

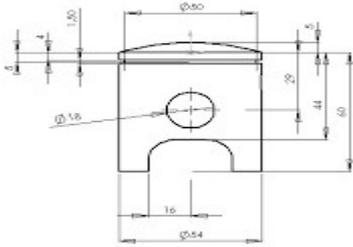
PG120: Outils pour la programmation C

Julien Allali / allali@enseirb-matmeca.fr



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



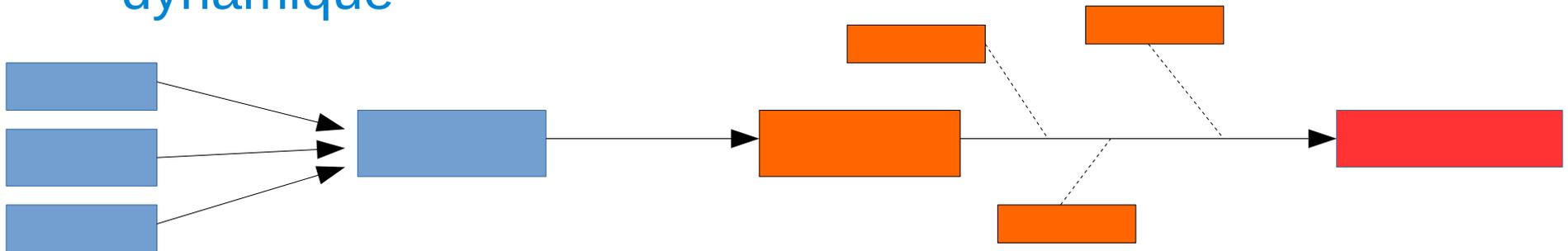
binaire
Hello.o



exécutable
Hello

Fichiers sources et compilation

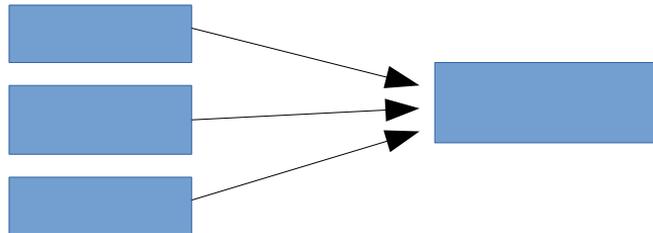
- Le compilateur est un programme qui lit des sources et produit un binaire possiblement exécutable
- La compilation comporte trois étapes :
 - Le pré-processing : sources => source
 - La compilation : source => binaire
 - L'édition de lien : binaires => exécutable, bibliothèque dynamique



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 les répertoires systèmes.
- Dès que le fichier foo.h est trouvé, alors la recherche s'arrête et le contenu du fichier est lu.

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;  
}
```



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;
}
```

gcc -E hello.c

```
# 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
```

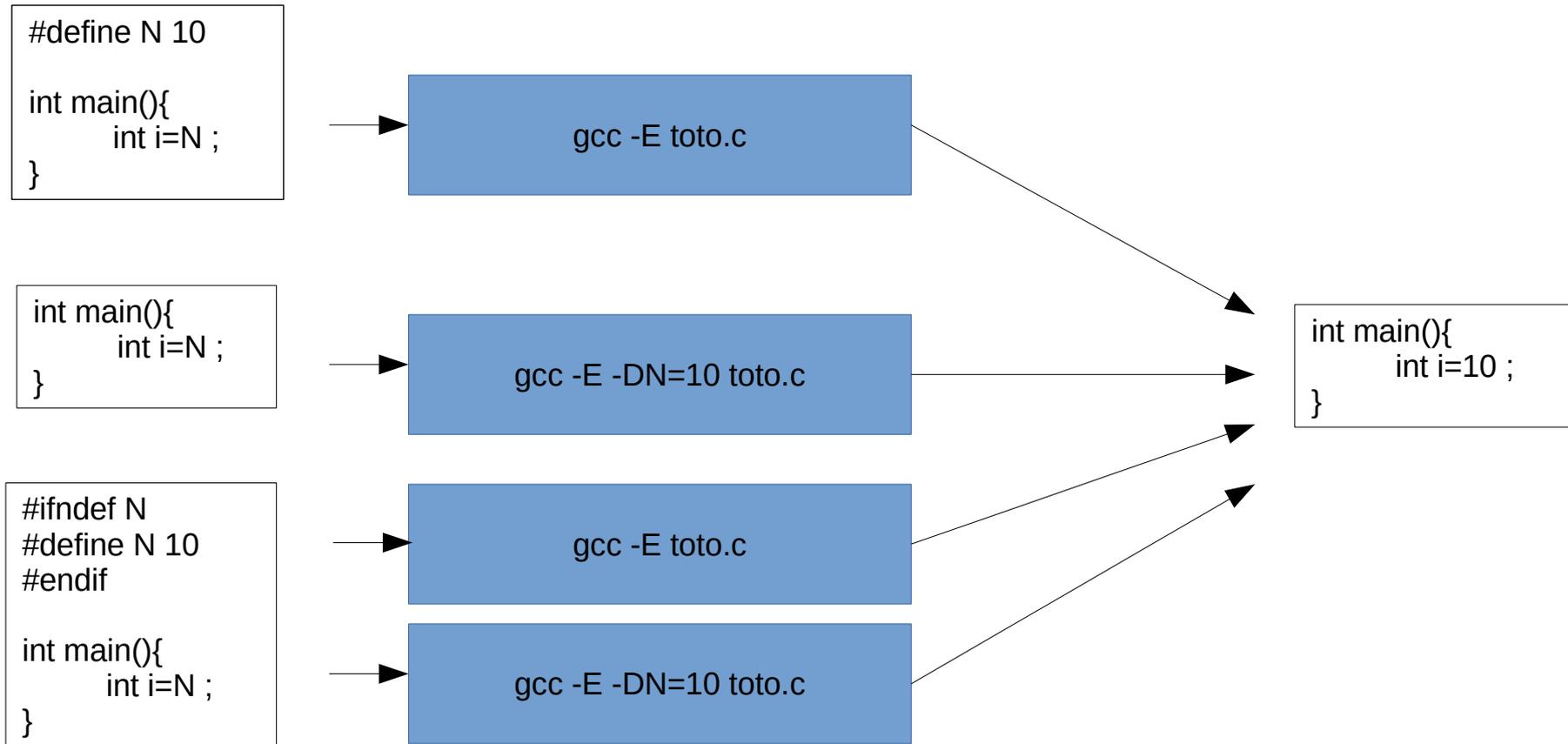
Retrait :

```
#undef
```

- Branchements conditionnels :

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

Macros : substitution



macros : concaténation et mise en chaîne

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

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



gcc -E toto.c



```
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) ;\
}
```



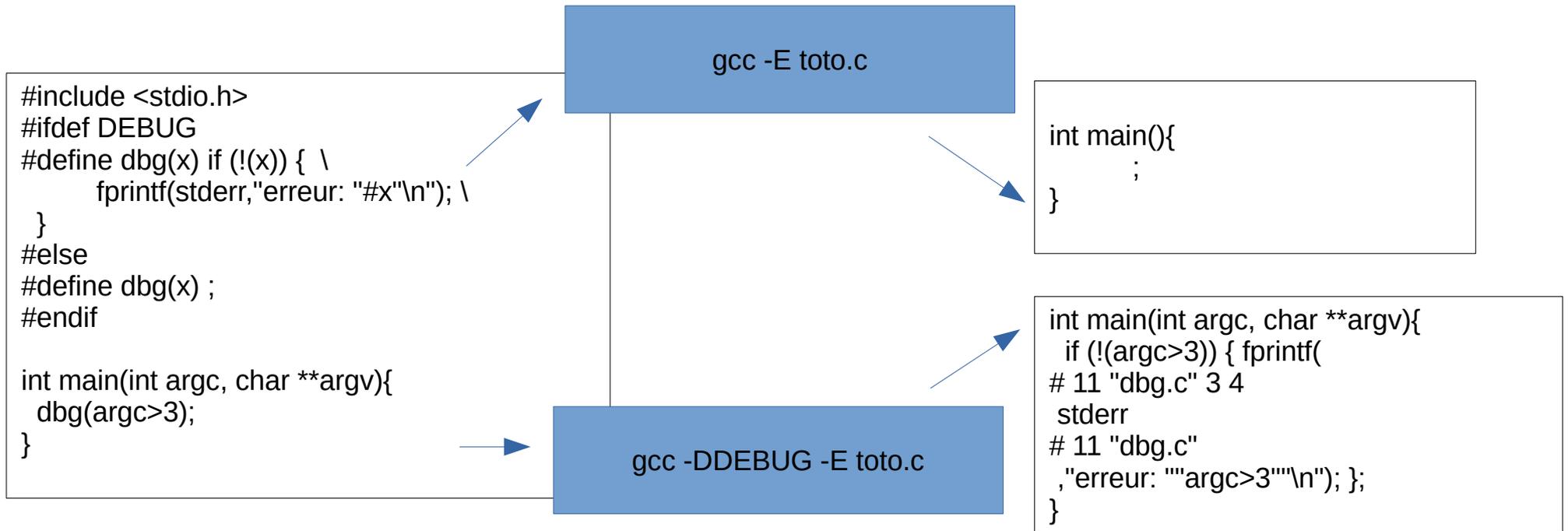
gcc -E toto.c



```
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

- Créer un répertoire `pg120`
- Dans ce répertoire créer un fichier `debug.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 debug.c -o debug2.c`
- **Quelles sont les valeurs de** `EXIT_SUCCESS` **et** `EXIT_FAILURE` ?

Exercices : macros

- Il est possible d'écrire une fonction macro avec des arguments variables :
`#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 debug.c` Dans ce cas la macro `NDEBUG` sera définie
`gcc -E debug.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, renommez votre fichier en `debug.h` et déplacez le main de test dans un fichier `debug_test.c`

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).

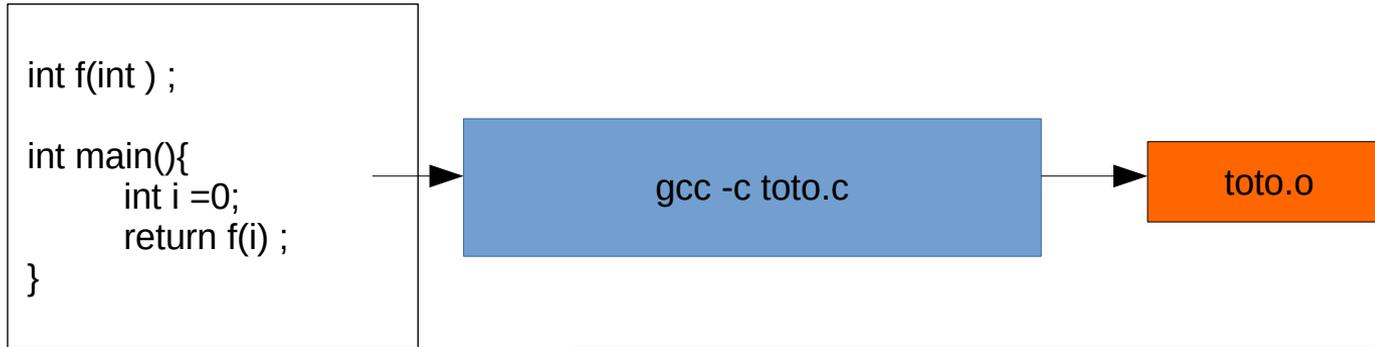


- Les instructions sont transformées en code machine, regroupées par fonctions.
- Lorsqu'une fonction est utilisée, celle-ci doit avoir été déjà traitée par le compilateur (sa déclaration et/ou son implémentation).
- Le produit est donc un fichier « *objet* » (extension par défaut *.o*)
- Ce fichier objet contient :
 - Les variables globales
 - 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



objdump -t toto.o

toto.o: file format elf64-x86-64

SYMBOL TABLE:

0000000000000000	df	*ABS*	0000000000000000	toto.c
0000000000000000	d	.text	0000000000000000	.text
0000000000000000	d	.data	0000000000000000	.data
0000000000000000	d	.bss	0000000000000000	.bss
0000000000000000	d	.note.GNU-stack	0000000000000000	.note.GNU-stack
0000000000000000	d	.eh_frame	0000000000000000	.eh_frame
0000000000000000	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) ;
}
```

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 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).

- 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 »

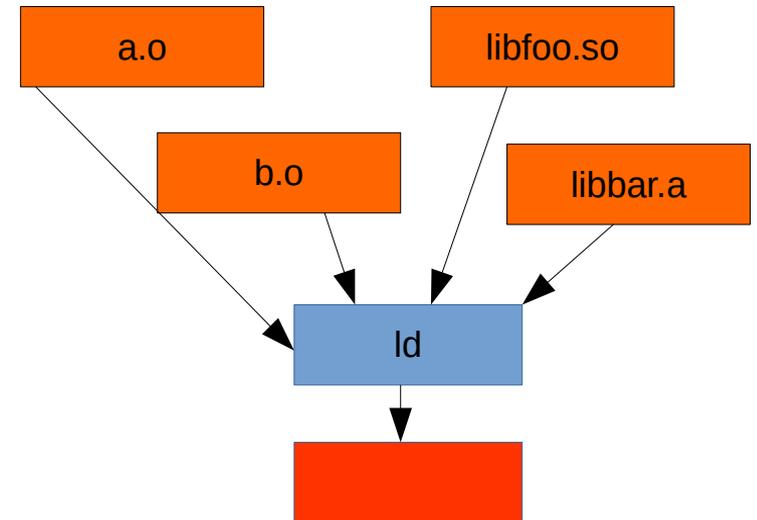
Exercices

- 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 ?

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



```
> 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

```
000000000601038 B __bss_start
000000000601038 b completed.7259
000000000601028 D __data_start
000000000601028 W data_start
000000000400430 t deregister_tm_clones
0000000004004b0 t __do_global_dtors_aux
000000000600e18 t __do_global_dtors_aux_fini_array_entry
000000000601030 D __dso_handle
000000000600e28 d _DYNAMIC
000000000601038 D _edata
000000000601040 B _end
00000000060103c B f
000000000400584 T _fini
0000000004004d0 t frame_dummy
000000000600e10 t __frame_dummy_init_array_entry
0000000004006b8 r __FRAME_END__
000000000601000 d _GLOBAL_OFFSET_TABLE_
          w __gmon_start__
0000000004003a8 T _init
000000000600e18 t __init_array_end
000000000600e10 t __init_array_start
000000000400590 R _IO_stdin_used
          w _ITM_deregisterTMCloneTable
          w _ITM_registerTMCloneTable
000000000600e20 d __JCR_END__
000000000600e20 d __JCR_LIST__
          w _Jv_RegisterClasses
```

→ 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
 - > 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 `exemple.c` et mettez ceux-ci dans un fichier `hello.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 `exemple.c`
- 5) Répéter les étapes 2 et 3.

L'importance des headers !

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

Les headers

- La déclaration de fonction :
 - ex : `int fonction_foo (char) ;`
 - Le nommage des paramètres n'est pas obligatoire.
- Les structures :
 - 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 ;
```

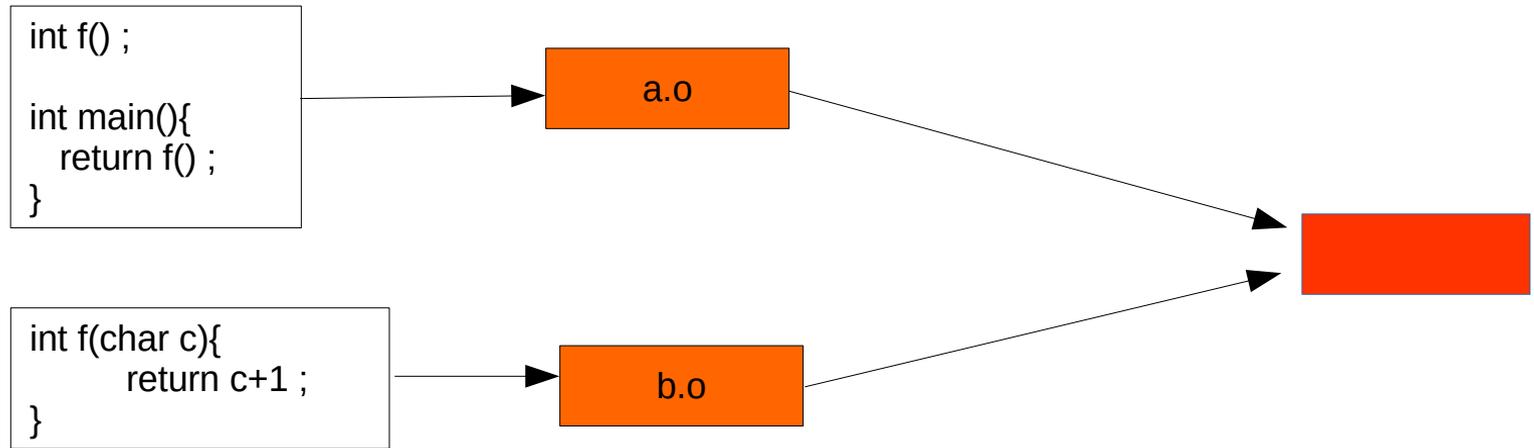
```
int v ;
```

- 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.

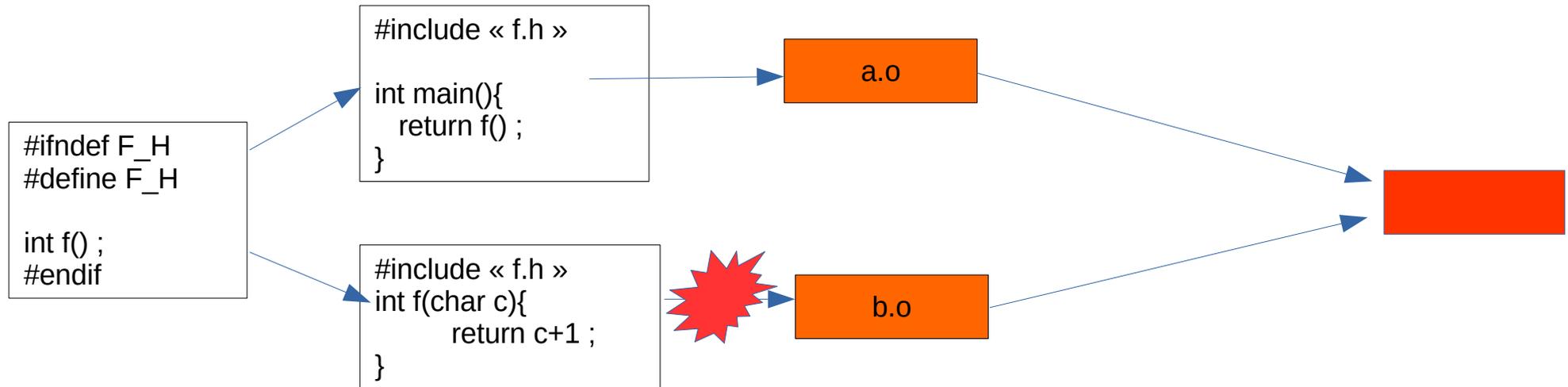
Les headers par l'exemple



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 `hello.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 `hello.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 `hello.c` et tester le bon fonctionnement de `CFLAGS` pour activer/désactiver les messages de debug.

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
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 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 _
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)
```



Exercices

- **Créer une bibliothèque statique** `libhello.a`
- **Créer l'exécutable** `example_static` à partir des fichiers `example.o` et `libhello.a`
- **Créer la bibliothèque dynamique** `libhello.so`
- **Créer l'exécutable** `example_dyn` à partir des fichiers `example.o` et `libhello.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.