Programmation système

Compilation séparée

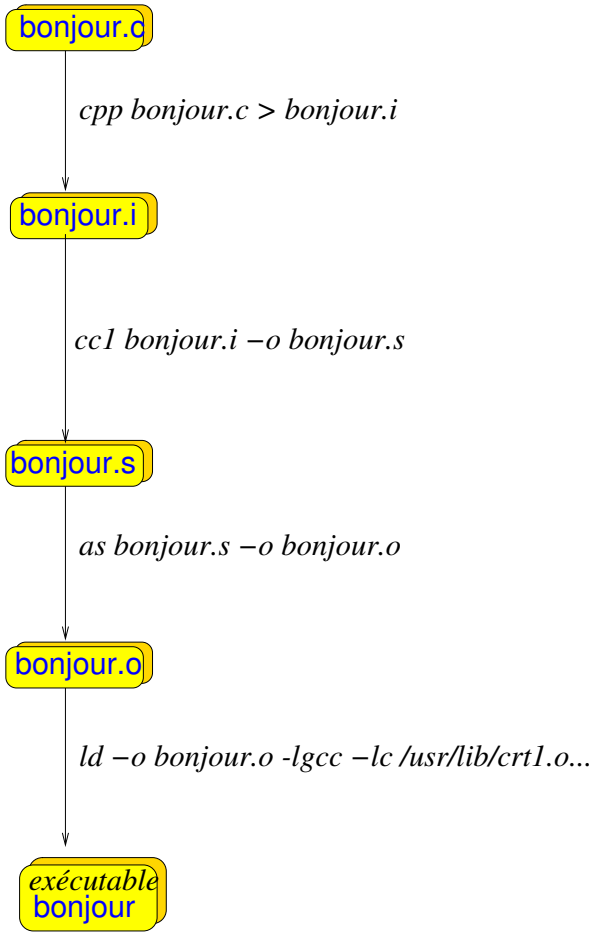
Tuyêt Trâm DANG NGOC
<dntt@dept-info.u-cergy.fr>

Université de Cergy-Pontoise

Phases de compilations

Compilation d'un programme :

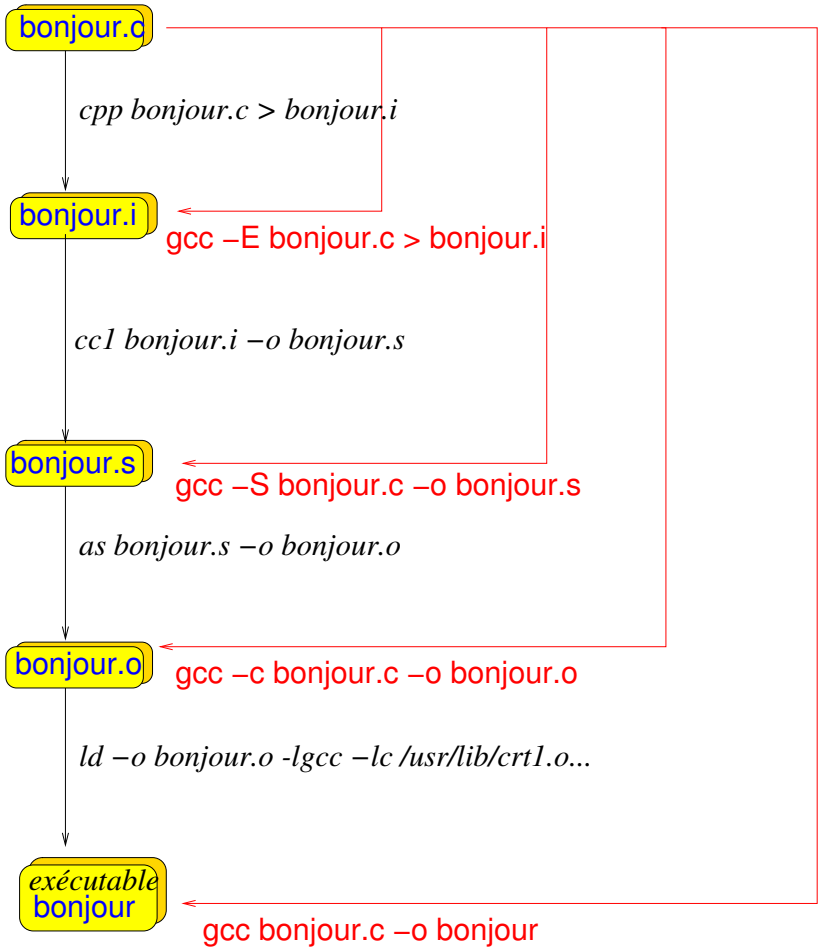
- ensemble de traitements successifs sur un fichier source
  1. pré-traitement
  2. compilation
  3. assemblage
  4. édition de lien
- but est de générer un fichier exécutable par la machine.



Phases de compilations

Compilation d'un programme :

- ensemble de traitements successifs sur un fichier source
  1. pré-traitement
  2. compilation
  3. assemblage
  4. édition de lien
- but est de générer un fichier exécutable par la machine.



Pré-traitement (*pre-processing*)

- inclus les fichiers d'en-tête
- enlève les commentaires
- remplace les macros par leurs valeurs

```
cpp fichier.c > fichier.i
```

ou directement depuis le fichier c

```
gcc -E fichier.c > fichier.i
```

Le fichier résultant porte souvent l'extension `.i`

Pré-traitement : Fichiers toto.c et toto.i

```c
#include <stdio.h>
#include "calcul.h"

/* Programme principal */
int main () {
    float resultat ;
    float r = 12 ;
    resultat = calcul_surface (r) ;
    printf ("Surface = %f\n",
                        resultat) ;
    return 0 ;
}
```

```c
# 748 "/usr/include/stdio.h" 2 3 4
extern int fileno (FILE *__stream) ;
extern void perror (__const char *__s);
extern int printf (__const char *__restri
# 1 "calcul.h" 1
float calcul_aire (float rayon) ;
float calcul_surface (float rayon) ;
# 4 "toto.c" 2

int main () {
    float resultat ;
    float r = 12 ;
    resultat = calcul_surface (r) ;
    printf ("Surface = %f\n", resultat) ;
    return 0 ;
}
```

Compilation

Le programme C obtenu après pré-traitement produit un fichier texte contenu du code en langage d'assemblage spécifique à la machine

- sur laquelle le code est compilé (compilation native)
- pour une autre machine dans le cas d'une compilation croisée

```
cc1 fichier.i -o fichier.s
```

ou directement depuis le fichier c

```
gcc -S fichier.c -o fichier.s
```

Le fichier résultant porte souvent l'extension `.s`

Fichier toto.s

```
        .file   "toto.c"
        .section        .rodata
.LC1:
        .string "Surface = %f\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $0x41400000, %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, (%esp)
        call    calcul_surface
        fstps   -4(%ebp)
        flds    -4(%ebp)
        fstpl   4(%esp)
        movl    $.LC1, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"
        .ident  "GCC: (GNU) 3.3.5 (Debian
```

Assemblage

Le code assembleur est ensuite assemblé pour générer du code machine. Le fichier produit est appelé fichier objet.

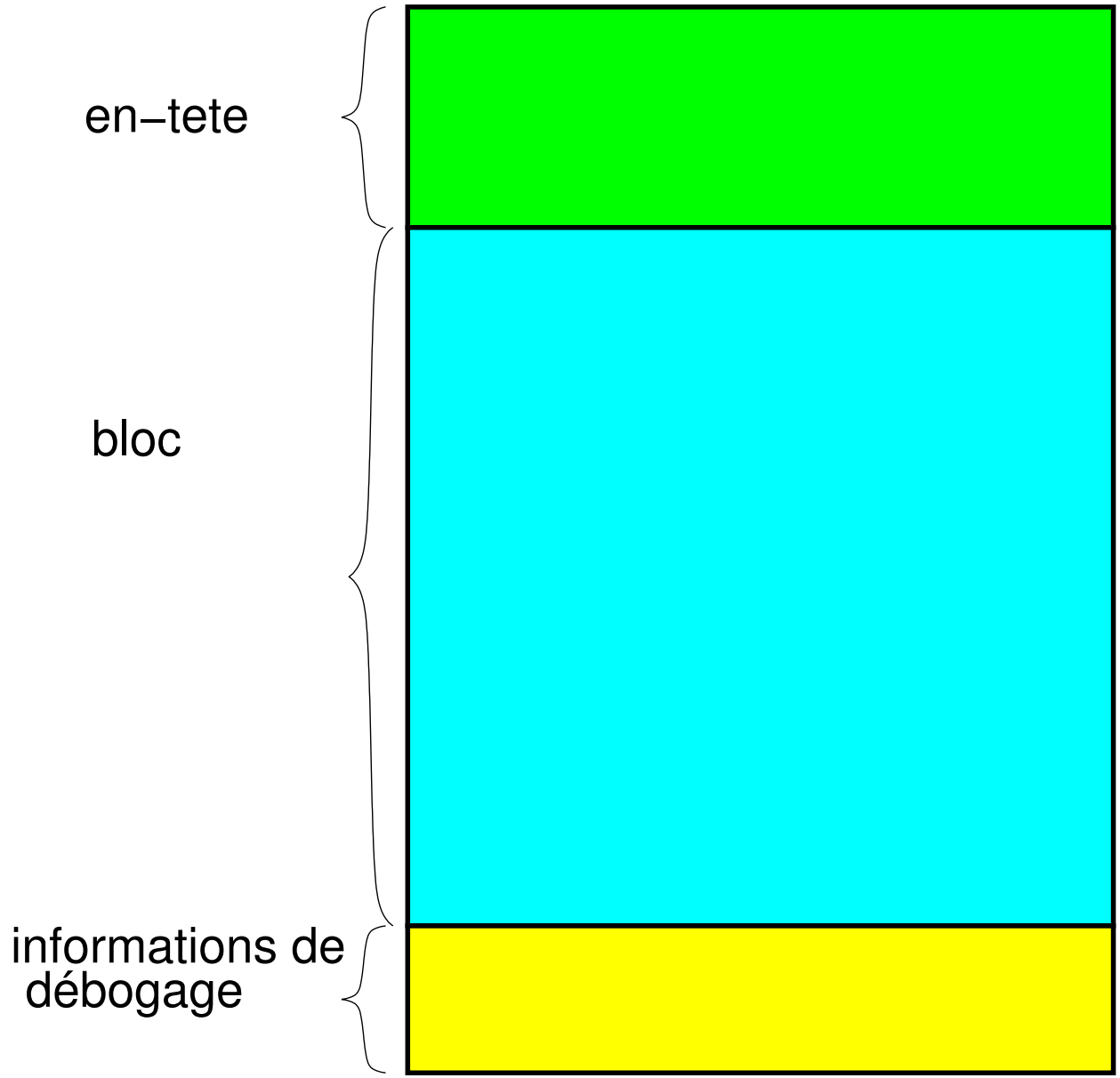
```
as fichier.s -o fichier.o
```

ou directement depuis le fichier .c

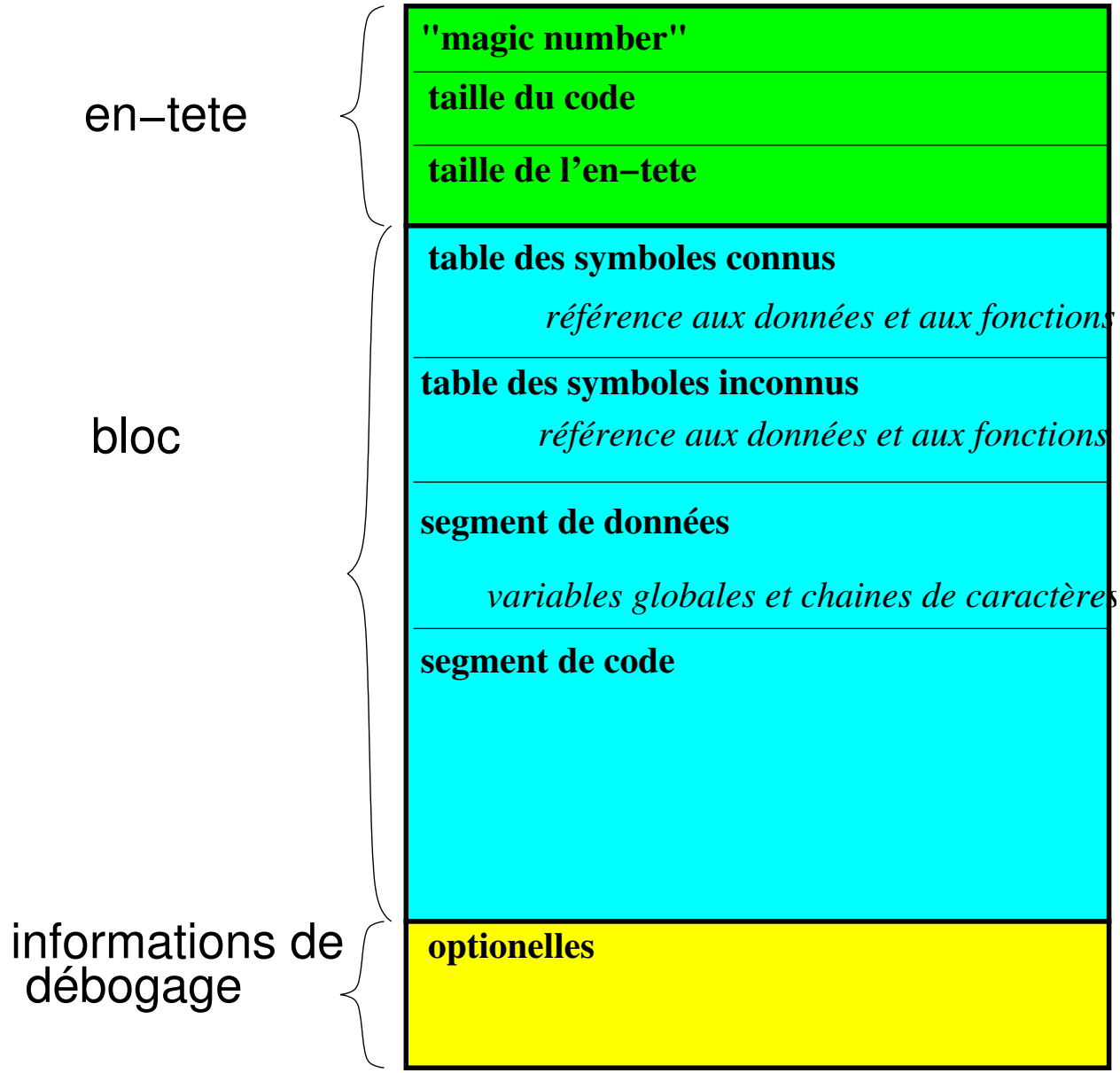
```
gcc -c fichier.i -o fichier.s
```

Le fichier résultant porte souvent l'extension `.o`

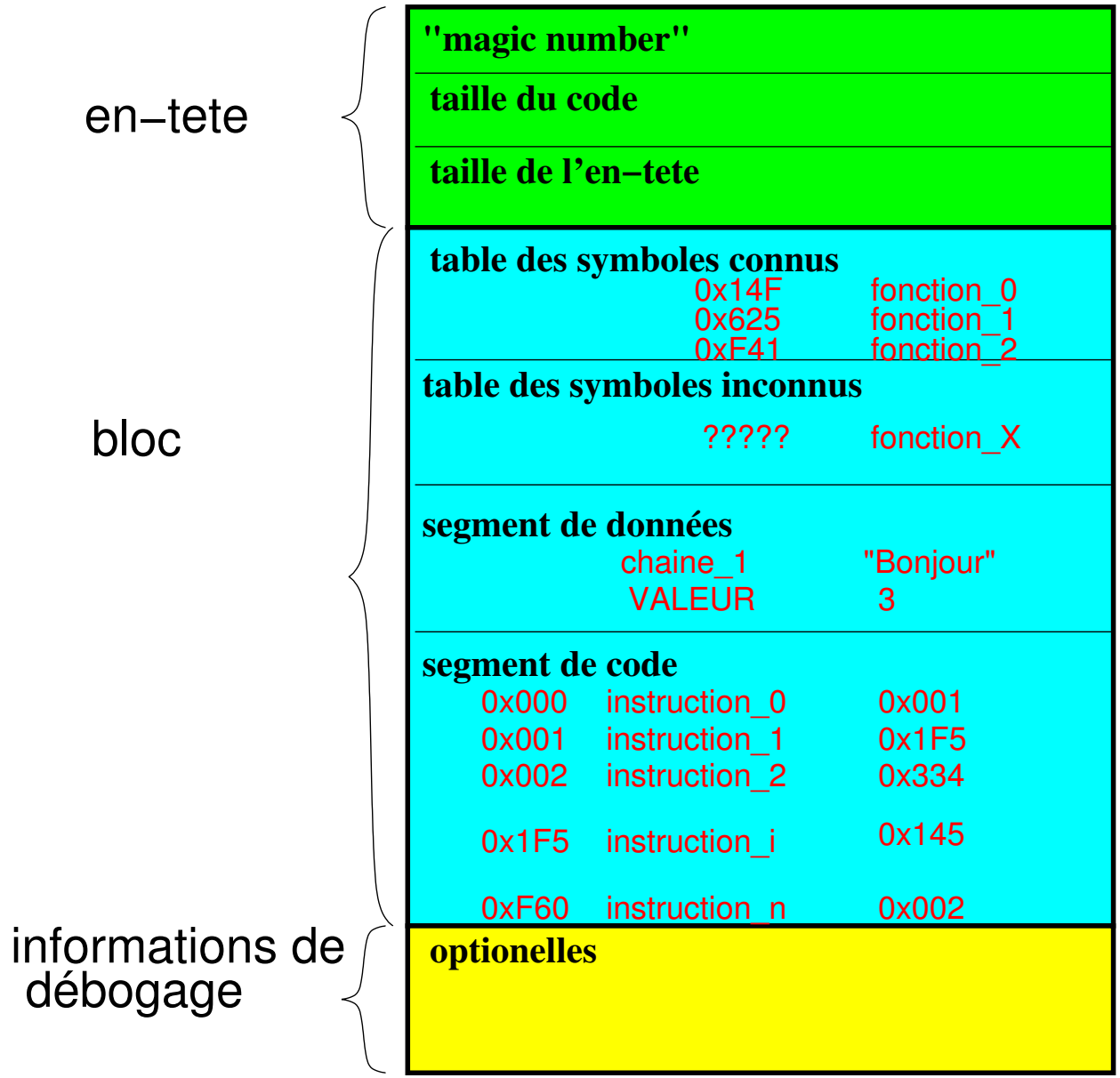
Structure d'un fichier objet



Structure d'un fichier objet



Structure d'un fichier objet



Edition de lien

L'édition de liens prend un ensemble de fichiers objets pour produire un programme exécutable.

- rassemble les fichiers objets du programme
- rassemble les fichiers objets du "système"
- extrait les fonctions des bibliothèques
  - les bibliothèques statiques
  - les bibliothèques dynamiques
- résoud les problèmes d'adressage

```
ldd fichier1.o fichier2.o fichier3.o fichiers_systeme.o -o executable
```

ou directement depuis les fichiers .c

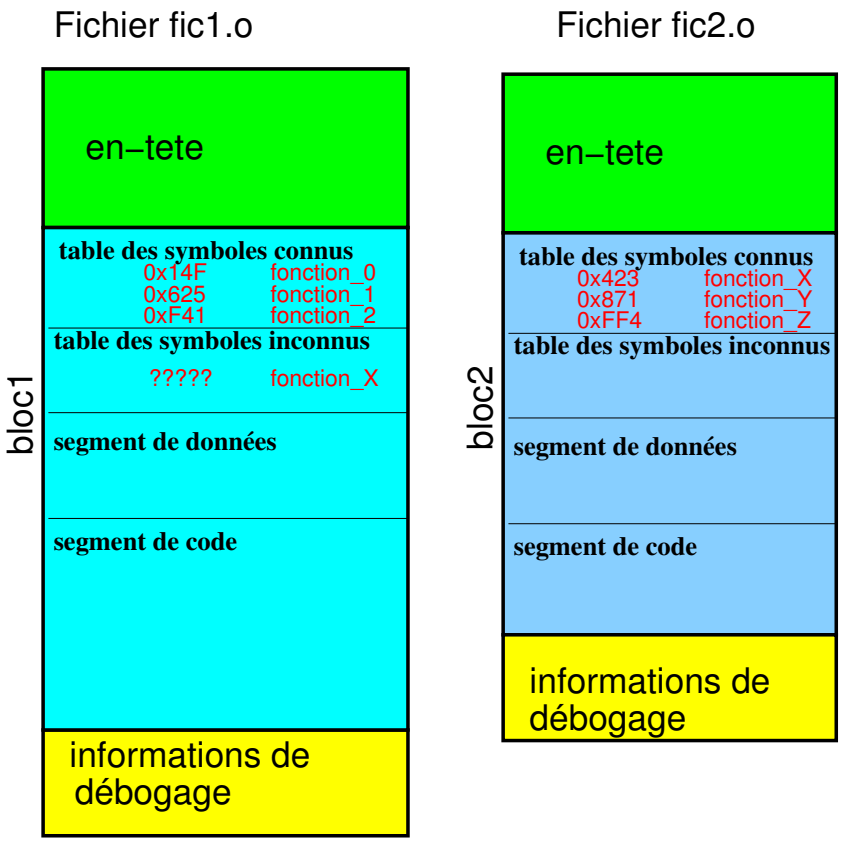
```
gcc fichier1.c fichier2.c fichier3.c -o executable
```

ou depuis les fichiers .o

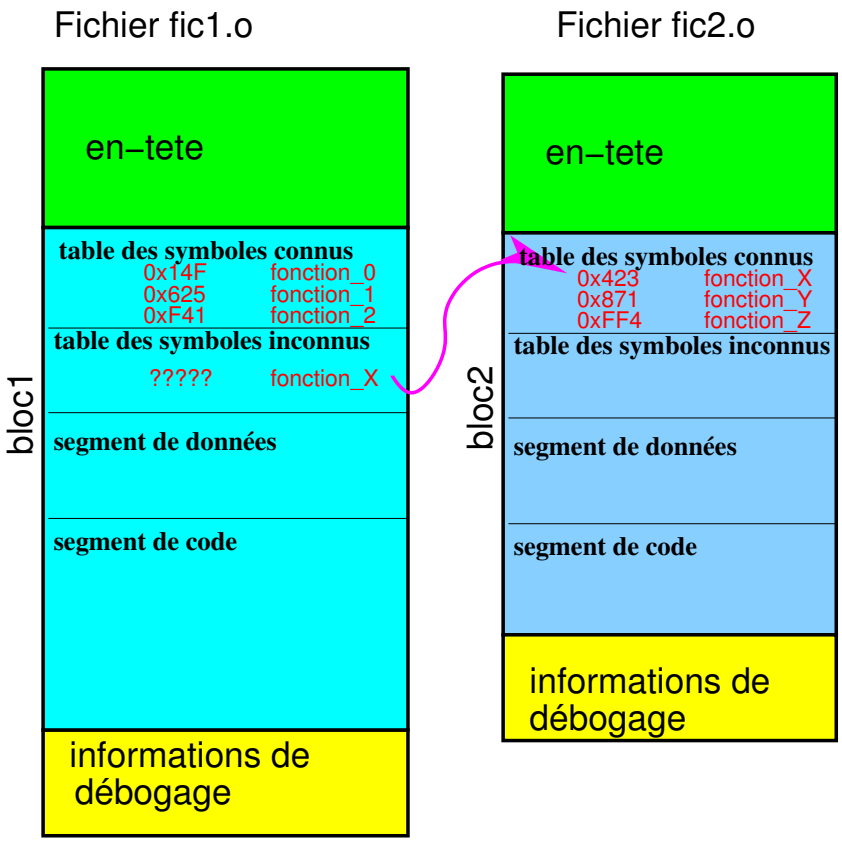
```
gcc fichier1.o fichier2.o fichier3.o -o executable
```

Le fichier exécutable résultant porte par défaut le nom `a.out`

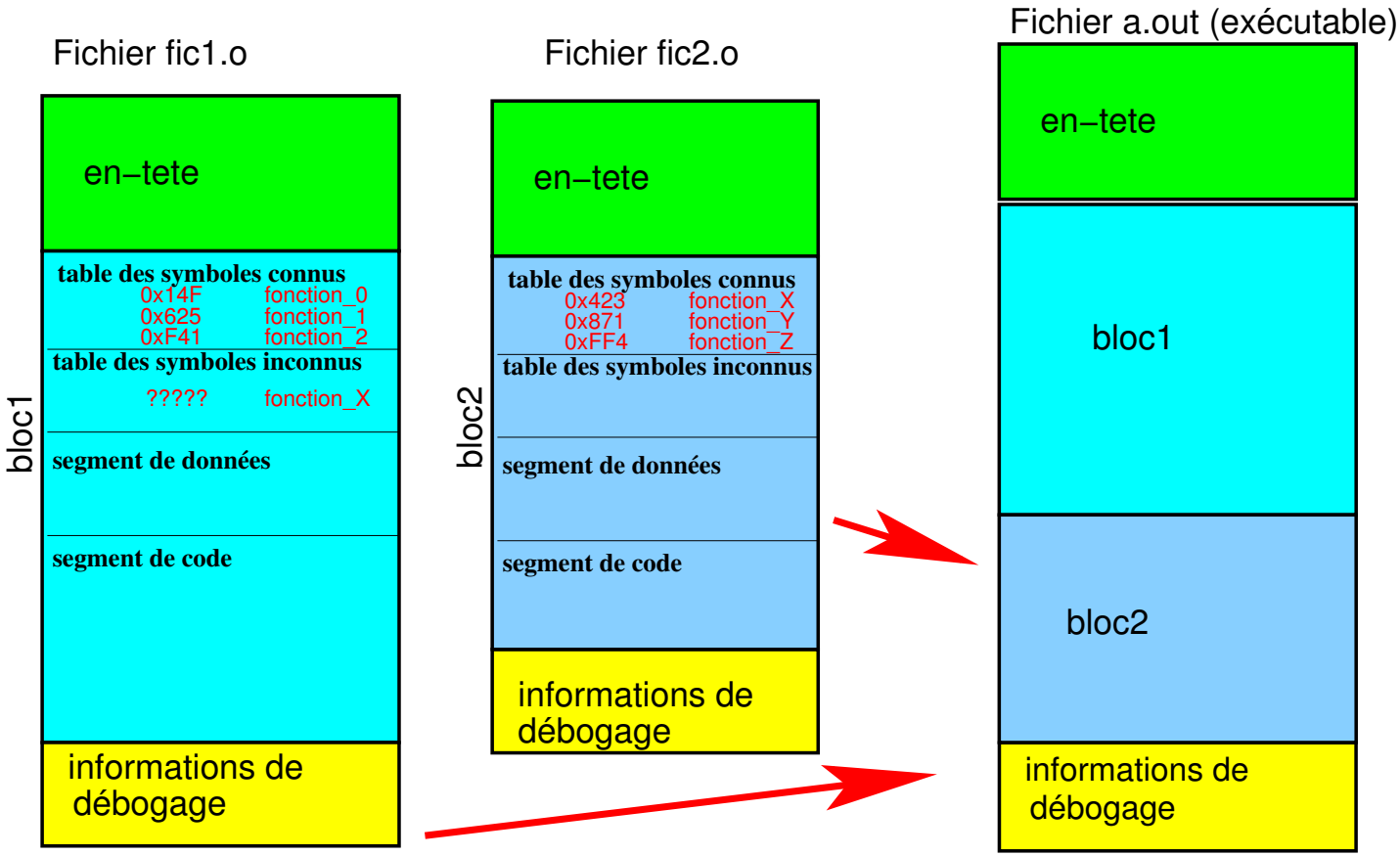
Edition de lien



Edition de lien

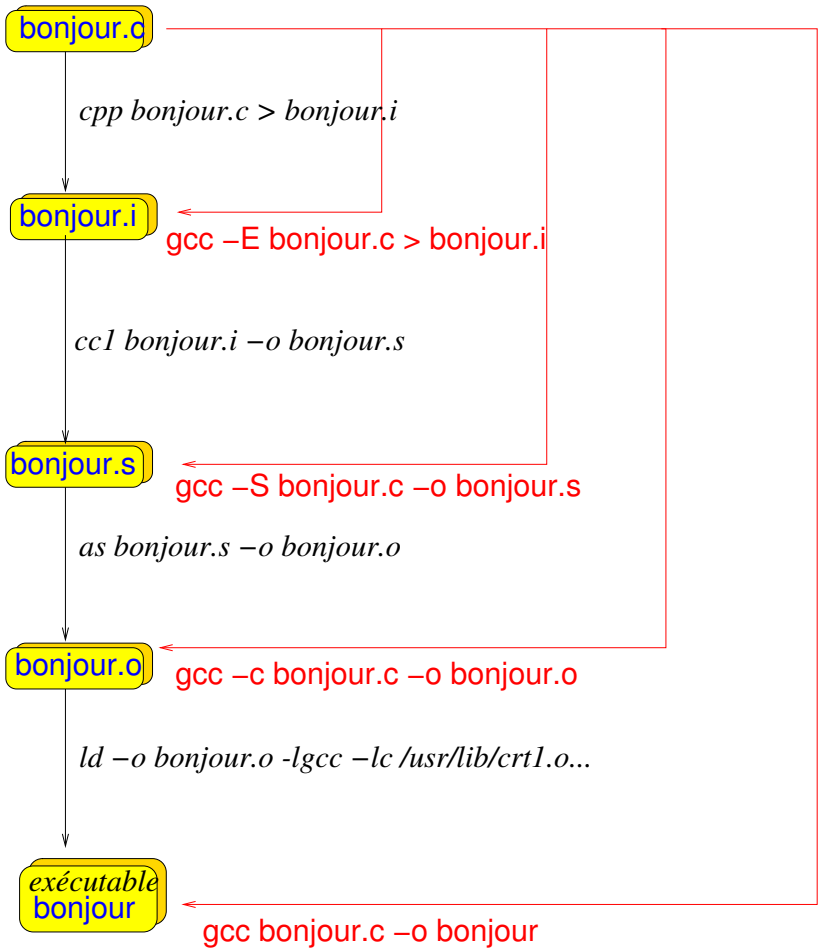


Edition de lien

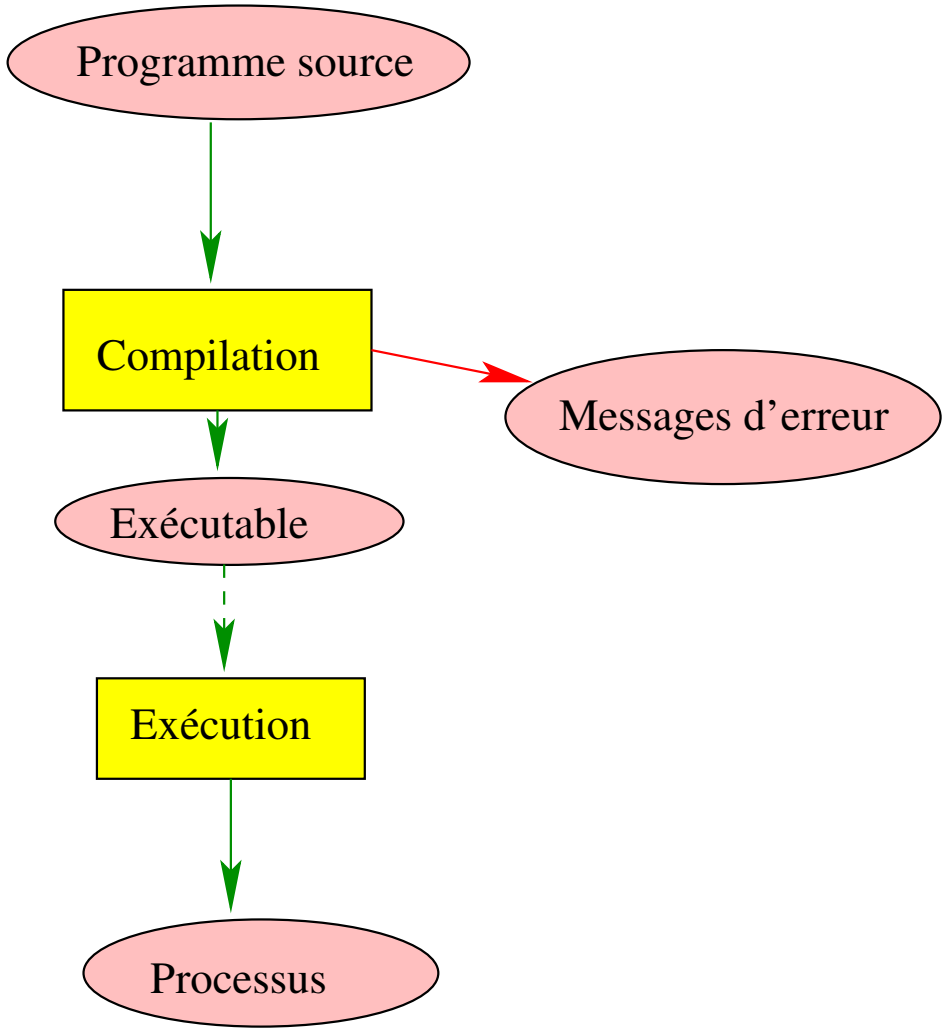


Récapitulatif : étapes d'une compilation

1. pré-traitement :
   cpp / **gcc -E**
2. compilation :
   cc1/ **gcc -S**
3. assemblage :
   as/ **gcc -c**
4. édition de lien :
   ld/ **gcc**



Exécution


Programme source
Compilation
Messages d'erreur
Exécutable
Exécution
Processus

Bibliothèque

Une bibliothèque est un ensemble de fichiers .o. Elle peut être liée statiquement ou dynamiquement à l'exécutable.

- bibliothèque statique :
  - contenu de la bibliothèque ajouté dans le fichier exécutable une fois pour toute.
  - Exécutable self-contained : tout le code est inclus dans le fichier exécutable.
  - Inconvénient : mise à jour de la bibliothèque $\Rightarrow$ recompilation du fichier source, à la charge de l'utilisateur.
- bibliothèque dynamique :
  - exécutable ne contient qu'un lien vers cette bibliothèque.
  - exécutable de plus petite taille.
  - code de la bibliothèque chargé au moment de l'exécution (runtime).
  - Avantage : pas de duplication du code d'une même bibliothèque entre les exécutables
  - numéros de versions utilisées pour gérer la compatibilité entre les versions d'une bibliothèque.

Bibliothèque statique

Création d'une bibliothèque statique

```
gcc -c fichier1.c fichier2.c fichier3.c
ar cr libessai.a fichier1.o fichier2.o fichier3.o
ranlib libessai.a
```

Utilisation d'une bibliothèque statique

```
gcc -c main.c
gcc -static -o executable main.o -L. -lessai
(équivalent gcc -static -o executable main.o libessai.a)
```

Bibliothèque dynamique

Création d'une bibliothèque dynamique

```
gcc -c fichier1.c fichier2.c fichier3.c
gcc -shared -o libessai.so fichier1.o fichier2.o fichier3.o
```

Utilisation d'une bibliothèque dynamique

```
gcc -c main.c
gcc -o executable main.o -L. -lessai
```

Remarques sur les recherches de chemins

Lors des phases de compilation, les fichiers nécessaires sont recherchés dans des répertoires par défaut, le plus souvent :

- `/usr/include` pour les fichiers d'en-tête (fichiers `.h`) inclus par par `#include <...>`
- `.` pour les fichiers d'en-tête inclus par `#include "..."`
- `/lib` et `/usr/lib` pour les bibliothèques statiques (fichiers `libXXX.a` et dynamiques(fichiers `libXXX.so`

On peut indiquer d'autres répertoires ou chercher les fichiers d'en-tête et bibliothèque au compilateur, en utilisant les options :

- **-I** pour indiquer un répertoire où chercher des fichiers d'en-tête
- **-L** pour indiquer un répertoire où chercher des fichiers bibliothèques

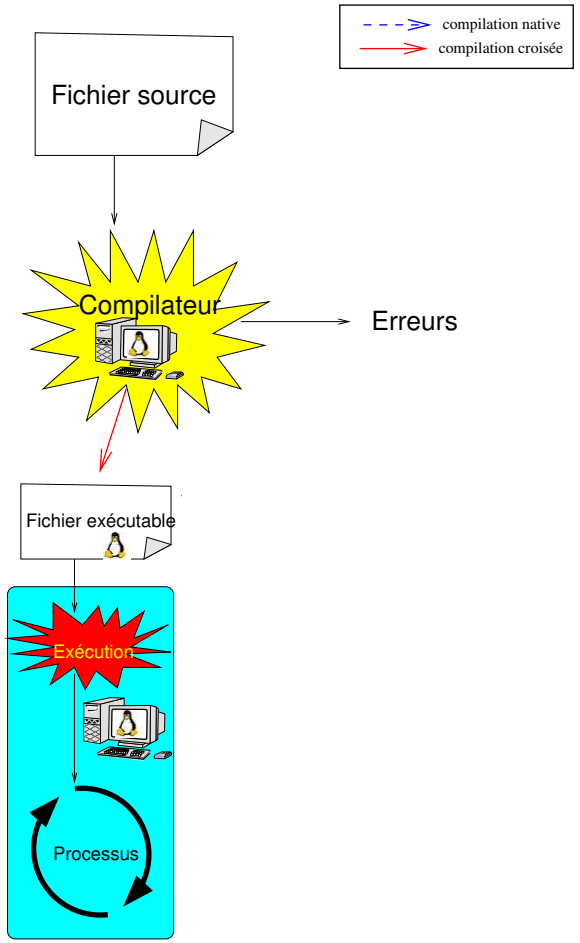
```
gcc essai.c -I/home/dntt/entete/
gcc essai.o -L/home/dntt/lib/ -L.
```

Remarques sur les recherches de chemins

Lors de la phase d'exécution, les bibliothèques dynamiques nécessaires à l'exécution sont recherchées dans les répertoires précisés sur votre système (`/lib`, `/usr/lib`, `/usr/X11R6/lib`, etc.).
On peut indiquer d'autres répertoires ou chercher les bibliothèque lors de l'exécution, en positionnant la variable d'environnement : `LD_LIBRARY_PATH`

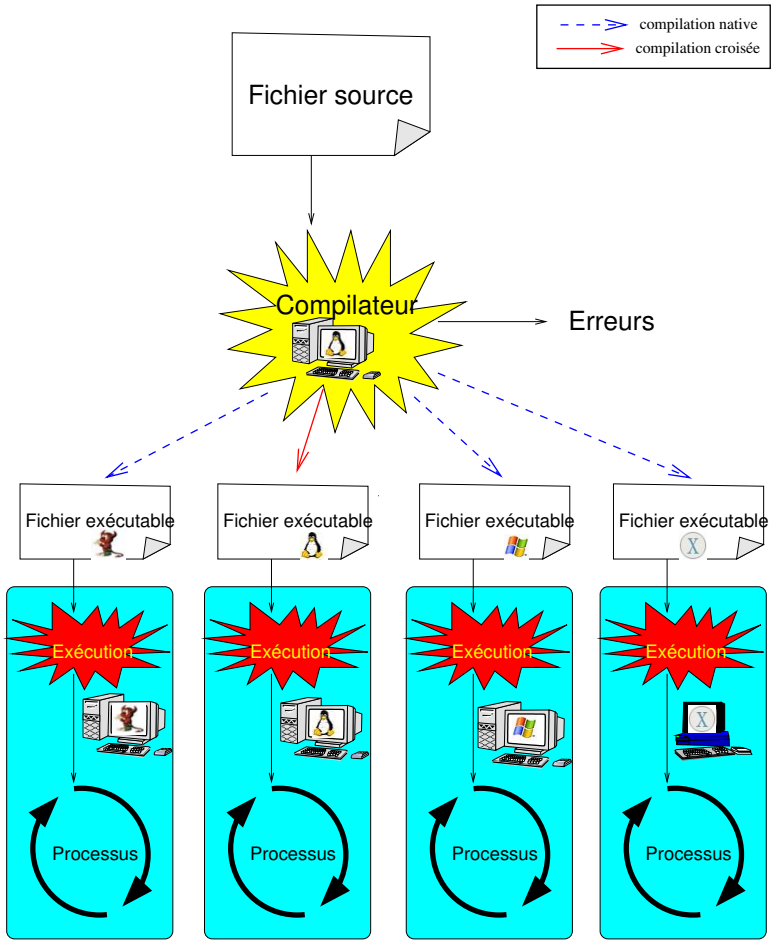
Compilation croisée

Pouvoir générer des fichiers objets et des exécutables sur d'autres architectures.



Compilation croisée

Pouvoir générer des fichiers objets et des exécutables sur d'autres architectures.

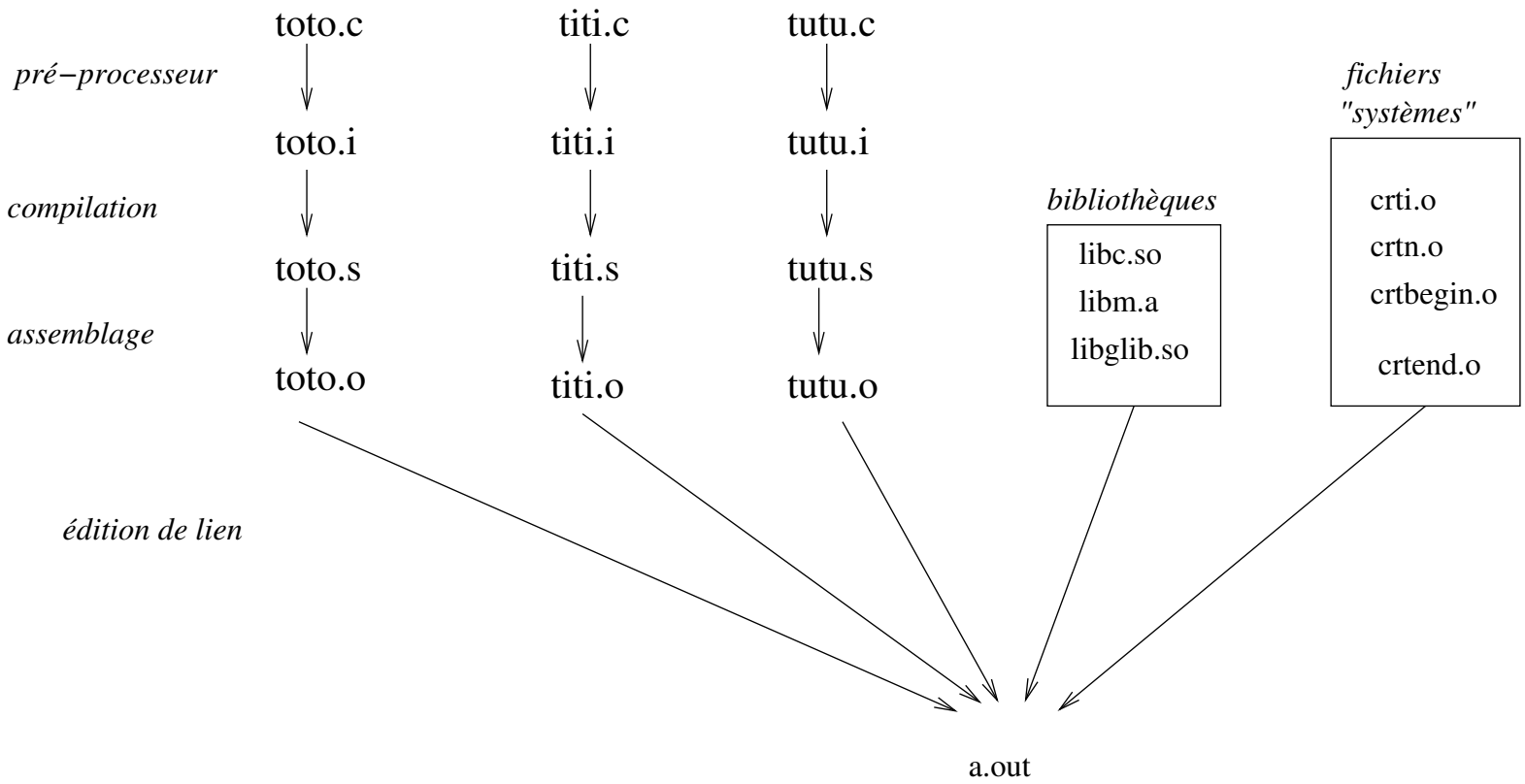


Autres options de gcc

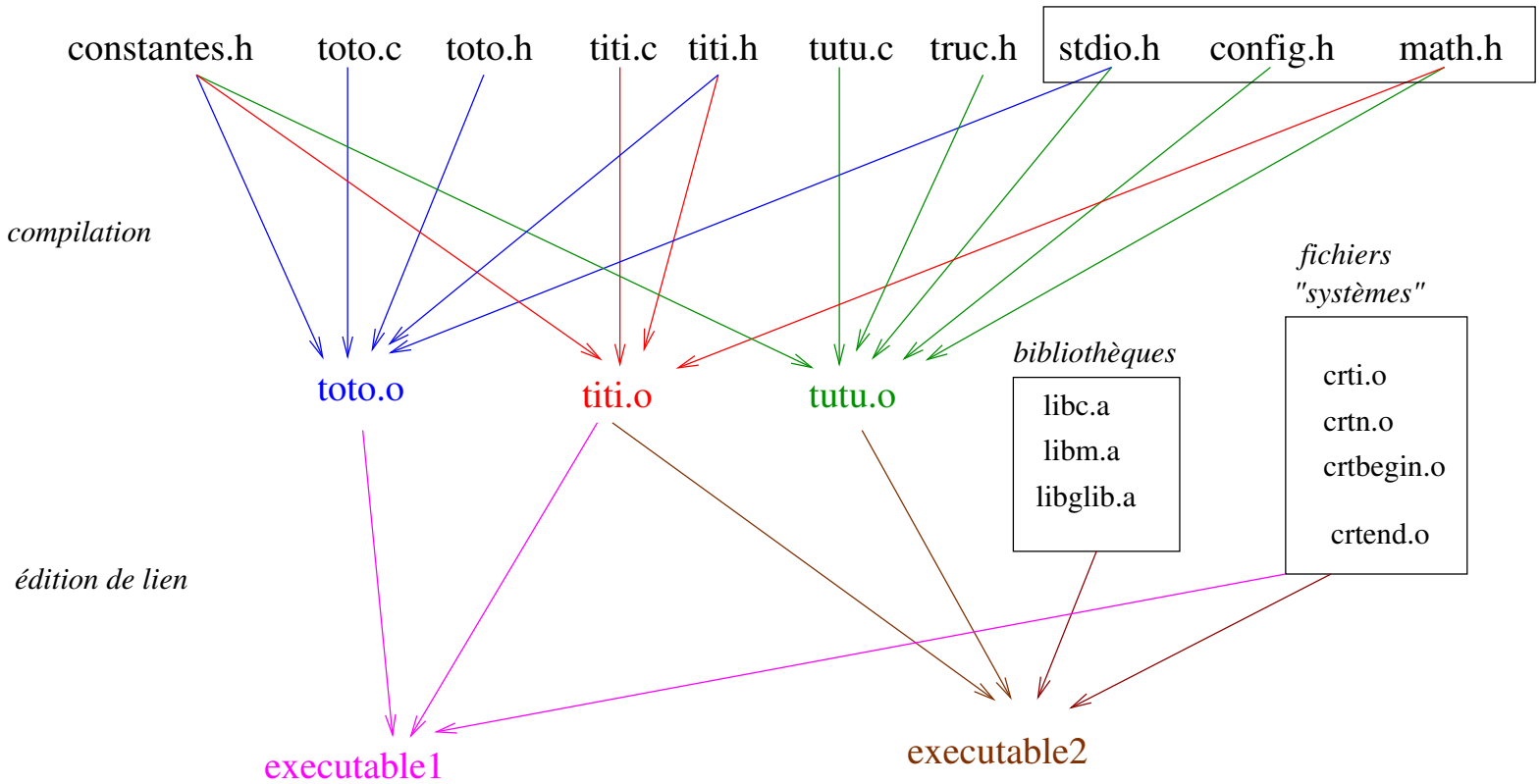
gcc supporte beaucoup d'autres options (indiqué par **man**). Dont :

- **-g** : inclu les options de débogage pour permettre d'utiliser ensuite un débogueur.
- **-O***x* (où x vaut 0, 1, 2, 3 oou s) : indique le niveau et le type d'optimisation
- **-Wall -pedantic** : affiche tous les "warning"
- **-D***symbole=valeur* : permet de déinir des constantes pour le préprocesseur. Equivalent à un `#define symbole valeur` dans le code.

Compilation



Dépendance de compilation



dépendance	compilation	cible
constantes.h, toto.c, toto.h	gcc -c toto.c	toto.o
constantes.h, titi.c, titi.h	gcc -c titi.c	titi.o
constantes.h, tutu.c, truc.h	gcc -c tutu.c	tutu.o
toto.o, titi.o	gcc -toto.o titi.o -o executable1	executable1
titi.o, tutu.o	gcc -titi.o tutu.o -o	executable2

Makefile

La commande **make** recherche dans le répertoire courant un fichier nommé `Makefile`. S'il existe, les directives qui y sont décrites sont suivies.
Format d'un fichier `Makefile` :

```
# 1) Partie déclaration des variables
nom\_variable = valeur

# 2) Partie directives :
cible : fichiers dépendants
        action (commande shell)
```

Partie déclaration des variables

Dans cette partie on y déclare toutes les variables qui seront utilisées avec `$nom_variable$` ou `${nom_variable}` dans la suite. En général, on définit de manière globale le compilateur utilisé, les options, etc.
Exemple

```
COMPILATEUR = gcc
COMP = -c
DEST = -o
CHEMIN_BIBLIOTHEQUE = -L/home/dntt/bib -L.
CHEMIN_ENTETE = -I/home/dntt/entete
BIBLIO = -lm -lglib -lc
```

Partie directives

Cette partie se compose d'un ensemble de directives écrites de la façon suivante :

```
cible : fichiers dépendants
        action (commande shell)
```

(attention, le blanc avant action, doit impérativement être une tabulation)

Exemple

```
toto.o : constantes.h toto.c toto.h
    gcc -c toto.c

titi.o : constantes.h titi.c titi.h
    gcc -c titi.c

tutu.o : constantes.h tutu.c truc.h
    gcc -c tutu.c & tutu.o

executable1 : toto.o titi.o
    gcc -toto.o titi.o -o executable1

executable2 : titi.o tutu.o
```

make

- On utilise le `Makefile` en tapant **make** *cible*.
- si la cible n'existe pas ou si elle est moins récente qu'au moins un des fichiers dépendant, alors l'action est exécutée.
- si le fichier dépendant est lui-même une cible d'une autre directive alors la directive correspondante est invoquée préalablement (et ainsi de suite).
- Il est d'usage d'utiliser des cibles comme **all**, **clean**, **print**, etc. qui sont des cibles fictives permettant de réaliser l'action indiquée, ou de générer les dépendances.

```
all : executable1 executable2

clean :
    rm -f  executable1 executable2 *.o

print :
   lpr *.c *.h
```

La commande **make** tapé sans cible utilise la première cible déclarée dans le Makefile.

Usuellement, on met la cible **all** comme première directive.

Usage avancé de **make**

- $@ : produit (ou but) de la règle
- $< : nom de la première dépendance (ou source)
- $? : toutes les dépendances plus récentes que le but
- $^ : toutes les dépendances
- $+ : idem mais chaque dépendance apparait autant de fois qu'elle est cité et l'ordre d'apparition est conservé.

Exemple :

```
%.o: %.c
	$(COMPILATEUR) -o $@ -c $<
```

Utilitaires utiles

- **indent** : formatte le code source de façon standardisé
- **lint** : vérificateur de sources C (eq. -Wall -pedantic)
- **strip** : supprime la table des symboles contenus par défaut dans un fichier objet. Cette opération permet de gagner de la place.
- **nm** : affiche la table des symboles
- **size** : donne le nombre (en décimal) doctets occupés par le texte et les données dun fichier-objet.
- **prof** interprète le fichier produit par l'exécution d'un programme compilé avec -p. Donne pour chaque procédure externe le pourcentage de temps passé à exécuter cette procédure, le nombre d'appels et le temps en millisecondes par appel.
- **ldd** imprime les bibliothèques partagées nécessaires au chargement dynamique du programme exécutable.

Recommandations

Toutes les options de **gcc**, des utilitaires ainsi que les informations sur les commandes se trouvent dans les pages de manuel, en tapant **man** *nom_commande*
Toutes les primitives et fonctions de bibliothèques sont décrites dans les pages 2 et 3 du manuel. Ex : **man 2 read**

gdb

gdb est le debuggueur de C. Il convient d'avoir compilé avec l'option -**g**. **gdb** permet d'analyser l'exécution d'un programme C.

- a priori : en suivant pas à pas l'exécution d'un programme C de façon interactive
- a posteriori : en analysant un fichier `core` généré lors d'un dump d'un programme qui (souvent) a planté.

Il existe une interface graphique **ddd** permettant de contrôler **gdb** de façon plus conviviale.

gdb

gdb permet à tout instant de :

- contrôler l'exécution du code
- afficher l'état de la mémoire
- afficher la valeur des variables
- afficher l'état de la pile d'exécution
- etc.

toto.c

```
#include <stdio.h>
#include "calcul.h"

/* Programme principal */
int main () {
    float resultat ;
    float r = 12 ;
    resultat = calcul_surface (r) ;
    printf ("Surface = %f\n", resultat) ;
    return 0 ;
}
```

calcul.h

```
#ifndef __CALCUL_H_
#define __CALCUL_H_

#define PI 3.1415

float calcul_aire (float rayon) ;
float calcul_surface (float rayon) ;

#endif /* __CALCUL_H_ */
```

calcul.c

```
#include <math.h>
#include "calcul.h"

float calcul_aire (float rayon) {
    return PI * pow (rayon, 2.0) ;
}

float calcul_surface (float rayon) {
    return 2 * PI * rayon ;
}
```

Makefile

```
all : executable1 executable2

toto.o : constantes.h toto.c toto.h
    gcc -c toto.c

titi.o : constantes.h titi.c titi.h
    gcc -c titi.c

tutu.o : constantes.h tutu.c truc.h
    gcc -c tutu.c & tutu.o

executable1 : toto.o titi.o
    gcc -toto.o titi.o -o executable1

executable2 : titi.o tutu.o
    gcc -titi.o tutu.o -o executable2 -lc -lm -lglib

clean :
    rm -f *.o executable1 executable2
```