PG120:
Outils pour la programmation C
6 séances de 2h en cours intégrés

Julien Allali / allali@enseirb-matmeca.fr

Objectifs du module

- Utilisation de git : commit/push/pull, branch…
- Compilation : macros & pré-compil, édition de liens, bibliothèques statiques et dynamiques
- Automatisation de la compilation avec Makefile
- Écriture de tests (macros et assert)
- gdb et valgrind

gestion de sources

- Un gestionnaire de sources permet de conserver l'historique des modifications apportées à un ensemble de fichiers.
- Il permet à un ensemble d'utilisateurs d’interagir sur un même code source.
- Tous les gestionnaires de code sources sont basés sur les principes de *diff* et *patch*
- Il existe deux grandes catégories de gestionnaires :
  - les centralisés
  - les décentralisés

Les gestionnaires centralisés

- Historiquement, cvs (concurent versioning system) et son successeur svn (subversion)
- Les gestionnaires centralisés reposent sur un serveur central qui archive toutes les modifications apportées au code.
- Les développeurs n'ont qu'une version du code (l'accès aux anciennes versions passe par des requêtes au serveur).
- Chaque modification incrémente un numéro de révision
- Les commandes de base de svn :
  - checkout, update, infos, status, commit, diff, revert

svn

- Chaque utilisateur interagit avec le serveur, il n'y a pas d'échange direct entre deux utilisateurs.
- Une méthodologie est associée à l'utilisation de svn, vous retrouverez cette méthodologie dans tout projet de développement (open source ou entreprise)

  « svn red book »

- Aujourd’hui svn n’est quasiment plus utilisé

Les gestionnaires dé-centralisés

- Dans ce cas, il n'y a pas de dépôt central.
- Chaque utilisateur gère son propre dépôt.
- Un protocole permet l'échange de modifications locales (commit) entre deux utilisateurs.
- Exemple : git
- Commandes de base :
  - clone, status, log, add, commit, push, pull, fetch, merge, branch, checkout

git : les commits, pull et push

- Dans git, les commits sont locaux (il n'y a pas de dépôt central).
- Les commits sont identifiés par un « code » (sha) unique construit principalement à partir de son contenu, du message et de l'auteur.
- On peut transmettre un ensemble (suite) de *commit* à un autre «dépôt» avec la commande *push*
- On peut réceptionner un ensemble de *commit* depuis un autre dépôt avec la commande *pull*.
- On peut ajouter/supprimer autant de dépôts connectés à notre dépôt avec « remote ». A faire dans les deux dépôts si l'on souhaite une symétrie.
- Note : un dépôt « standard » utilisateur n'accepte pas de recevoir un push (pour cela il doit être *bare*)

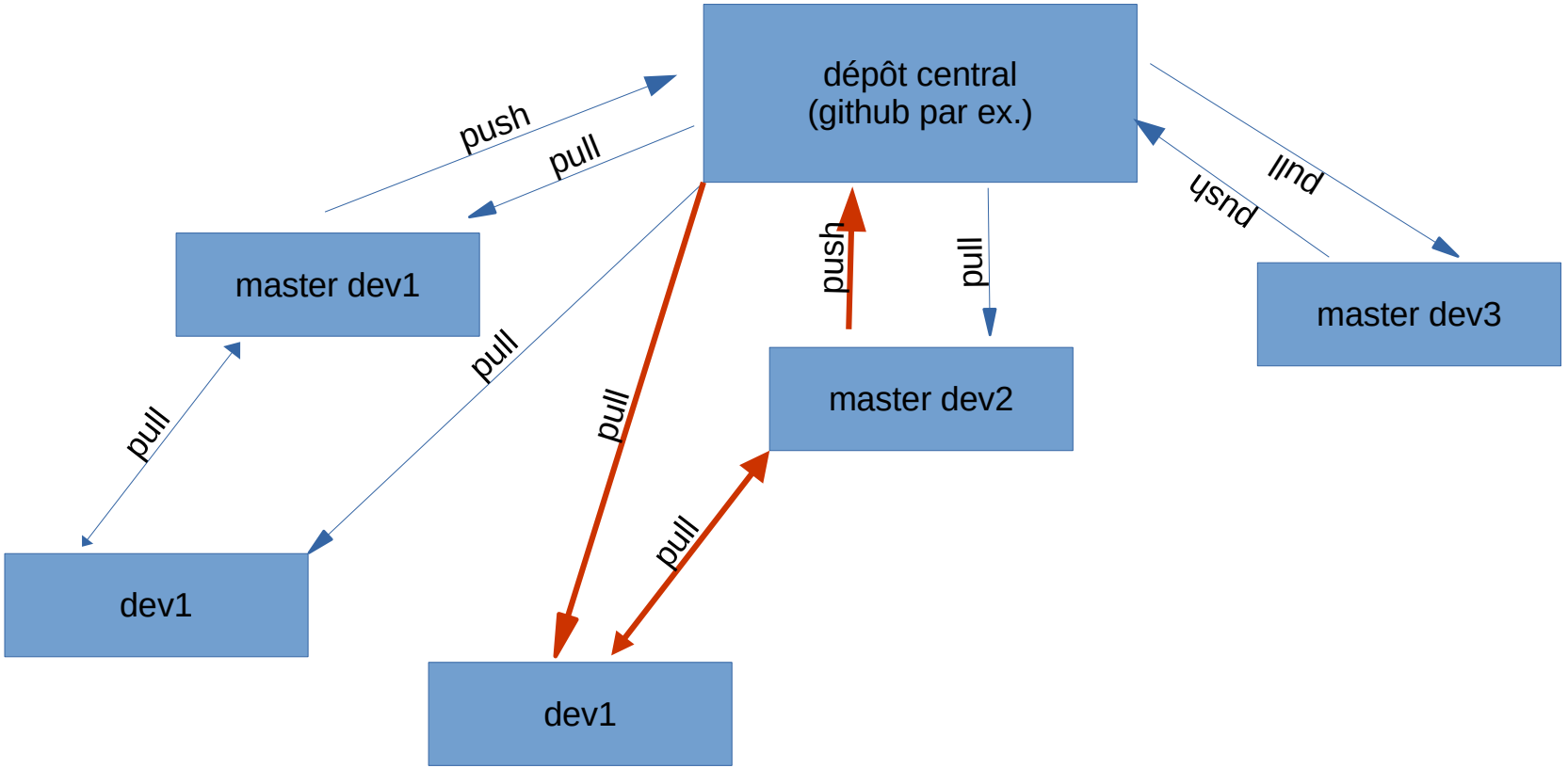
Git : quelques commandes

- git status : liste les fichiers modifiés et non archivés
- git log : affiche l'historique des modifications
- git add : ajoute un fichier pour le prochain commit
- git commit : crée un point d'histoire à partir des derniers fichiers ajoutés (*git commit --amend* permet de modifier le dernier commit)
- git remote -v : affiche la liste des dépôts associés
- git pull : se mettre à jour par rapport à un dépôt
- git push : transmettre nos modifications à un dépôt
- git config : accède/modifie des propriétés générales

re-centralisation

- L'avantage d'être en décentralisé et de pouvoir faire des « commit » sans connexion à un serveur.
- Pour la plupart des projets, il est cependant nécessaire d'avoir une référence : on utilise alors un dépôt git comme tel (github ou thor par exemple). Un tel dépôt s’appelle un « bare », il a la spécificité d’accepter les *push*
- On peut ensuite mettre en place un système de propagation hiérarchique des « commits ».

git


dépôt central
(github par ex.)
push
pull
master dev1
pull
pull
dev1
push
pull
master dev2
pull
pull
dev1
usnd
llnd
master dev3

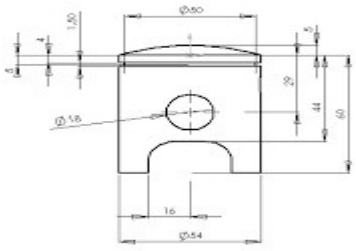
Clone et config

- Sur la plateforme « thor », aller sur la page de PG120 et trouver l'unique dépôt.
- Cloner ce dépôt sur votre machine (*git clone <URL> ; préférer le lien ssh*)
- Utiliser la commande *remote* de *git* (avec option) pour retrouver l'adresse du dépôt
- Utiliser la commande *git config -l* puis modifier user.email pour que cela soit votre adresse enseirb-matmeca (niveau local ou global).
- Créer un répertoire à la racine nommé selon votre login et ajouter dans ce répertoire un fichier « login » contenant votre login (git n'aime pas les répertoires vides).
- Utiliser les commandes status, *add* puis *commit* pour enregistrer cette modification. Vous pouvez choisir l'éditeur utilisé pour saisir le message via la variable de config *core.editor* ou encore utilisé l'option *-m*
- Aller sur thor et regarder si votre répertoire est visible
- Utiliser la commande push (cela va être compliqué)
- Aller sur thor et regarder si votre répertoire est visible
- Utiliser la commande *git log* pour voir les modifications

La compilation, les binaires et bibliothèques

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.



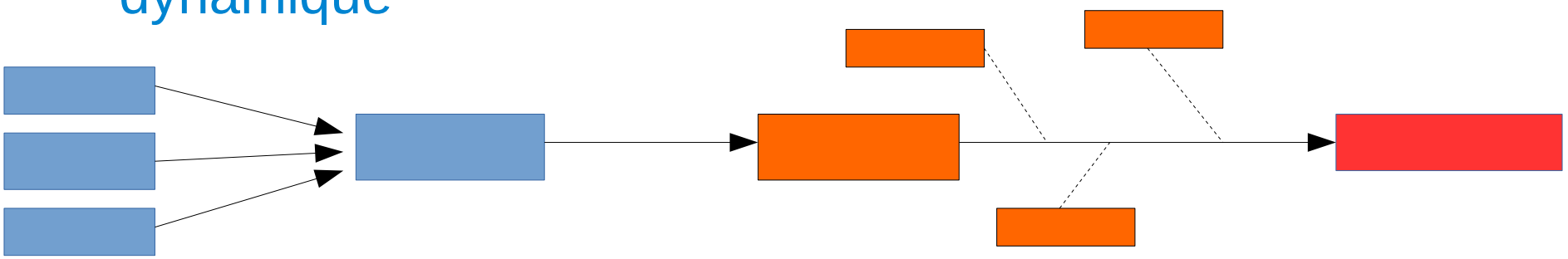
SOURCE
Hello.c

binaire
Hello.o

exécutable
Hello

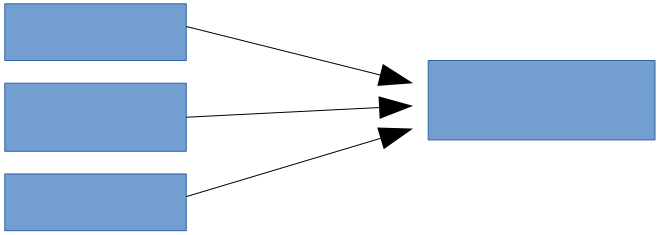
Fichiers sources et compilation

- 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



Pré-compilation : #include

- La macro #include peut être utilisée avec ”path/foo.h” ou bien <foo.h>
- L’option *-Ipath* (pas d’espace entre le *I* et *path*) permet d’indiquer des répertoires dans lesquels chercher des fichiers inclus (l’option peut être utilisée plusieurs fois).
- Si l’on utilise <path/foo.h>, le compilateur cherche path/foo.h relativement aux répertoires indiqués dans les option -I, puis dans les répertoires systèmes (/usr/include/ par ex).
- Si l’on utilise ”path/foo.h”, alors le compilateur cherche relativement au répertoire du fichier compilé puis avec ceux de l’option **-I** et enfin avec les répertoires systèmes.
- Dès que le fichier foo.h est trouvé, alors la recherche s’arrête et le contenu du fichier est lu.

define

- la macro *#define* permet de créer une macro avec ou sans valeur associée :
  - #define Linux
  - #define SIZE 500
  - #define PLUS1(i) i+1
  - #define X(a,b) (a>0) ? a:b

define et tests

- On peut tester l’existance ou la valeur d’une macro :
  - #ifdef Linux
    - …. code ….
  - #endif

```
#if SIZE > 500
…. code ….
#else
…. code ….
#endif
```

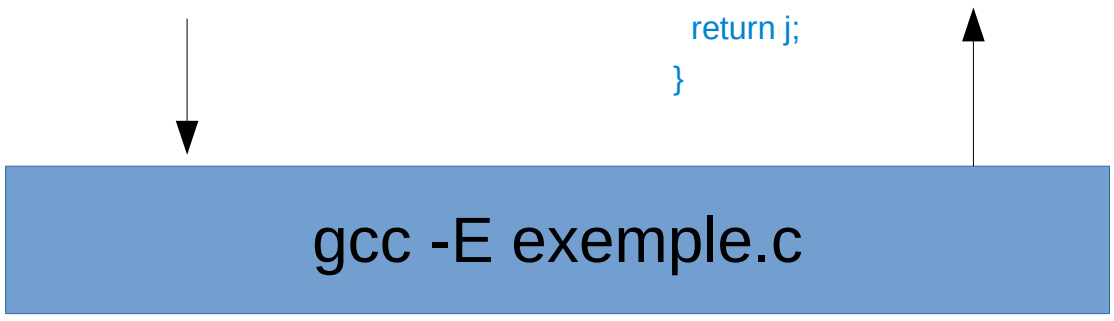
Macros : à la compilation

- 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

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

- Quelques mots sur les macros :

  substitution :

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

  mise en chaîne de caractères :

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

  Le concatenation :

  ```
  #define v(x) var##x
  ```

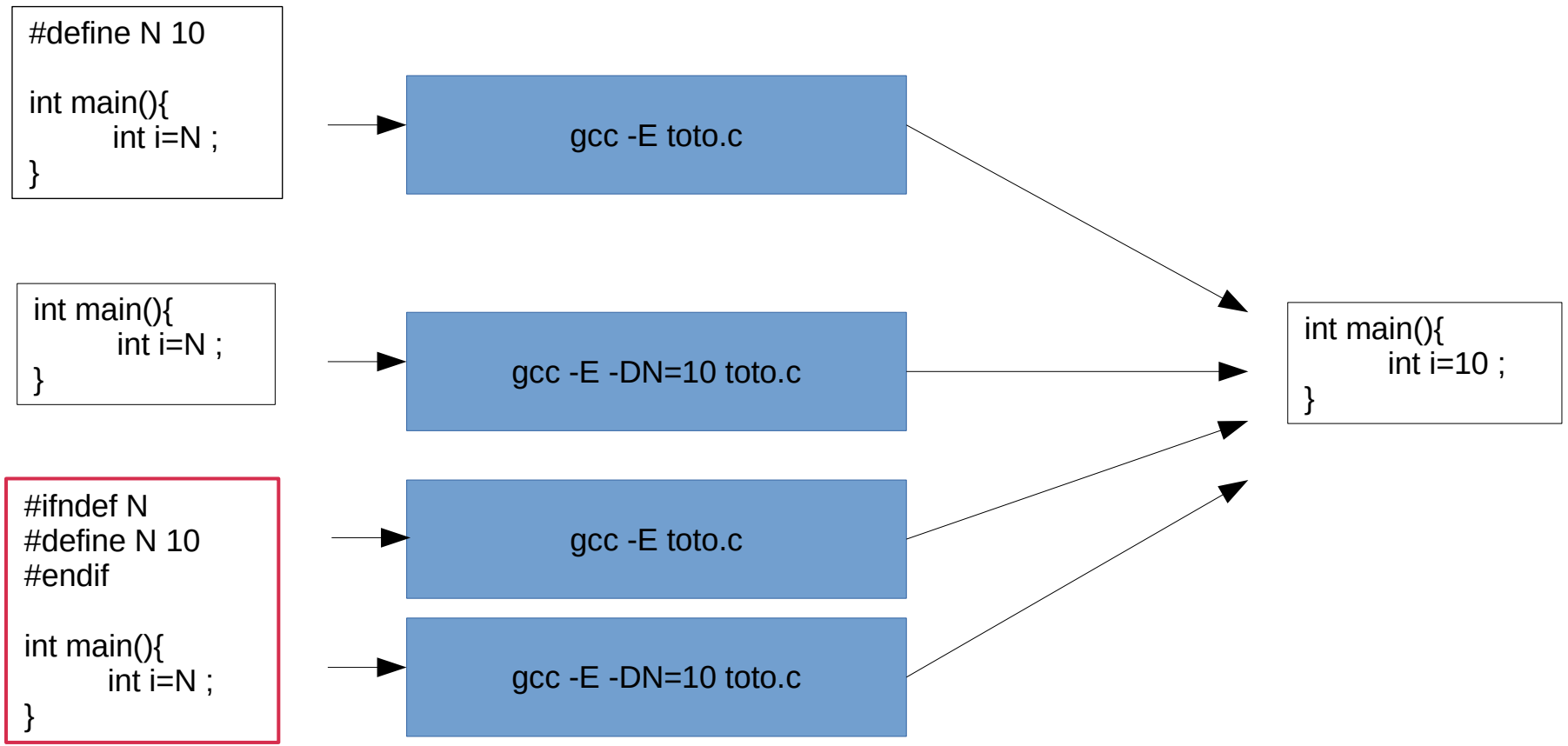
  Retrait :

  ```
  #undef
  ```

- Branchements conditionnels :

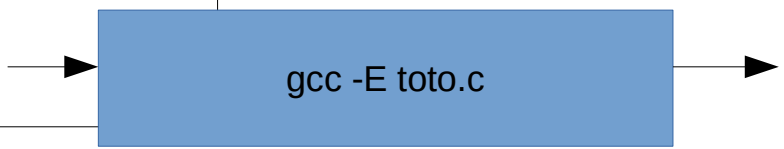
  ```
  #ifdef, #ifndef, #endif, #else, #elif, #defined
  ```

Macros : constante et bonne pratique



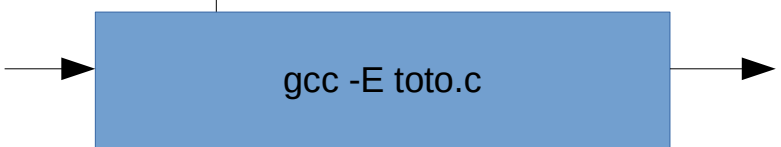
macros : concaténation et mise en chaîne

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

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




```
struct point_int {
int value ;
} ;

struct point_float{
float value ;
} ;
```

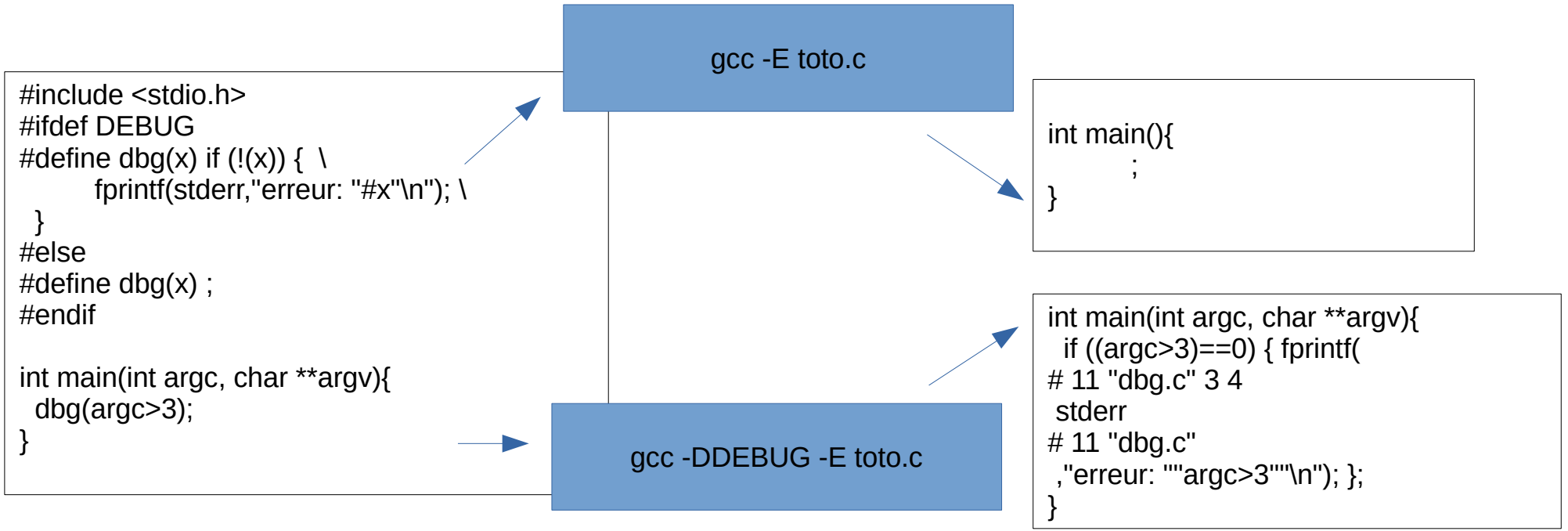
```
#define msg(x) if (x) { \
        printf(#x) ; \
  }

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 :



Exercices : gcc -E

- Dans votre répertoire dans le dépôts (celui nommé comme votre login), créer un fichier macro*.c* avec la fonction *main* suivante :

```
#include <stdlib.h>

int main(int argc, char **argv){
if (argc>0)
return EXIT_FAILURE ;
return EXIT_SUCCESS ;
}
```

- Compiler avec la commande : gcc -E macro.c -o macro2.c
- Quelles sont les valeurs de EXIT_SUCCESS et EXIT_FAILURE ?

- Il est possible d'écrire une fonction macro avec un nombre d'arguments variable :
  #define d(...) printf(__VA_ARGS_)

- Ajouter cette macro dans votre fichier et tester là (par ex : *d(«arg :%d \n»,argc) ;*)
- Modifier *d* en une macro *debug* qui affiche le message passé en argument sur la sortie erreur standard.

- Il est possible de définir une macro depuis la ligne de compilation, par exemple :
  gcc -E -DNDEBUG macro.c Dans ce cas la macro NDEBUG sera définie
  gcc -E macro.c Dans ce cas la macro n'est pas définie.

- Faîtes en sorte que votre macro *debug* affiche si la macro NDEBUG n'est pas définie et n'affiche rien si celle-ci est définie.
- Pour finir, renommer votre fichier en *debug.h* et déplacer le main de test dans un fichier *debug_test.c*
- Ajouter une macro *TRACE* qui affiche sur stderr l'argument passé à la macros avant d'exécuter celui-ci : on pourra l'utiliser comme ceci : TRACE(x=10) ; ou encore TRACE(r=algorithm(n)) ; TRACE(printf(« hello »)) ; etc...

une fois terminé : commit/push !

La compilation

Après la pré-compilation, vient la phase de compilation :

- Vérification syntaxique, en particulier sur l'usage des fonctions
- Génération de code machine (binaire, lisible si en assembleur).

- Les instructions sont transformées en code machine, regroupées par fonctions.
- Lorsqu'une fonction est utilisée, celle-ci doit avoir été déjà traitée par le compilateur (sa déclaration et/ou son implémentation).

- Le produit est donc un fichier « *objet* » (extension par défaut *.o*)
- Ce fichier objet contient :
  - Les variables globales (nommées ou anonymes)
  - Le code de machine regroupé par fonction.
  - Les fonctions « manquantes » (*printf* par exemple)

Analyse 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

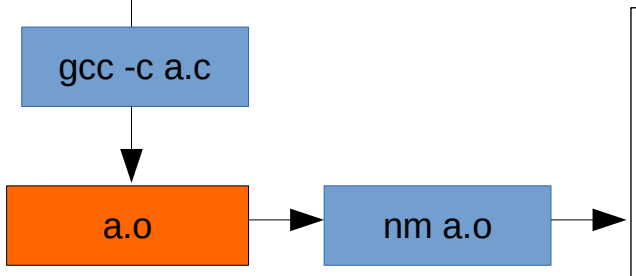
objdump -t toto.o →

```
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
```

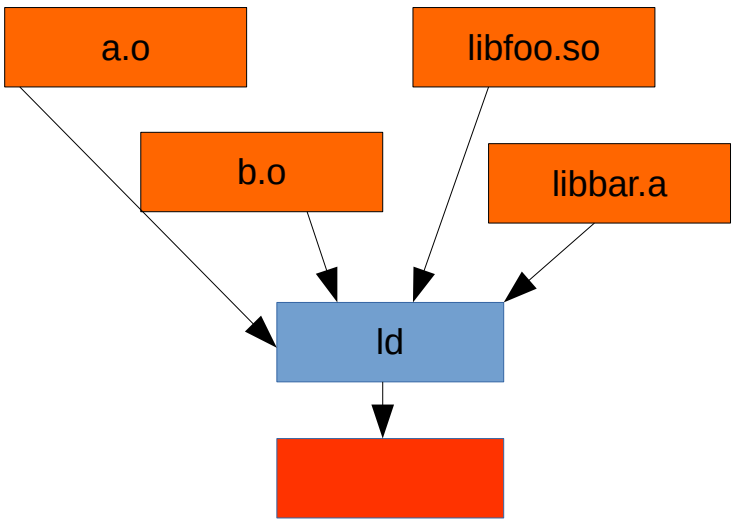
U : The symbol is undefined.
T : The symbol is in the text (code) section
C : The symbol is common. Common symbols are uninitialized data.
b : The symbol is in the uninitialized data section (known as BSS).

Exercices

- Dans un fichier example.c créer une variable globale g, une fonction void hello() qui affiche la valeur g sur le terminal et un main qui appelle cette fonction, modifie la variable g et appelle à nouveau hello
- Compiler ce fichier en objet et analyser celui-ci avec les outils nm et objdump
- Quel est l'impact de l'usage de static sur la variable globale ?
- Déclarer la variable globale extern, quelle conséquence sur le binaire produit ?
- add/commit/push

L'édition de liens

- L'édition de lien consiste à interconnecter différents « objets » pour créer une bibliothèque dynamique ou bien un binaire exécutable.
- En entrée, l'éditeur de lien (ld appelé via gcc) 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

```
> 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
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000600e20 d __JCR_END__
0000000000600e20 d __JCR_LIST__
                 w _Jv_RegisterClasses
0000000000400580 T __libc_csu_fini
0000000000400510 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000004004f6 T main
0000000000400470 t register_tm_clones
0000000000400400 T _start
0000000000601038 D __TMC_END__
```

segfault, pourquoi ?

Parce que f est une fonction dans a.c mais une variable globale dans b.c

L'édition de liens

- Aucune cohérence de type des symboles n'est effectuée lors de l'édition (ni de nature, ni de type)
  - > 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
```

Exercices

1) Extraire la fonction hello et g du fichier example.c (le main reste dans ce fichier) et mettez ceux-ci dans un fichier algo.c
2) Compiler les deux fichiers en objets (gcc -c), résoudre les soucis si nécessaire.
3) Assembler les deux objets pour produire un exécutable que vous testez
4) Modifier l'implémentation de hello pour que celle-ci renvoie un entier et prenne un paramètre qu'elle affichera, ne modifier pas example.c
5) Répéter les étapes 2 et 3.
6) « git status » vous invite à ajouter les .o . Pour dire à git d'ignorer ces fichiers créer un fichier .gitignore dans lequel vous mettrez une expression régulière par ligne : vérifier que cela fonctionne puis ajouter ce fichier au dépôt.

L'importance des headers !

- Les *headers* servent de « contrats » entre différentes entités compilées séparément (modules, bibliothèques).
- Les *headers* ne contiennent pas d'implémentation! (attention aux globales : elles doivent être « extern » dans le header)
- Un *header* est un fichier .h qui contient :
  - la déclaration de fonctions
  - la déclaration de structures
  - les macro
  - les déclaration de « globales » à venir (avec extern)

Les headers

- La déclaration de fonction :
  - ex : int fonction_foo (char) ;
  - Le nommage des paramètres n'est pas obligatoire.
- Les structures : plusieurs options
  - Selon que l'on souhaite donner accès au champs de la structure, on pourra soit pré-déclarer soit déclarer :

point.h

```
struct Point ;
```

point.c

```
struct Point{
        int x ;
        int y ;
}
```

seul point.c peut créer des struct Point et accéder aux champs. Les autres doivent obligatoirement utiliser des pointeurs.

point.h

```
struct Point{
        int x ;
        int y ;
}
```

tout le monde peut allouer des structures et accéder au champs

Les headers

- L'avantage de la pré-déclaration de structure est de masquer l'implémentation et de pouvoir modifier l'implémentation sans impacter les utilisateurs (on en reparle plus tard avec les API).
- Les macros (déjà vu). Pour éviter les inclusions multiples, on utilisera la technique :

```
#ifndef ID
#define ID
….
#endif
```

Remarque: On pourra utiliser

#pragma once

mais cela ne fait pas parti du standard et peut ne pas être géré par tous les compilateurs.

Les headers

- Les globales, c'est à dire les variables dont on souhaite que tout le monde puisse y accéder.
- On évitera de telle variable (effet de bord…).
- Les variables globales doivent être déclarées dans le « .c » car un symbole est généré pour cette variable.
- Afin d'indiquer que ce symbole existera (édition de liens), on le déclare en « extern » dans le .h :

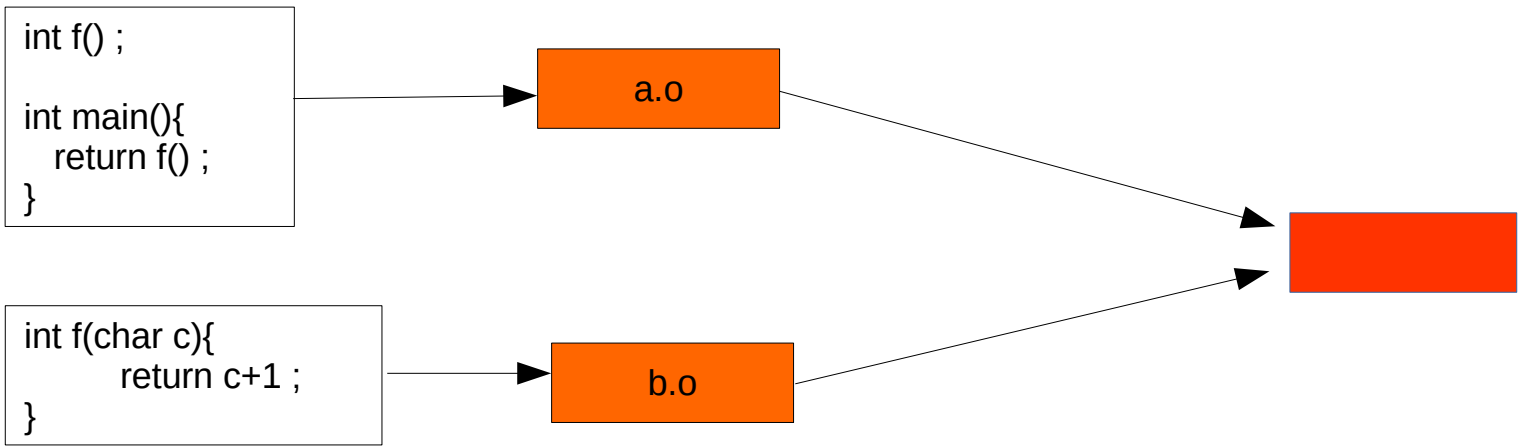
```
extern int v ;                int v ;
```

- Ainsi l'utilisateur du .h aura le symbole « v » comme manquant dans le .o. Lors de l'édition de lien, il y aura association entre le symbole v manquant et le symbole v fournit.

Les headers

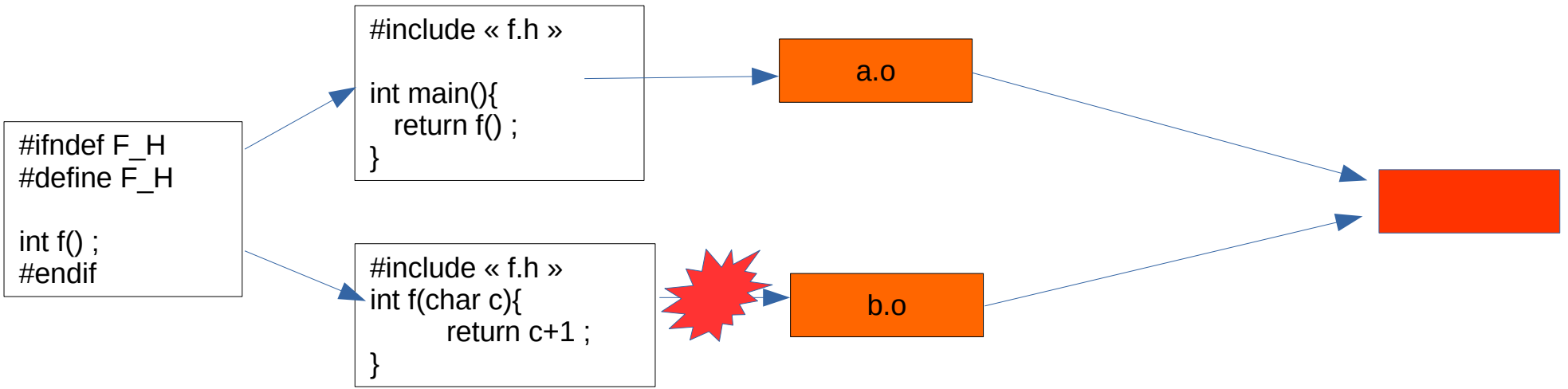
- Après l'étape de pré-compilation, tous les header inclus (directement ou indirectement) sont regroupés en un seul source.
- Si un macro est définie plusieurs fois, il y a erreur dans la pré-compilation.
- Si une structure, globale ou fonction est définie plusieurs fois, il y a erreur lors de la compilation, d'où l'importance de se protéger contre les inclusions multiples.

Les headers par l'exemple



Les deux sources compilent sans problème.
Il n'y a pas de header, la cohérence n'est pas garantie, le comportement final n'est pas prédictible.

Les headers par l'exemple



La code de la fonction f dans le fichier b.c n'est pas cohérent avec la déclaration dans f.h => erreur à la compilation.

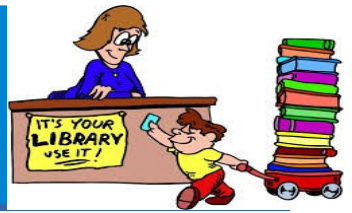
les headers: qq règles

- Toujours protéger contre l’inclusion multiple
- Mettre le minimum nécessaire d’include dans le .h
- Ne mettre que les fonctions faisant parties de l’API du module
- Pas de globales directement: extern

Exercices

- Écrire le fichier algo.h et modifier les fichiers sources en conséquences.
- Compiler en objet et observer l'impossibilité de compiler (normalement).
- Modifier pour que cela fonctionne.
- Regrouper dans un script shell compile.sh les commandes pour :
  - Compiler algo.c et example.c en objet
  - Regrouper ces objets au sein d'un exécutable
- Faites en sorte que les arguments passés au compilateur pour la phase de pré-compilation/compilation soient définis dans une variable d'environnement CFLAGS, et que ceux utilisés lors de l'édition de liens soient dans une variable LDFLAGS
- Incorporé l'utilisation de debug.h dans algo.c et tester le bon fonctionnement de CFLAGS pour activer/desactiver les messages de debug.
- add/commit/push

Les bibliothèques statiques



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

```
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
00000000004004f6 T main
```

Les bibliothèques dynamiques

- Une bibliothèque dynamique est binaire qui regroupe un ensemble de symboles.
- Les binaires (.o) **doivent avoir été produit** avec l'option *-fPIC*
- 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)
```

Les bibliothèques dynamiques : utilisation

- l'option *-lfoo* (pas d'espace entre le l et le nom de la bibliothèque) demande à l'éditeur de lien de chercher un fichier nommé libfoo.so
- Par défaut, ld cherche dans des répertoires « standards » (/usr/lib par ex).
- L'option *-Lpath* permet d'ajouter le répertoire path dans les répertoires où chercher une bibliothèque.
- *path* n'est pas conservé dans le binaire, seul le lien vers libfoo.so est présent. On indique au système des répertoires de recherche additionnels en utilisant la variable d'environnement LD_LIBRARY_PATH :

```
export LD_LIBRARY_PATH=/home/user/libs:/home/user/extra
```

- attention à l'usage de chemin relatif pour *path*

Exercices

- Créer une bibliothèque statique libalgo.a
- Créer l'exécutable example_static à partir des fichier example.o et libalgo.a
- Créer la bibliothèque dynamique libalgo.so pour cela vous créerez l'objet *algo_fpic.o*
- Créer l'exécutable example_dyn à partir des fichier example.o et libalgo.so
- Donner la ligne de commande permettant d'exécuter le programme example_dyn
- Ajouter l'ensemble des commandes ci-dessus dans votre script shell.
- Pour chaque fichier modifié, indiquer ce qui doit être recompilé : indiquer cela en commentaires dans votre script shell.
- Enregistrer votre travail dans le dépôt : commit/push