Langage C avancé
Écriture de programmes sur plusieurs fichiers

Samuel KOKH

CEA Saclay
samuel.kokh@cea.fr

MACS – Institut Galilée

La compilation « niveau zéro »

main.c

```
#include<stdio.h>
...
#define Adef ...
#define Bdef ...
...
double r_g=2.1;
const int a_g=2;
...
char F(int x){
...
}

char G(int x){
...
}
char H(int x){
...
}

int main(void){
...
}
```

preprocesseur
compilateur
éditeur de liens

prog.exe

exécutable

Compilation « niveau zéro »

L'ensemble du code est regroupé dans un seul fichier.

La compilation « niveau zéro »

main.c

```
#include<stdio.h>
...
#define Adef ...
#define Bdef ...
...
double r_g=2.1;
const int a_g=2;
...
char F(int x){
...
}

char G(int x){
...
}
char H(int x){
...
}

int main(void){
...
}
```

```
$ gcc -Wall main.c -o prog.exe
```

prog.exe

`exécutable`

Compilation « niveau zéro »

L'ensemble du code est regroupé dans un seul fichier.

Production de l'exécutable en une seule commande shell.

L'appel de `gcc` se charge de la séquence: `préprocesseur` `compilateur` `éd. des liens`

gcc peut se charger de déclencher plusieurs programmes (préprocesseur, compilateur, éditeur de liens)

La compilation « niveau zéro »

main.c

```
#include<stdio.h>
...
#define Adef ...
#define Bdef ...
...
double r_g=2.1;
const int a_g=2;
...
char F(int x){
...
}

char G(int x){
...
}
char H(int x){
...
}

int main(void){
...
}
```

```
$ gcc -Wall -c main.c
$ gcc main.o -o prog.exe
```

prog.exe

exécutable

Compilation « niveau zéro »

L'ensemble du code est regroupé dans un seul fichier.

Version équivalente en 2 passes :

1. un appel à `gcc` lance le preprocesseur puis le compilateur pour produire le fichier objet `main.o`
2. appel à `gcc` lance l'éditeur de lien pour produire l'exécutable

Que se passe-t'il lors des phases de compilation ?

Préprocesseur

Compilation

Édition de liens

Que se passe-t'il lors des phases de compilation ?

Préprocesseur | Compilation | Édition de liens

Préprocesseur = cpp = **C** **P**re**P**rocessor

Le préprocesseur lit les fichiers sources (`*c` et `*.h`).

Le préprocesseur :

- cherche des directives de précompilation (macros cpp) dans les fichiers sources
- modifie le fichier source en fonction de ces directives (remplacement syntaxique).

Que se passe-t'il lors des phases de compilation ?

Préprocesseur | Compilation | Édition de liens

file.c

```
#include <math.h>
#include "myheader.h"
...
#define PI 3.14159
#define aMACRO(a,b) (a)*((b)++)
...
double res=cos(0.5*PI);
int i=0;
...
res=f(3*PI,1);
i=aMACRO(4,i);
...
```

préprocesseur →

file.i

```
→ contenu du fichier math.h
→ contenu du fichier myheader.h
...

...
double res=cos(0.5*3.14159);
int i=0;
...
res=f(3*3.14159,1);
i=(4)*((i)++);
...
```

- invocation directe : `$ cpp file.c file.i`
- invocation directe via gcc : `$ gcc -E file.c -o file.i`

Que se passe-t'il lors des phases de compilation ?

Préprocesseur Compilation Édition de liens

file.c

```
#include <math.h>
#include "myheader.h"
...
#define PI 3.14159
#define aMACRO(a,b) (a)*((b)++)
...
double res=cos(0.5*PI);
int i=0;
...
res=f(3*PI,1);
i=aMACRO(4,i);
...
```

préprocesseur →

file.i

```
→ contenu du fichier math.h
→ contenu du fichier myheader.h
...


...
double res=cos(0.5*3.14159);
int i=0;
...
res=f(3*3.14159,1);
i=(4)*((i)++);
...
```

Lors de la production d'un fichier objet via `gcc -c`, l'appel au precompilateur est « encapsulé » : il n'y pas de production de fichier post-processé, le résultat du post-process est directement passé au compilateur.

Que se passe-t'il lors des phases de compilation ?

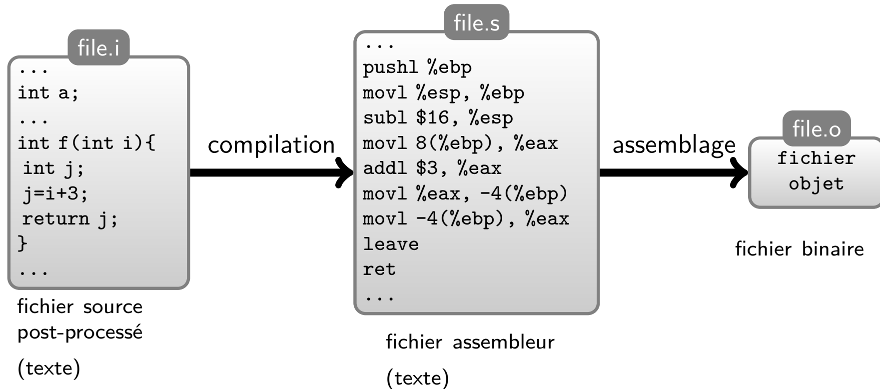
Préprocesseur | Compilation | Édition de liens

Ce qu'on appelle souvent (par concision) compilation des binaires s'effectue en deux étapes : la compilation proprement dite, puis l'assemblage.

Que se passe-t'il lors des phases de compilation ?

Préprocesseur | **Compilation** | Édition de liens

Ce qu'on appelle souvent (par concision) compilation des binaires s'effectue en deux étapes : la compilation proprement dite, puis l'assemblage.



Que se passe-t'il lors des phases de compilation ?

Préprocesseur | **Compilation** | Édition de liens

Appels explicites

- compilation : source post-processé (`*.i`)→ assembleur (`*.s`)
  ```
  $ gcc -S file.i
  ```
- assemblage : assembleur (`*.s`)→ fichier objet (`*.o`)
  ```
  $ gcc -c file.s
  ```

Que se passe-t'il lors des phases de compilation ?

Préprocesseur | Compilation | Édition de liens

En pratique

On sépare rarement toutes les étapes de production qui vont du fichier source `*.c` jusqu'au fichier objet `*.o`. On laisse en général le soin à `gcc` de gérer toutes les étapes intermédiaires.

commande	préprocesseur	compilation	assemblage	fichiers de sortie
`$ gcc -E`	×			`*.i`
`$ gcc -S`	×	×		`*.s`
`$ gcc -c`	×	×	×	`*.o`

Usage courant : `*.c` → `*.o`

```
$ gcc -Wall -c file.c
```

-Wall = option de compilation courante (facultatif).

Que se passe-t’il lors des phases de compilation ?

Préprocesseur | Compilation | Édition de liens

Principe général

L’édition de lien permet de combiner des portions de code binaire provenant de fichiers différents afin de réunir des données ou des portions de code provenant de fichiers binaires différents.

Deux manière d’effectuer cette combinaison :

- lien statique
- lien dynamique

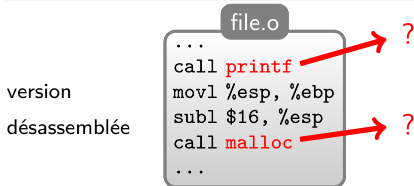
(ces points seront détaillés ultérieurement)

Que se passe-t'il lors des phases de compilation ?

Préprocesseur | Compilation | **Édition de liens**

Cas particulier du fichier source unique

Le code source peut faire appel à des fonctions standard de la librairie C : `malloc`, `printf`,...



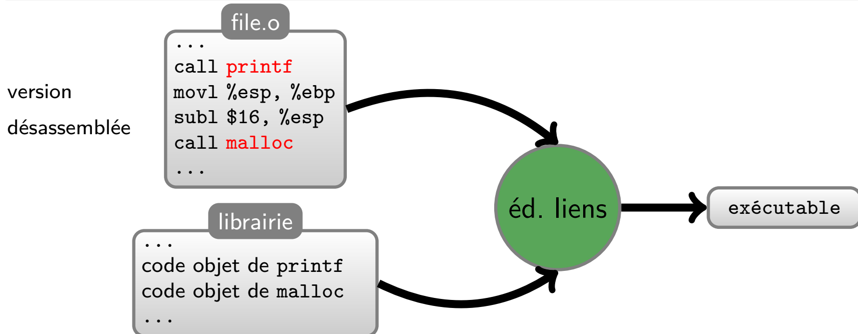
Le code source entré par le programmeur ne contient pas de définition pour ces fonctions (`malloc`, `printf`,...).

Que se passe-t'il lors des phases de compilation ?

Préprocesseur | Compilation | **Édition de liens**

Fonction de l'éditeur de liens

L'éditeur de liens permet de résoudre ces références (et aussi d'autres, comme nous le verrons) afin de produire l'exécutable.



Vers la mutualisation des ressources

Intérêt des librairies et de l'éditeur de liens

Chaque code peut faire référence à une définition commune de fonctions (`malloc`, etc.), sans avoir à inclure ses propres.

La mise en commun est réalisée grâce à l'éditeur de liens.

Réduction et rationalisation des ressources.

Comment poursuivre dans ce sens ?

compilation de sources réparties sur plusieurs fichiers.

Vers la mutualisation des ressources

Intérêt des librairies et de l'éditeur de liens

Chaque code peut faire référence à une définition commune de fonctions (`malloc`, etc.), sans avoir à inclure ses propres.

La mise en commun est réalisée grâce à l'éditeur de liens.

Réduction et rationalisation des ressources.

Comment poursuivre dans ce sens ?

compilation de sources réparties sur plusieurs fichiers.

Pourquoi répartir des données sur plusieurs fichiers ?

Simplifier la gestion des sources

Nombre de lignes de code (en ne comptant que les fichier *.cpp) pour qt-x11-opensource-src-4.5.2 :

$$2.396051 \times 10^6$$

Modulariser le code

Isoler des blocs de code sources (développement collectifs).

Diminuer les temps de recompilation

Temps de compilation de plusieurs heures

Du code source à un fichier au code source à plusieurs fichiers

Principe

- délimiter des blocs de code cohérents
- identifiers les interactions entre blocs de code
- identifiers les dépendances entre blocs de code

On verra plus loin quelques règles d'organisation pratiques,...

main.c

```
#include<stdio.h>
...
#define Adef ...
#define Bdef ...
...
double r_g=2.1;
const int a_g=2;
...
char F(int x){
...
}

char G(int x){
...
}
char H(int x){
...
}

int main(void){
...
}
```

main.c

```
#include<stdio.h>
...
#define Adef ...
#define Bdef ...
...
double r_g=2.1;
const int a_g=2;
...
char F(int x){
...
}

char G(int x){
...
}
char H(int x){
...
}

int main(void){
...
}
```

main.c

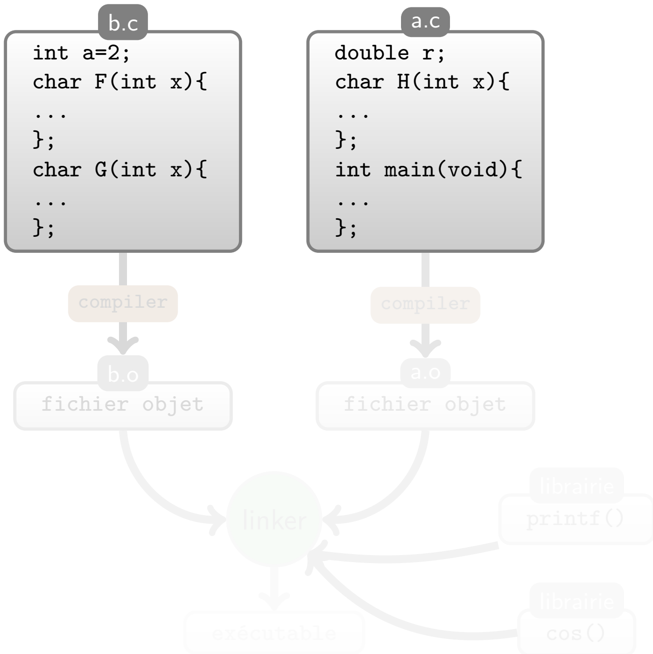
```c
#include<stdio.h>
...
#define Adef ...
#define Bdef ...
...
double r_g=2.1;
const int a_g=2;
...
char F(int x){
...
}

char G(int x){
...
}
char H(int x){
...
}

int main(void){
...
}
```

a.c

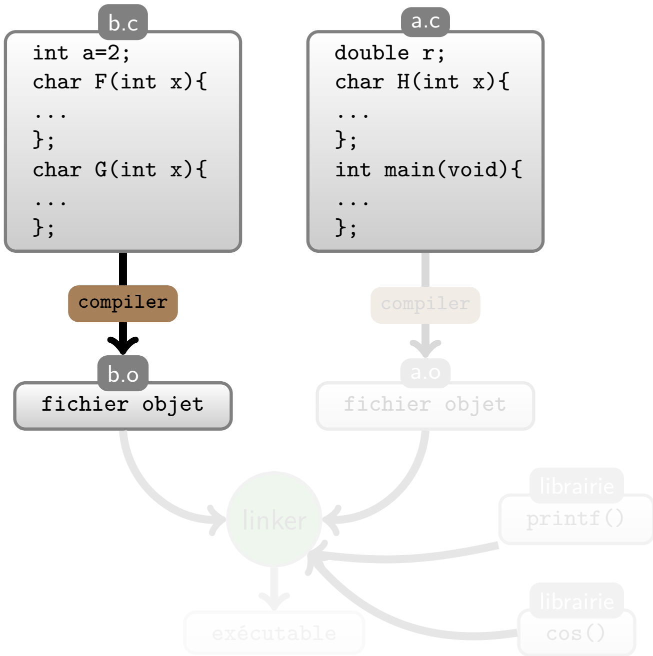
```c
#define Bdef ...
...
double r_g = 2.1;
...
char H(int x){
...
};
int main(void){
...
};
```

b.c

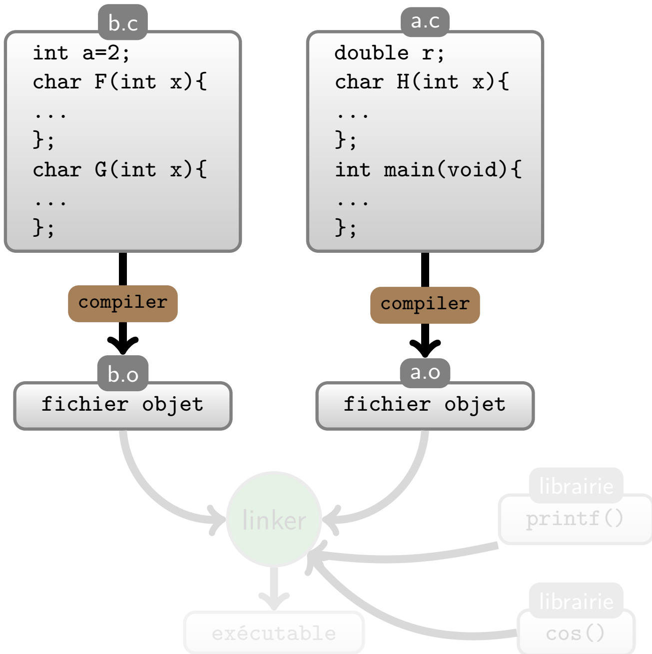
```c
...
#define Adef ...
...
const int a_g = 2;
...
char F(int x){ ...
};
char G(int x){ ...
};
```

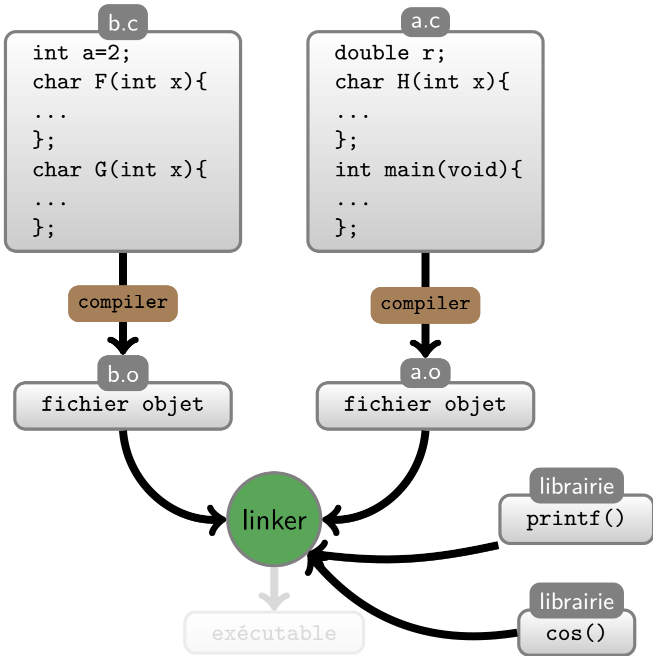


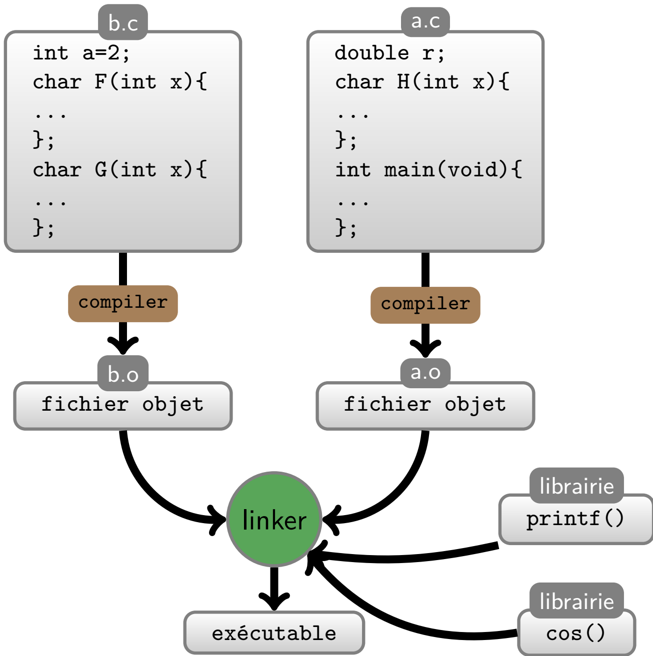
b.c
int a=2;
char F(int x){
...
};
char G(int x){
...
};
a.c
double r;
char H(int x){
...
};
int main(void){
...
};
compiler
compiler
b.o
fichier objet
a.o
fichier objet
linker
librairie
printf()
librairie
cos()
exécutable



b.c
int a=2;
char F(int x){
...
};
char G(int x){
...
};
a.c
double r;
char H(int x){
...
};
int main(void){
...
};
compiler
compiler
b.o
fichier objet
a.o
fichier objet
linker
librairie
printf()
librairie
cos()
exécutable


b.c
int a=2;
char F(int x){
...
};
char G(int x){
...
};
a.c
double r;
char H(int x){
...
};
int main(void){
...
};
compiler
compiler
b.o
fichier objet
a.o
fichier objet
linker
librairie
printf()
librairie
cos()
exécutable

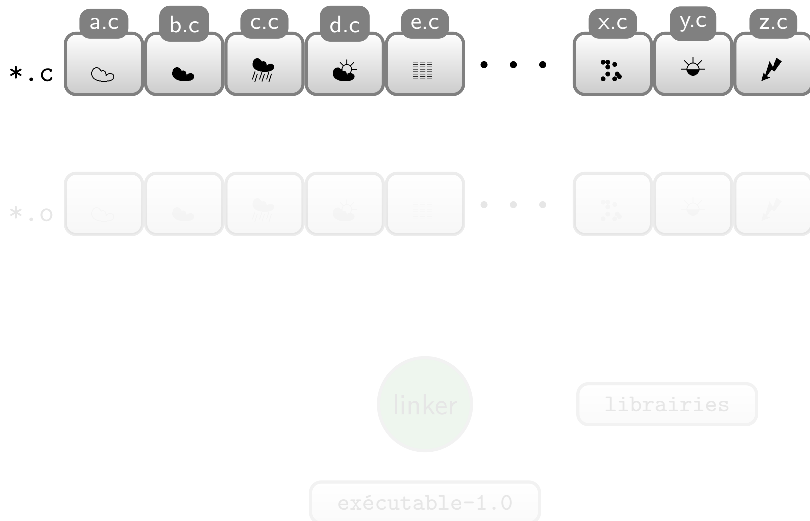

b.c
int a=2;
char F(int x){
...
};
char G(int x){
...
};
a.c
double r;
char H(int x){
...
};
int main(void){
...
};
compiler
compiler
b.o
fichier objet
a.o
fichier objet
linker
librairie
printf()
librairie
cos()
exécutable


b.c
int a=2;
char F(int x){
...
};
char G(int x){
...
};
a.c
double r;
char H(int x){
...
};
int main(void){
...
};
compiler
compiler
b.o
fichier objet
a.o
fichier objet
linker
librairie
printf()
librairie
cos()
exécutable

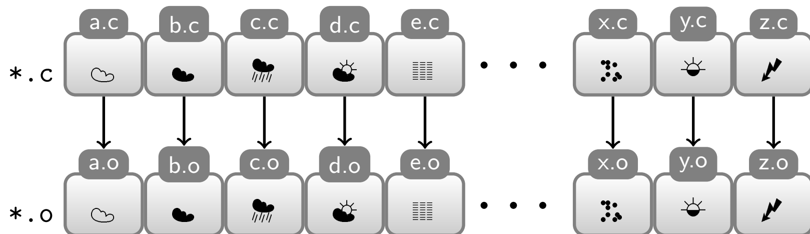
Recompilation

La compilation séparée peut permettre des gains de temps très substantiels pour les phases de recompilation.

Recompilation



Recompilation



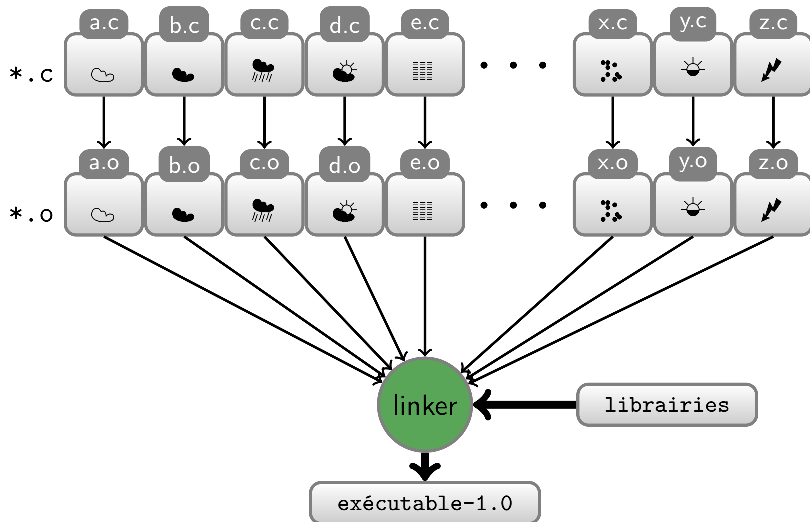
linker

librairies

exécutable-1.0

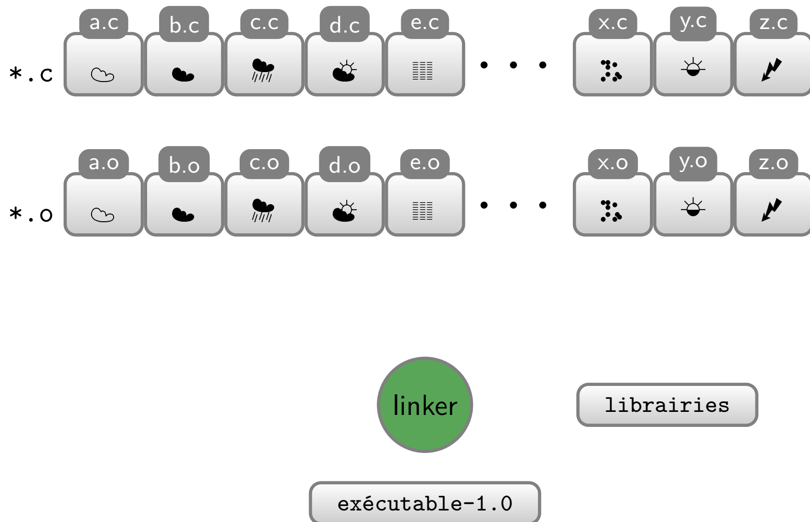
compilation séparée des *.o

Recompilation

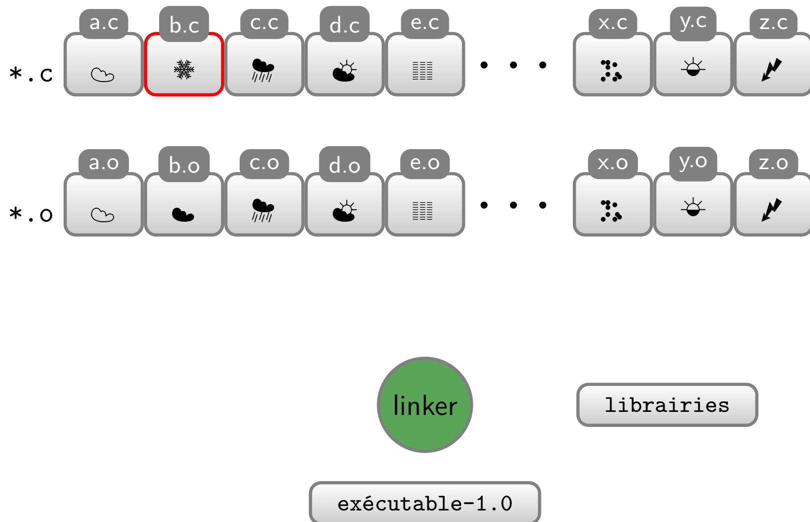


Édition des liens → exécutable-1.0

Recompilation

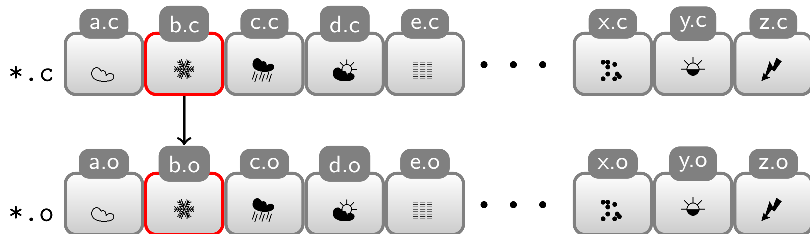


Recompilation



modification de b.c

Recompilation



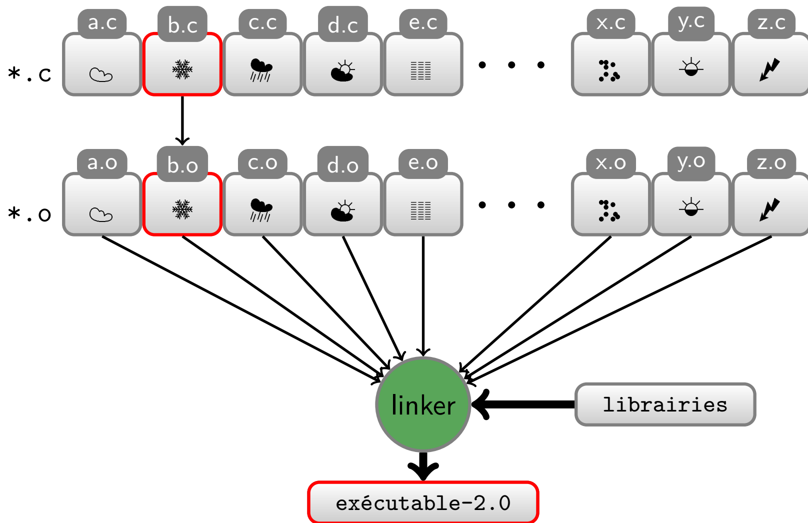
linker

librairies

exécutable-1.0

compilation du seul fichier objet b.o

Recompilation



nouvelle phase d'édition des liens → exécutable-2.0

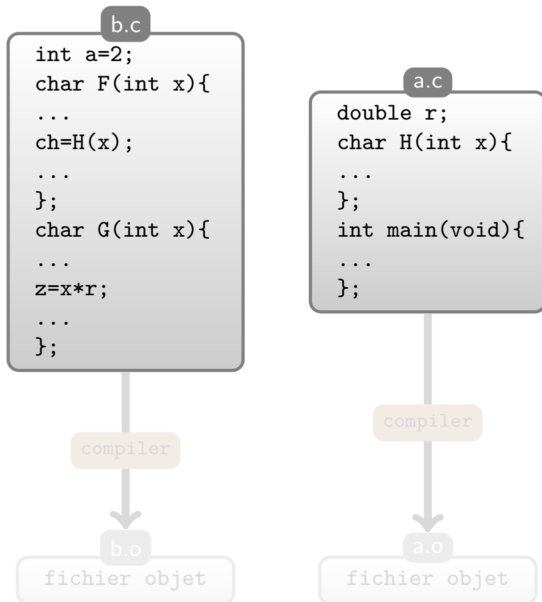
Recompilation

La compilation séparée permet lors d'une recompilation après modification de fichiers sources de ne recompiler que les seuls fichiers qui ont été modifiés, au lieu de l'ensemble des fichiers.

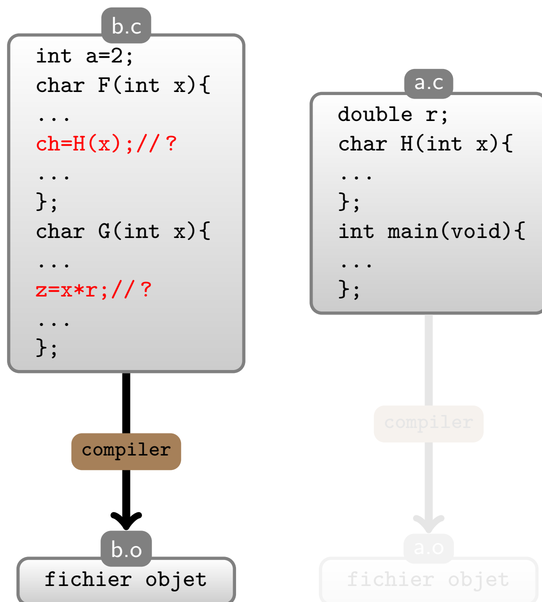
Une fois les nouveaux fichiers objets créés, on procède de manière classique pour l'édition des liens afin de créer un nouvel exécutable.

A titre d'exemple : qt-4.3.2 comporte 1845 fichiers `*.h` et `*.cpp`.

Problème des dépendances entre fichiers



Problème des dépendances entre fichiers



Problèmes des dépendances entre fichiers

Que faire ?

Il faut pouvoir partager des informations entre les différents fichiers, et ce, avant l'édition de liens !

Que doit-on partager ?

- informations concernant les fonctions
- informations concernant les variables communes : les variables globales
- les directives préprocesseurs afin qu'elles soient bien définies partout où elles sont utilisées
- les définitions des types propres au projet établies via : `struct`, `union` et `typedef`

Problèmes des dépendances entre fichiers

Que faire ?

Il faut pouvoir partager des informations entre les différents fichiers, et ce, avant l'édition de liens !

Que doit-on partager ?

- informations concernant les fonctions
- informations concernant les variables communes : les variables globales
- les directives préprocesseurs afin qu'elles soient bien définies partout où elles sont utilisées
- les définitions des types propres au projet établies via : `struct`, `union` et `typedef`

Déclaration de fonctions

Déclarer une fonction sans la définir

On peut utiliser une fonction dans un fichier sans l'avoir définie (id est sans avoir donné le code source qui décrit ce que fait la fonction).

Il suffit pour celà d'avoir simplement déclarée la fonction.

Déclaration d'une fonction

On averti le compilateur de notre intention de définir une fonction ailleurs (plus loin dans le même fichier ou dans un autre fichier), et on lui fournit les informations nécessaires aux tâches qu'il doit effectuer.

Déclaration de fonctions

déclaration de fonctions

```
 ...
 int afct(double x, double y);  // cette fonction
                                // est définie ailleurs
 void* bfct(char str[], int z); // cette fonction
                                // est définie ailleurs
 ...
 ...
 ptr=bfct(mystr,zz); // on peut appeler la fonction afct
                     // dans le code source
 res=bfct(x,y);      // on peut appeler la fonction bfct
                     // dans le code source
...
```

La définition des fonctions déclarées peut avoir lieu au sein du même fichier ou dans un autre fichier.

Déclaration de fonctions

déclaration de fonctions

```
 ...
 extern int afct(double x, double y);   // cette fonction
                                        // est définie ailleurs
 extern void* bfct(char str[], int z); // cette fonction
                                        // est définie ailleurs
 ...
 ...
 ptr=bfct(mystr,zz); // on peut appeler la fonction afct
                     // dans le code source
 res=bfct(x,y);      // on peut appeler la fonction bfct
                     // dans le code source
...
```

Syntaxe équivalente avec le mot-clef `extern` (facultatif).

Déclaration de fonctions : en pratique

Déclaration comme une sorte de « publication »

On rend connu des autres fichiers, l'existence de la fonction `fct` définie dans `a.c`.

a.c

```
...
int fct( int x){
int res;
...
return res;
}
...
```

b.c

```
...
...
...
...
...
y=fct(3);
...
```

c.c

```
...
...
...
...
a=fct(b-2);
...
...
```

d.c

```
...
...
...
U=fct(V)*fct(T);
...
...
...
```

Déclaration de fonctions : en pratique

Déclaration comme une sorte de « publication »

On rend connu des autres fichiers, l'existence de la fonction `fct` définie dans `a.c`.

a.c
```
...
int fct( int x){
int res;
...
return res;
}
...
```

b.c
```
...
int fct( int x);
...
...
...
y=fct(3);
...
```

c.c
```
...
int fct( int x);
...
...
a=fct(b-2);
...
...
```

d.c
```
...
int fct( int x);
...
U=fct(V)*fct(T);
...
...
...
```

Déclaration de fonctions : en pratique

Mutualisation des déclarations

Utilisation des fichiers en-têtes (ou fichier include/header) `*.h`

a.h
```c
...
int fct( int x);
...
```

a.c
```c
...
int fct( int x){
int res;
...
return res;
}
...
```

b.c
```c
...
#include "a.h"
...
...
...
y=fct(3);
...
```

c.c
```c
...
#include "a.h"
...
...
a=fct(b-2);
...
...
```

d.c
```c
...
#include "a.h"
...
U=fct(V)*fct(T);
...
...
...
```

Déclaration des variables globales

b.c
```c
int a=2;
char F(int x){
...
z=3.1*r;
...
};
char G(int x){
...
l=strlen(s);
...
};
```

compiler

b.o
fichier objet

a.c
```c
double r;
char s[3]=".t";
char H(int x){
...
j=a++;
...
};
int main(void){
...
};
```

compiler

a.o
fichier objet

Le même problème de « publication d'information » entre fichiers se pose pour les variables globales.

Problème en 2 volets

- déclaration
- initialisation

Déclaration des variables globales

b.c

```
int a=2;
char F(int x){
...
z=3.1*r;//?
...
};
char G(int x){
...
l=strlen(s);//?
...
};
```

compiler

b.o

fichier objet

a.c

```
double r;
char s[3]=".t";
char H(int x){
...
j=a++;//?
...
};
int main(void){
...
};
```

compiler

a.o

fichier objet

Le même problème de « publication d'information » entre fichiers se pose pour les variables globales.

Problème en 2 volets

- déclaration
- initialisation

Déclaration des variables globales

b.c

```c
int a=2;
char F(int x){
...
z=3.1*r;//?
...
};
char G(int x){
...
l=strlen(s);//?
...
};
```

compiler

b.o

fichier objet

a.c

```c
double r;
char s[3]=".t";
char H(int x){
...
j=a++;//?
...
};
int main(void){
...
};
```

compiler

a.o

fichier objet

Le même problème de « publication d'information » entre fichiers se pose pour les variables globales.

Problème en 2 volets

- déclaration
- initialisation

Déclaration de variables globales

Le rémède est similaire à celui utilisé pour le fonctions.

Déclarer une variable sans la définir

- On peut utiliser une variable gloable dans un fichier sans l'avoir définie.

Déclaration d'une variable

On avertit le compilateur de noter intention de définir une variable ailleurs (plus loin dans le même fichier ou dans un autre fichier). On fournit au compilateur les informations nécessaire aux tâches qu'il doit exécuter

Déclaration de variables globales

- **Déclaration d'une variable globale** : avertit le compilateur de l'existence de la variable et de son type
- **Définition d'une variable globale** : demande au compilateur d'allouer de l'espace mémoire pour la variable et demande éventuellement son initialisation

Déclaration de variables globales

déclaration de variables globales

```
 ...
 extern int a; // cette variable est définie ailleurs on peut
               // à présent l'utiliser dans la suite du fichier
 extern struct Complex_t z; // cette variable est définie ailleurs
                            // on peut à présent l'utiliser
                            // dans la suite du fichier
...
int fct(int x){
   int res;
   extern int b; // b est une variable globale définie ailleurs;
                 // on peut à présent l'utiliser dans ce bloc
   ...
   return res;
}
```

La définition des variables déclarées peut avoir lieu au sein du même fichier ou dans un autre fichier.

Déclaration de variables globales

déclaration de variables globales

```
 ...
 extern int a; // cette variable est définie ailleurs on peut
               // à présent l'utiliser dans la suite du fichier
 extern struct Complex_t z; // cette variable est définie ailleurs
                            // on peut à présent l'utiliser
                            // dans la suite du fichier
...
int fct(int x){
   int res;
   extern int b; // b est une variable globale définie ailleurs;
                 // on peut à présent l'utiliser dans ce bloc
   ...
   return res;
}
```

Mot-clef `extern`

Pour la déclaration des variables l'utilisation de `extern` est considérée comme obligatoire contrairement aux fonctions).

Déclaration de variables globales

déclaration de variables globales

```
 ...
 extern int a; // cette variable est définie ailleurs on peut
               // à présent l'utiliser dans la suite du fichier
 extern struct Complex_t z; // cette variable est définie ailleurs
                            // on peut à présent l'utiliser
                            // dans la suite du fichier
...
int fct(int x){
   int res;
   extern int b; // b est une variable globale définie ailleurs;
                 // on peut à présent l'utiliser dans ce bloc
   ...
   return res;
}
```

Utilisation dans un fichier source d'une variable globale

Chaque utilisation dans un fichier source d'une variable globale doit être précédée d'une déclaration préalable de cette variable.

Définition des variables globales

La définition d'une variable globale est faite

- hors du corps de toute fonction dans un fichier source *.c
- un seul et unique fois sur l'ensemble de tous les fichiers source

La définition peut être associée à une initialisation de la variable globale induite par le programmeur. Autrement, une variable globale est toujours initialisée à zéro.

Définition des variables globales

```
int a=4; // hors de tout corps de fonction
         // définition et initialisation à 4
struct Complex_t z; // hors de tout corps de fonction
                    // définition et initialisation
                    // à 0
```

Déclaration de fonctions : en pratique

a.c

```
...
...
...
...
...
int i;
...
/*code utilisant
les var. globales*/
...
```

b.c

```
...
...
...
...
...
double d;
...
/*code utilisant
les var. globales*/
...
```

c.c

```
...
...
...
...
...
ch c='i';
...
/*code utilisant
les var. globales*/
...
```

d.c

```
...
...
...
...
...
int A[2];
...
/*code utilisant
les var. globales*/
...
```

Déclaration de fonctions : en pratique

a.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
int i;
...
/*code utilisant
les var. globales*/
...
```

b.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
double d;
...
/*code utilisant
les var. globales*/
...
```

c.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
ch c='i';
...
/*code utilisant
les var. globales*/
...
```

d.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
int A[2];
...
/*code utilisant
les var. globales*/
...
```

Déclaration de fonctions : en pratique

glob.h

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
...
```

a.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
int i;
...
/*code utilisant
les var. globales*/
...
```

b.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
double d;
...
/*code utilisant
les var. globales*/
...
```

c.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
ch c='i';
...
/*code utilisant
les var. globales*/
...
```

d.c

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
int A[2];
...
/*code utilisant
les var. globales*/
...
```

Déclaration de fonctions : en pratique

glob.h

```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
...
```

a.c

```
...
#include "glob.h"
...
...
...
int i;
...
/*code utilisant
les var. globales*/
...
```

b.c

```
...
#include "glob.h"
...
...
...
double d;
...
/*code utilisant
les var. globales*/
...
```

c.c

```
...
#include "glob.h"
...
...
...
ch c='i';
...
/*code utilisant
les var. globales*/
...
```

d.c

```
...
#include "glob.h"
...
...
...
int A[2];
...
/*code utilisant
les var. globales*/
...
```

Déclaration de fonctions : en pratique

glob.h
```c
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
...
```

glob.c
```c
...
char ch='i';
int A[2];
double d;
int i;
...
```

a.c
```c
...
#include "glob.h"
...
...
...
int i;
...
/*code utilisant
les var. globales*/
...
```

b.c
```c
...
#include "glob.h"
...
...
...
double d;
...
/*code utilisant
les var. globales*/
...
```

c.c
```c
...
#include "glob.h"
...
...
...
ch c='i';
...
/*code utilisant
les var. globales*/
...
```

d.c
```c
...
#include "glob.h"
...
...
...
int A[2];
...
/*code utilisant
les var. globales*/
...
```

Déclaration de fonctions : en pratique

glob.h
```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
...
```

glob.c
```
...
char ch='i';
int A[2];
double d;
int i;
...
```

a.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

b.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

c.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

d.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

Déclaration de fonctions : en pratique

glob.h
```
...
extern char ch;
extern int A[2];
extern double d;
extern int i;
...
```

glob.c
```
...
char ch='i';
int A[2];
double d;
int i;
...
```

- mutualisation des sources
- modularité

a.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

b.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

c.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

d.c
```
...
#include "glob.h"
...
...
...
...
...
/*code utilisant
les var. globales*/
...
```

Partage des directives de préprocess

b.c

```
...
res=cos(PI*x);
...
z=AVERAGE(3,y);
...
```

a.c

```
#define PI=3.14159
#define AVERAGE(a,b) ((a)+(b))/2
...
```

compiler

compiler

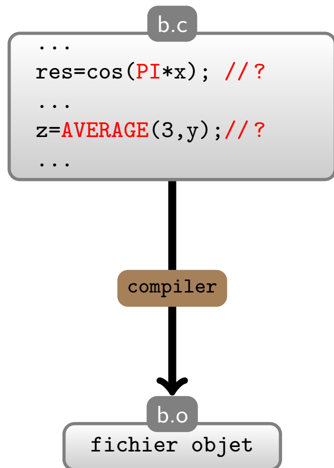
b.o

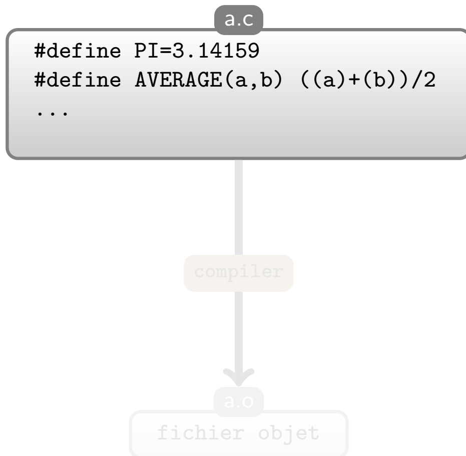
fichier objet

a.o

fichier objet

Partage des directives de préprocess





Partage des directives préprocesseurs

b.c

```
#include "defcpp.h"
...
res=cos(PI*x);
...
z=AVERAGE(3,y);
```

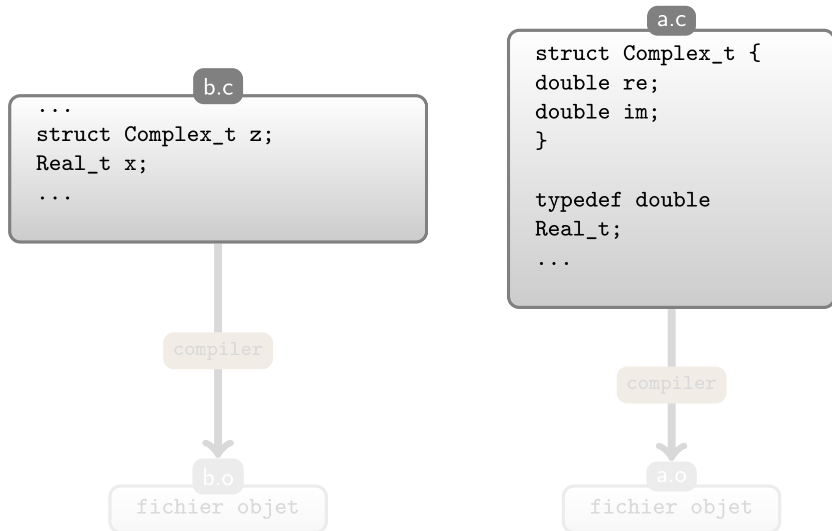
defcpp.h

```
...
#define PI=3.14159
#define AVERAGE(a,b) ((a)+(b))/2
...
```

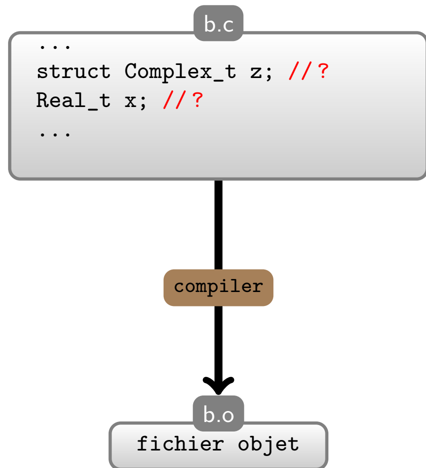
Mutualisation des définitions préprocesseurs

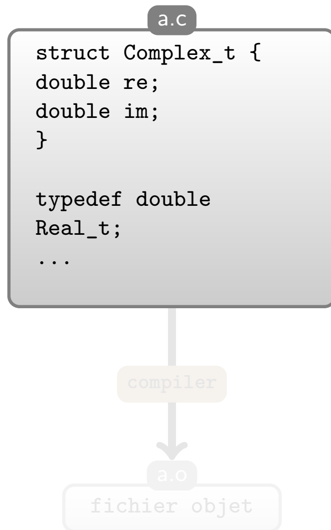
Là aussi, l'utilisation de fichiers en-têtes permet de gérer plus efficacement les directives préprocesseurs.

Partage des définitions de types



Partage des définitions de types





Partage des directives préprocesseurs

b.c

```
#include "deftypes.h"
...
struct Complex_t z;
Real_t x;
...
```

deftypes.h

```
...
struct Complex_t {
double re;
double im;
};
typedef double Real_t;
...
```

Mutualisation des définitions de types

Là aussi, l'utilisation de fichiers en-têtes permet de gérer plus efficacement les définitions de types.

Protection des ressources dans un fichier

Masquage/protection de ressources

Au contraire du partage de ressources, il peut être utile de restreindre la définition de certains éléments d'un fichier source au seul fichier.

Variables globales et fonctions

Utilisation du mot-clef `static` pour restreindre la définition d'une fonction ou d'une variable globale à un fichier.

static et fonctions

Toutes les fonctions ci-dessous ne peuvent être appelées que depuis le fichier `file.c` où elles sont définies.

file.c

```
static extern int afct(int i); // fonction définie ci-après
static int bfct(double x); // fonction définie ci-après
...
int afct(int i){
...
}

int bfct(double x){
...
}

static int cfct(char c){ // fonction définie seulement pour file.c
...
}
```

Il est ainsi possible d'avoir deux fonctions `static` différentes de même nom dans deux fichiers différents.

static et variables globales

La définition des variables globales ci-dessous est restreinte au fichier `file.c`

file.c

```
static extern int a; // variable globale définie ci-après
static extern float b; // variable globale définie ci-après
static char c='a';    // variable gloabale définie pour le
                      // seul fichier file.c
...
int a=1;              // définition et initialisation de a
float b;              // définition de b
...
}
```

Il est ainsi possible d'avoir deux variables gloables `static` différentes de même nom dans deux fichiers différents.

A propos de l'usage des variables globales

En règle général il est fortement déconseillé d'utiliser des variables globales.

Une variable globale est « vivante » pendant toute l'exécution du programme.
Conséquence : rechercher où une variable globale est utilisée/modifée nécessite *a priori* de parcourir **tout le code** (que faire si le code fait plusieurs centaines de milliers, voire millions de ligne de code ?). Ainsi la lecture du code devient extrêmement complexe.

Une variable globale est modifiable par n'importe quelle fonction à n'importe quel moment.
Conséquence : tout appel de fonction peut mettre en cause l'intégrité des valeurs d'une variable globale.

A propos de l'usage des variables globales

Si l'usage de variables globales est nécessaire : il sera utile d'essayer de les distinguer clairement des autres variables en utilisant une règle de nommage spécifique.

```
int globCounter;
int glob_Counter;
int _Counter;
int Counter_g;
```

A propos de l'usage des variables globales

Une méthode plus radicale peut consister à créer une variable de type struct destinée à accueillir toutes les variables globales. Ainsi :

- on force la syntaxe
- on regroupe toutes les variables globales au sein d'une seule

```
struct GlobalVar_t {
int compteur;
char id[3];
int iMax;
int jMax;
} ;

struct GlobalVar_t globVar;
...
printf("%d\n",globVar.iMax); // syntaxe contraignante
```