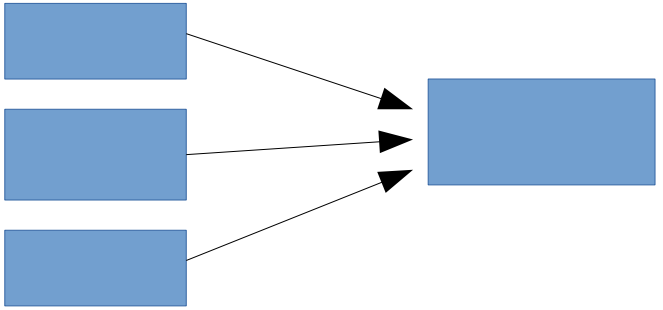
Fichier source et compilation

- 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 dynamique



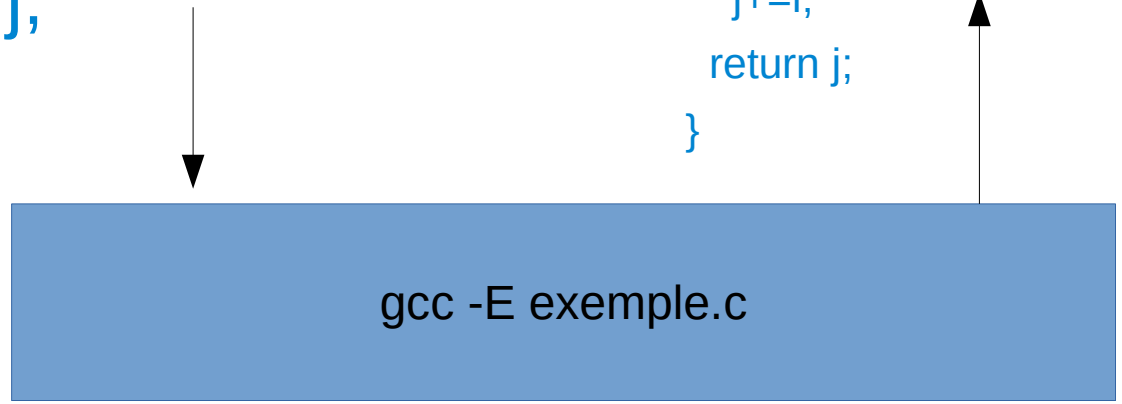
La pré-compilation

- 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 remplacées 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;
}
```

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



int main(){
  printf("hello\n");
  return 0;
}
```

La pré-compilation : include

- *#include <foo>* :
  - Recherche le fichier *foo* dans un ensemble de répertoires prédéfinis : */usr/include ; /usr/local/include* et également dans les répertoires indiqués via l'option -Idir du compilateur, par exemple `gcc -I/home/allali/include .` - I est prioritaire sur les répertoires système.
- *#include ''foo''* :
  - foo est d'abord cherché relativement au fichier source, puis, selon le même schémas que ci-dessus.

La pré-compilation : MACRO

- Une macro est avant tout soit existante (defined), soit inexistante (undefined).
- Si elle existe, elle peut optionnellement avoir une valeur et possiblement être une fonction
- Les macro repose sur une mécanique de substitution : il n’y a pas de notion de typage etc.

La pré-compilation : MACRO

```
#define M       // M existe à partir de maintenant, mais n'a pas de valeur associée

#if defined(M) // test l'existence de M
…    // code qui sera gardé pour la compilation si M existe
#endif

#ifdef M         // equivalent
…
#endif

#if !defined(M)
….  // code gardé si M n'existe pas
#endif

#ifndef M
….
#endif
```

La pré-compilation : MACRO

Exemple de macro :

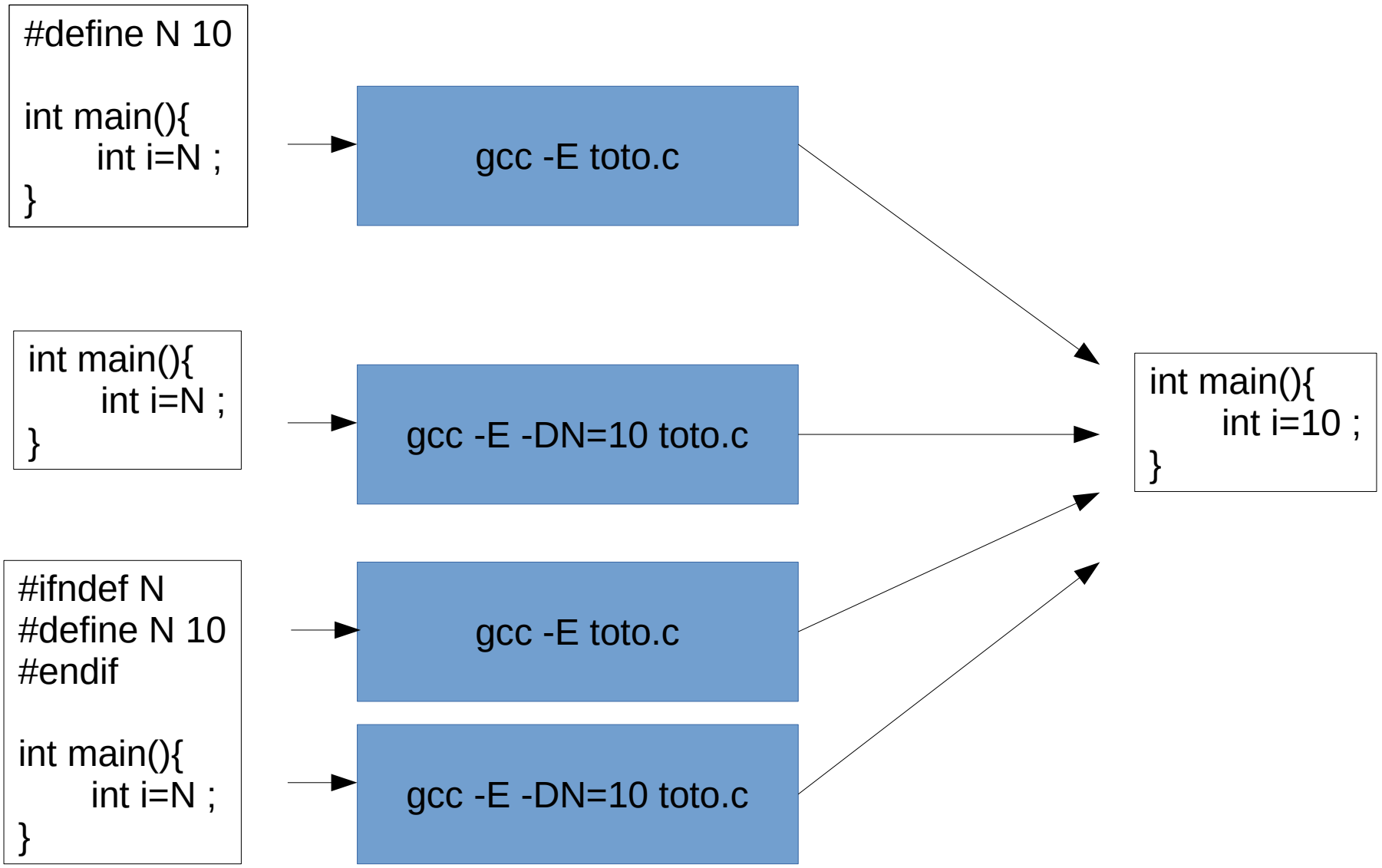
```
#define M 10
#define P1(a) a+1
#define S(a,b) b,a
```

etc...

Macros : à la compilation

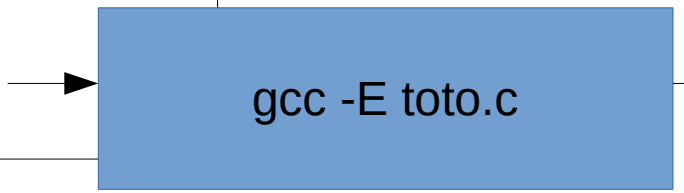
- Il est possible de définir une macro lors de la compilation avec l'option -D :
  - gcc -DN=10
    - => for(i=0;i<N;++i)…
    - => int array[N] ;
  - gcc -DLinux
    - => #ifdef Linux …. #endif
- Ceci entraîne de la **modularité** : il est possible de paramétrer un code sans avoir à éditer les fichiers sources

Macros : substitution



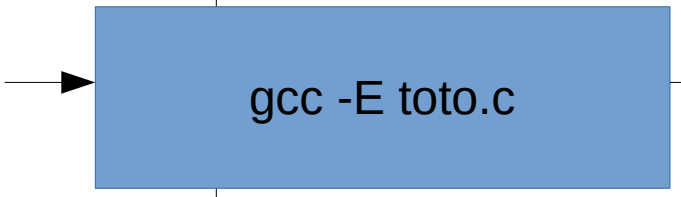
macros : concaténation et mise en chaîne

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

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




```
struct point_int {
int value ;
} ;

struct point_float{
float value ;
} ;
```

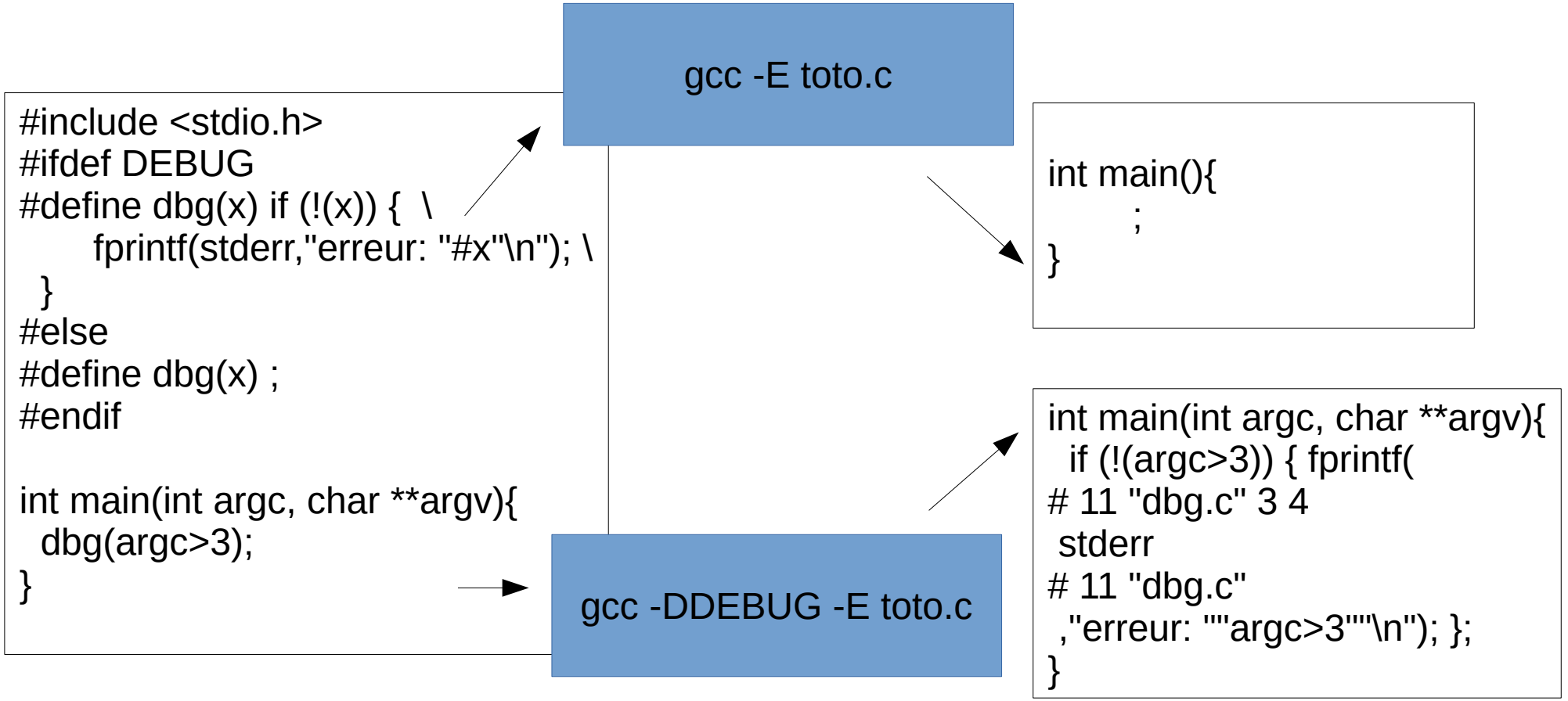
```
#define msg(x) if (x) { \
        printf(#x) ; \
  }

int main(){
  int i=0 ;
  msg(i+1) ;
}
```




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

macros : mise en chaîne

Exemple d'utilisation pour une macro de debug type assert :



La compilation

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

- Les instructions sont transformées en code machine, regroupées 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.

Anaylse d’un binaire (.o)

- Deux outils permettent de nous donner des informations sur le contenu d’un binaire :
  - nm
  - objdump
- Ils permettent de lister :
  - les fonctions implémentées
  - les fonctions manquantes
  - les variables globales
  - les constantes globales (chaînes de caractères par exemple)

Compilation : objdump

```
int f(int ) ;

int main(){
        int i =0;
        return f(i) ;
}
```

gcc -c toto.c → toto.o

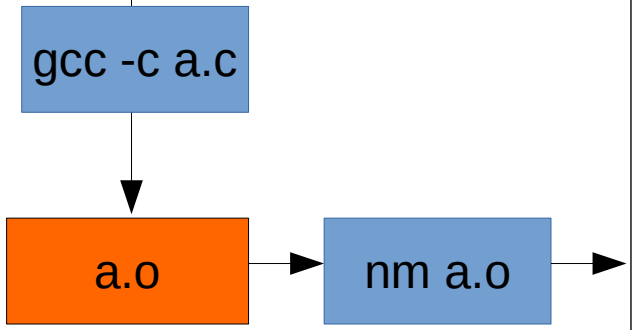
objdump -t toto.o →

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

SYMBOL TABLE:
0000000000000000 l    df *ABS*    0000000000000000 toto.c
0000000000000000 l    d  .text    0000000000000000 .text
0000000000000000 l    d  .data    0000000000000000 .data
0000000000000000 l    d  .bss     0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack      0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text    000000000000001b main
0000000000000000         *UND*    0000000000000000 f
```

Compilation : nm

```
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 fonctions : nom + instructions
  - Un ensemble de symboles « manquants »

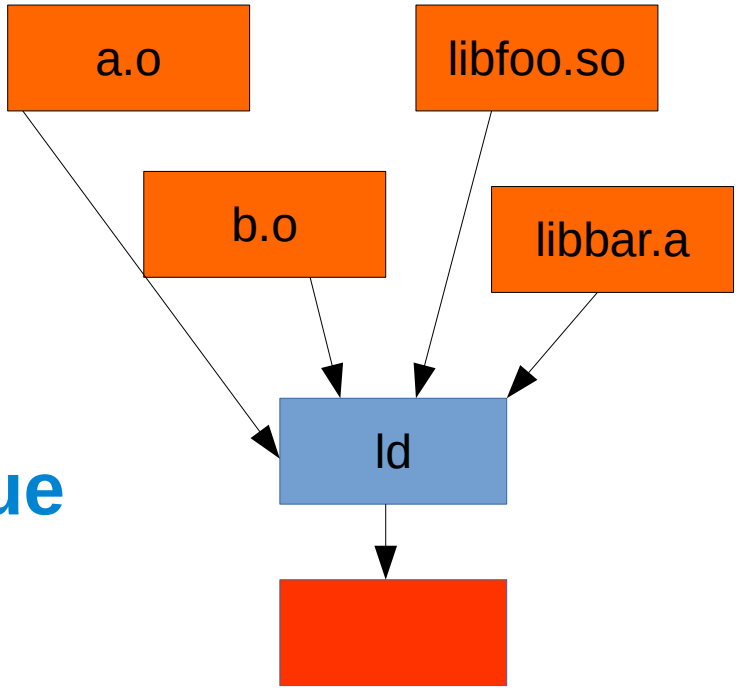



```
                 U f
0000000000000000 T g
0000000000000004 C i
0000000000000000 b j
0000000000000004 b k.1748
0000000000000015 T main
                 U x
```

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 dynamique 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 :
  - soit une bibliothèque **dynamique**
  - soit 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 pas la bibliothèque).

```
nm a.out  | grep -v _
000000000040051b T c
0000000000601038 b completed.7259
000000000040050b 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 fournit 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 _
0000000000601040 b completed.7259
                 U d
0000000000400696 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)
```

Utilisation d’une bibliothèque

- Pour utiliser une bibliothèque on utilise l’option *-lfoo*
- *Les bibliothèques sont cherchés dans des répertoires systèmes (/usr/lib par ex). On peut ajouter des répertoires de recherche avec l’option -Lrep*
- Le compilateur va chercher un fichier *libfoo.so* en priorité puis un fichier *libfoo.a* s’il ne trouve pas de version dynamique
- On peut « forcer » l’utilisation de la version statique ainsi :
  - gcc exemple.c -Wl,-Bstatic -lfoo -Wl,-Bdynamic -lboo -L.

Question

#QDLE#Q#ABC*D#60#

- Laquelle de ces affirmations est fausse :
  - Les bibliothèques dynamiques permettent la correction de bug sans toucher aux exécutables qui les utilisent.
  - L'utilisation des bibliothèques statiques produit des exécutables plus volumineux
  - L'utilisation des bibliothèques dynamiques est plus « sécurisée »
  - La suppression d'une bibliothèque dynamique sur mon système va empêcher l’exécution de programmes qui l'utilisent.

En bref...

- La *compilation* se fait en deux ou trois étapes:
  - pré-compilation: substitution des macros, gestion des inclusions => un unique fichier source
  - compilation: vérification des types et production d'un binaire avec possiblement des dépendances (symboles non résolus)
  - édition de liens: résolution des dépendances, inclusion de binaire ou ajout de liens vers de bibliothèques dynamiques (pas de vérification de types)
- Une bibliothèque statique est une archive de binaire (.o)
  - son utilisation ajoute les .o de l'archive nécessaires à la production finale
- Une bibliothèque dynamique est un binaire sans dépendances non résolues.
  - son utilisation ajoute un lien entre le binaire final et la bibliothèque.