

Chapitre 7

Compilation et modularisation

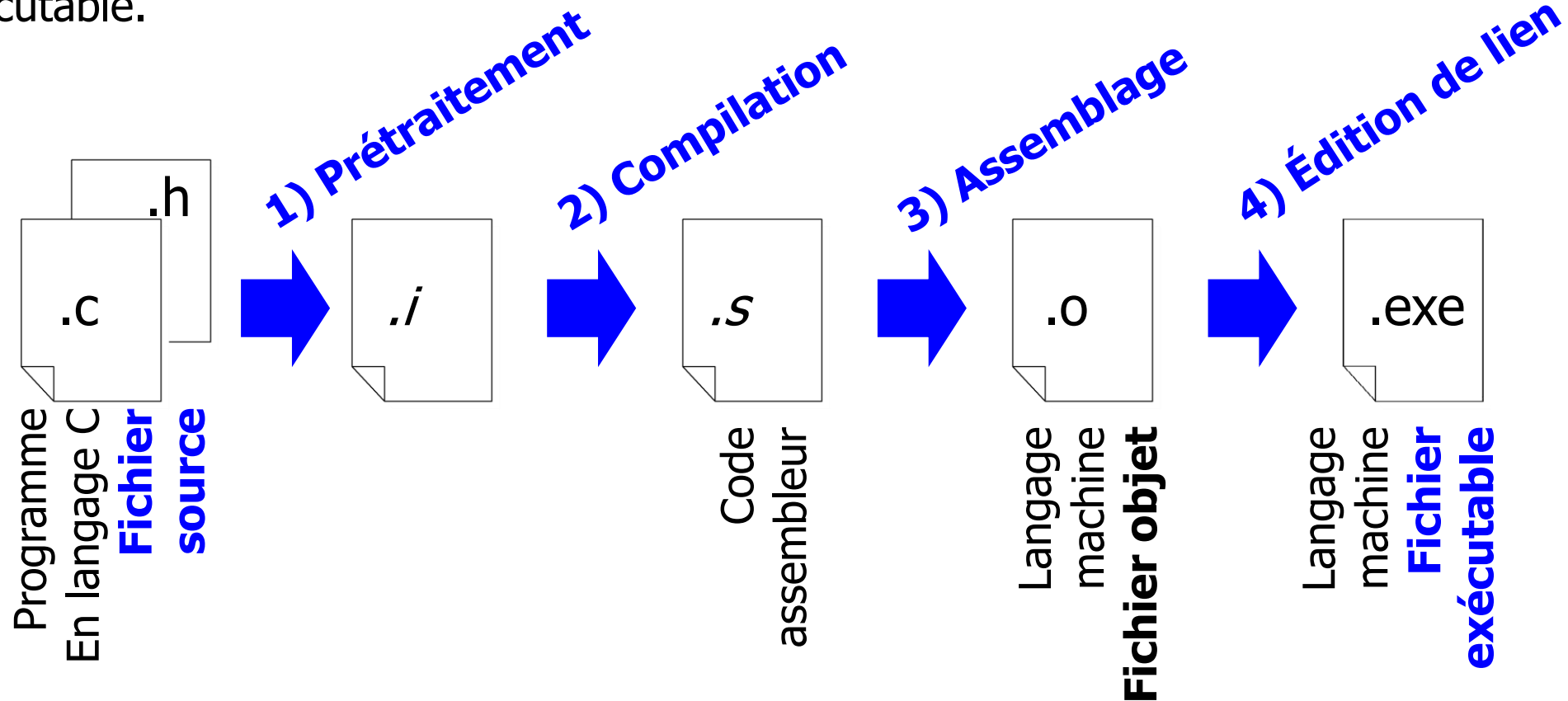
Plan

1. Compilateur

1. Prétraitement
 2. Compilation
 3. Assemblage
 4. Editeur de liens
2. L'outil `gcc`
 3. Modularisation
 4. L'outil `make`

7.1 Les phases de la compilation

Le compilateur C de `gcc` est un programme qui transforme un fichier codé en langage C en un fichier exécutable.



Plan

1. Compilateur

1. Prétraitement

2. Compilation

3. Assemblage

4. Editeur de liens

2. L'outil `gcc`

3. Modularisation

4. L'outil `make`

7.1.1 Le prétraitement / pré-compilation

Le fichier source subit des **transformations purement textuelles**

Les directives de prétraitement commencent par #

Par exemple

Définition de constante et de macro

`#define`, `#undef`, `#defined`

Inclusion de fichier

`#include`

Compilation conditionnelle

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

7.1.1 Le prétraitement / pré-compilation

Déclarations de constantes

```
#define PI 3.14
```

Déclarations de macros => **incorrecte**

```
#define MAX(x,y) x>y?x:y  
#define SQUARE(x) x*x
```



MAX (4 , 6) ?
2*MAX (4 , 6) ?
SQUARE (5) ?
SQUARE (3+2) ?

Déclarations de macros => **correcte**

```
#define MAX(x,y) ((x)>(y))?(x):(y)  
#define SQUARE(x) ((x)*(x))
```

7.1.1 Le prétraitement / pré-compilation

Compilation conditionnelle

```
#define _DEBUG  
#ifdef _DEBUG  
    printf(...);  
#endif
```

```
#ifndef _MON_FICHER_H  
#define _MON_FICHER_H  
  
/* Mettre toutes les déclarations ici */  
  
#endif
```

https://fr.wikibooks.org/wiki/Programmation_C/Pr%C3%A9processeur

7.1.1 Le prétraitement / pré-compilation

Détection du système d'exploitation

`__WIN32` ou `__WIN32__`
`linux` ou `__linux__`
`__APPLE__`

Visual C++

`__MSC_VER`

Version du compilateur gcc

`__GNUC__` ou `__GNUC_MINOR__`

Divers

`__MINGW32__` ou `__CYGWIN__` ou `__CYGWIN32__`

http://fr.wikibooks.org/wiki/Programmation_C/Pr%C3%A9processeur

7.1.1 Le prétraitement / pré-compilation

Symboles prédéfinis (ANSI/ISO)

`__FILE__` `__DATE__` `__TIME__` `__TIMESTAMP__`
`__LINE__` `__STDC__`

Exemple d'utilisation dans une définition multiligne (avec /)

```
#define ASSERT(cond)    if( !(cond) )    /\n{\n    printf( "assert error \nLine: %d    /\n                \nFile(%s)\n",    /\n                __LINE__, __FILE__ ) ; }
```

Définition de constantes lors de la compilation

```
gcc ... -D __DEBUG ...
```

```
gcc ... -U __DEBUG ...
```

Plan

1. Compilateur

1. Prétraitement

2. Compilation

3. Assemblage

4. Editeur de liens

2. L'outil `gcc`

3. Modularisation

4. L'outil `make`

7.1.2 La compilation

La **compilation** proprement dite **traduit** le fichier généré par le prétraitement en un fichier texte contenant du code **assembleur**

L'assembleur est une suite d'instructions pour un microprocesseur particulier qui utilise des mnémoniques rendant la lecture possible.

```
myLoop: dec CX
mov  tab[BX],CX ; fills in the table
inc  BX         ; index increment
cmp  CX,0
jne  myLoop
```

Plan

1. Compilateur

1. Prétraitement
2. Compilation

3. Assemblage

4. Editeur de liens
2. L'outil `gcc`
3. Modularisation
4. L'outil `make`

7.1.3 L'assemblage

Cette opération transforme le code assembleur en un **fichier binaire**, c'est-à-dire en instructions directement compréhensibles par le processeur.

Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur.

Le fichier produit par l'assemblage est appelé **fichier objet**.

```
00010110 10010010
00100010 10100101
10010010 00100100
00100101
10010001 10010010
```

Vue binaire

```
...É%] ô%Ã%uø%Ö%} ü
```

Vue ascii

Plan

1. Compilateur

1. Prétraitement
2. Compilation
3. Assemblage

4. Editeur de liens

2. L'outil `gcc`
3. Modularisation
4. L'outil `make`

7.1.4 L'édition de liens

Un programme est souvent séparé en plusieurs fichiers sources :
→ appel à des bibliothèques de fonctions

Une fois chaque code source assemblé, il faut **lier entre eux les différents fichiers objets**.

L'édition de liens produit alors soit :

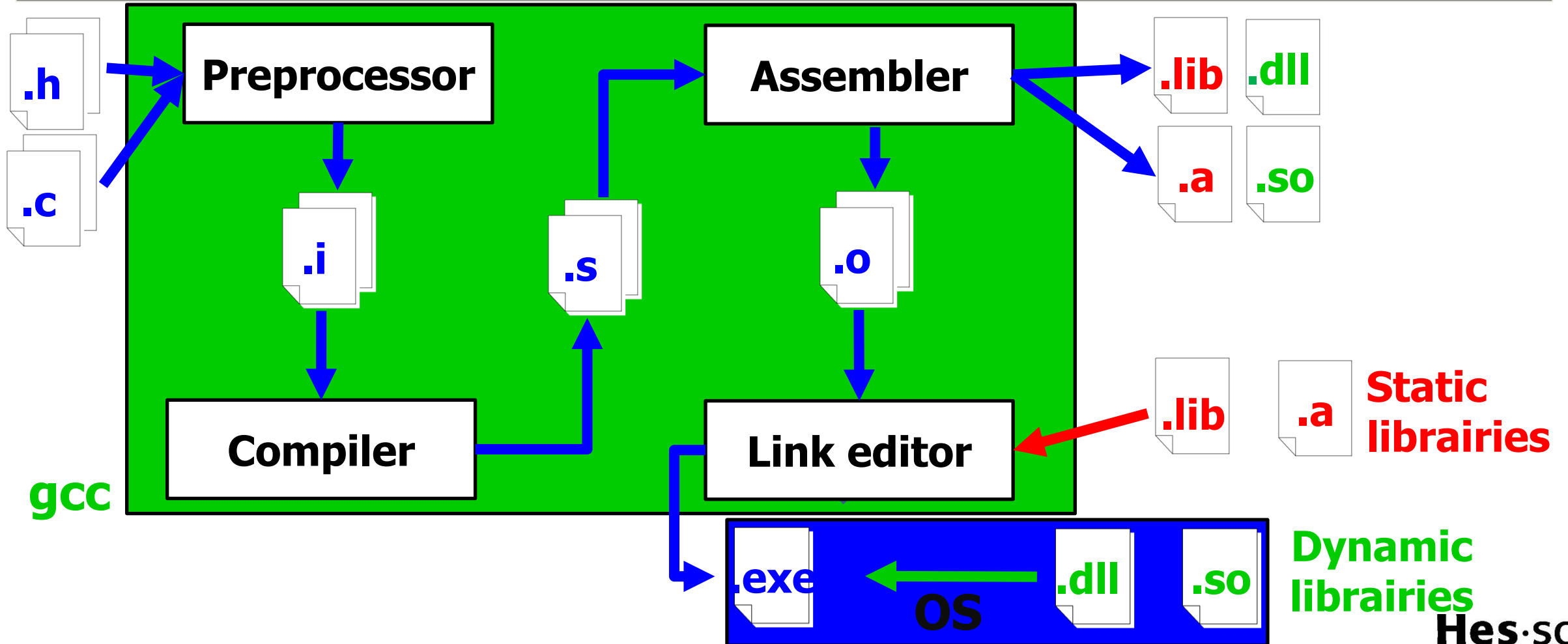
Un fichier *exécutable*

Une bibliothèque de fonctions

Plan

1. Compilateur
 1. Prétraitement
 2. Compilation
 3. Assemblage
 4. Editeur de liens
- 2. L'outil gcc**
3. Modularisation
4. L'outil **make**

7.2 Étapes de compilation avec gcc



7.2 Les extensions avec gcc

- .c** pour les fichiers **sources**
- .h** pour les d'**en-têtes** («*headers*»)
- .i** les fichiers prétraités par le préprocesseur
- .s** les fichiers en code assembleur
- .o** les fichiers **objets** en code machine
- .exe** les **exécutables Windows**

- .a** Bibliothèques statiques Linux (archive de fichiers objets)
- .lib** Bibliothèques statiques Windows («*Static Link Library*»)
- .dll** Librairies dynamiques Windows («*Dynamic Link Library*»)
- .so** Librairies dynamiques Linux («*Sharing Object* »)

Note : **gcc** est l'acronyme de « GNU Compiler Collection* »

7.2 Le compilateur gcc

Option	
<code>--version</code>	Affiche le numéro de version et les droits d'auteur du gcc
<code>--help</code>	Imprimer sur la sortie standard une description des options de la ligne de commande comprises par gcc
<code>-o nom</code>	Nom du fichier de sortie
<code>-g</code>	Génère les informations symboliques de débogage
<code>-Wall</code>	Affiche tous les warnings

```
gcc -Wall -g -o toto.exe toto.c
```

7.2 Le compilateur gcc

Option de gcc	préprocesseur	Compilateur	Assembleur	Editeur de Lien	Fichiers générés
-E	✓	✗	✗	✗	Texte (.i)
-S	✓	✓	✗	✗	Fichier assembleur (.s)
-c	✓	✓	✓	✗	Fichier objet (.o)
	✓	✓	✓	✓	Fichier exécutable (.exe)

7.2 Compilateur gcc

Création d'un **exécutable** à partir du fichier source `toto.c`

Directement

```
gcc -Wall -g -o toto.exe toto.c
```

En générant un fichier objet `toto.o`

1) Compilation

```
gcc -Wall -g -o toto.o -c toto.c
```

2) Édition des liens

```
gcc -Wall -g -o toto.exe toto.o
```

Plan

1. Compilateur
 1. Prétraitement
 2. Compilation
 3. Assemblage
 4. Editeur de liens
2. L'outil `gcc`
- 3. Modularisation**
4. L'outil `make`

7.3 Modularisation

PROGRAMME 1

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

```
void myHelloWorld(void);
```

```
int main(void)
{
    myHelloWorld();
    return 0;
}
```

```
void myHelloWorld(void)
{
    printf("hello world");
}
```

PROGRAMME 2

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

```
void myHelloWorld(void);
```

```
int main(void)
{
    myHelloWorld();
    myHelloWorld();
    return 0;
}
```

```
void myHelloWorld(void)
{
    printf("hello world");
}
```

7.3 Modularisation

PROGRAMME

```
#include <stdio.h>  
#include "module.h"  
void myHelloWorld(void);  
  
int main(void)  
{  
    myHelloWorld();  
    return 0;  
}  
  
void myHelloWorld(void)  
{  
    printf("hello world");  
}
```

module.h

```
void myHelloWorld(void);
```

module.c

```
#include <stdio.h>  
#include "module.h"  
  
void myHelloWorld(void)  
{  
    printf("hello world");  
}
```

PROTOTYPE

DEFINITION

MODULE

7.3 Modularisation

prog.c

```
#include "module1.h"
#include "module2.h"

int main(void)
{
    myHelloWorld1();
    myHelloWorld2();
    return 0;
}
```

MODULE 2

module1.h **MODULE 1**

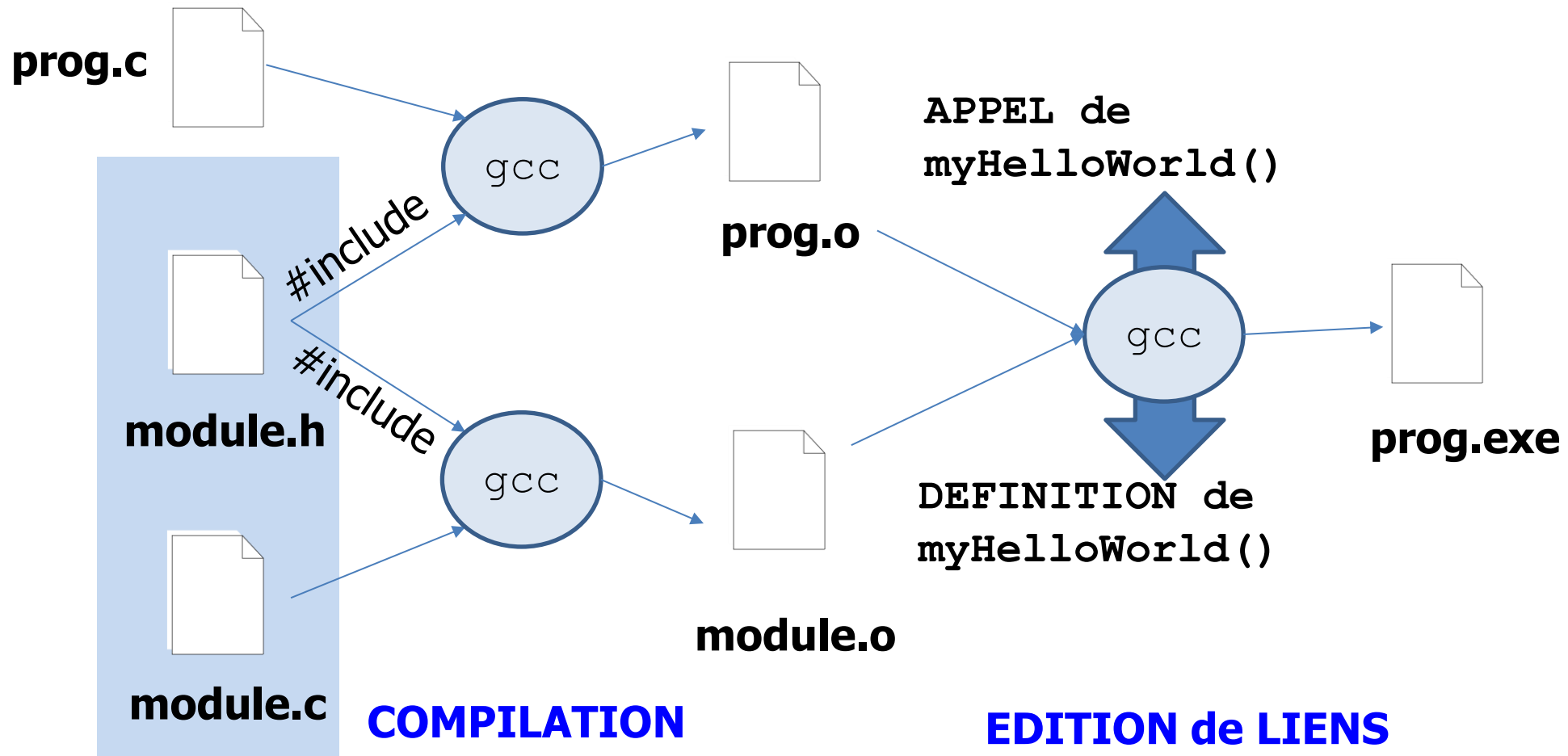
```
#ifndef _MODULE1_H
#define _MODULE1_H
void myHelloWorld1(void);
#endif
```

module1.c

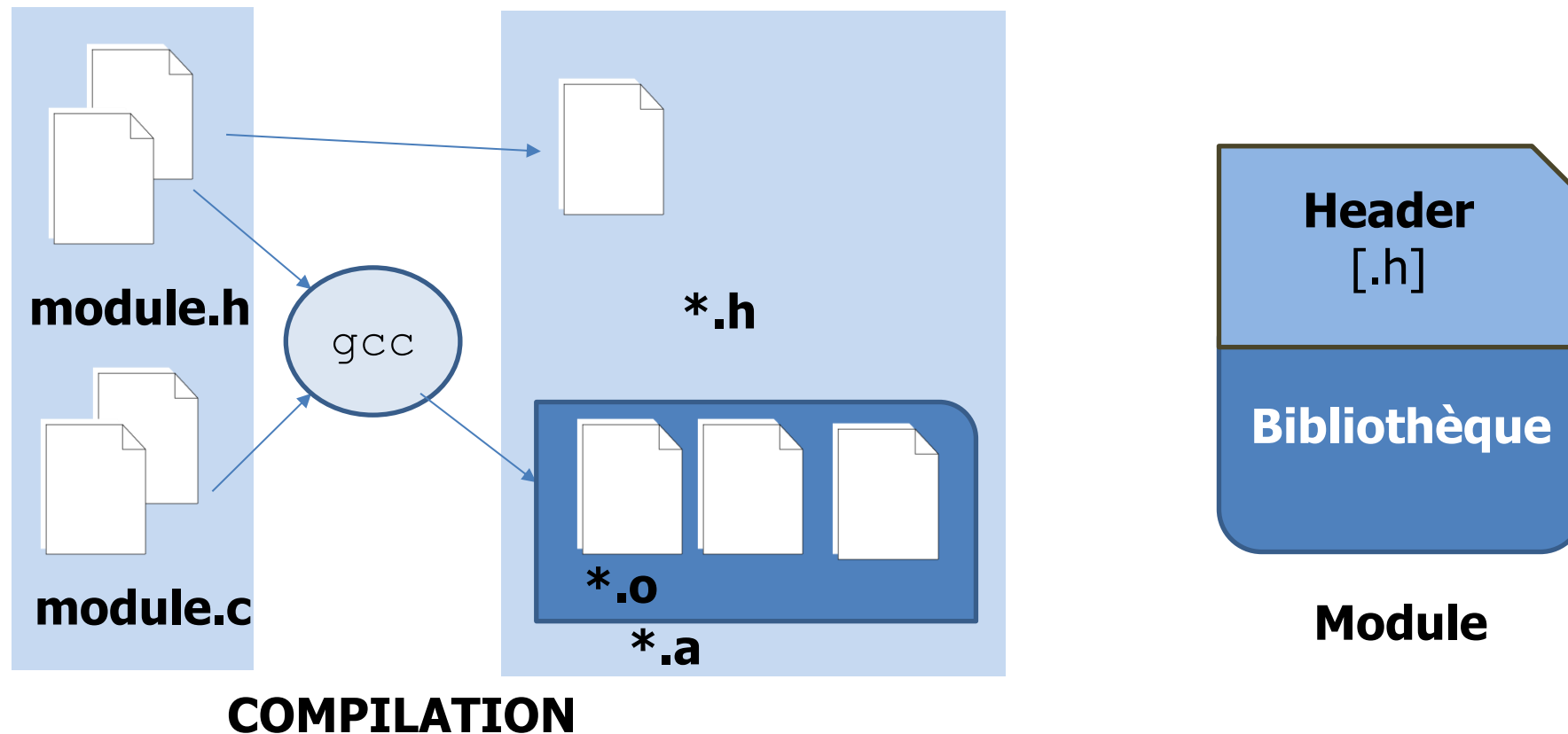
```
#include <stdio.h>
#include "module1.h"

void myHelloWorld1(void)
{
    printf("hello W.");
}
```

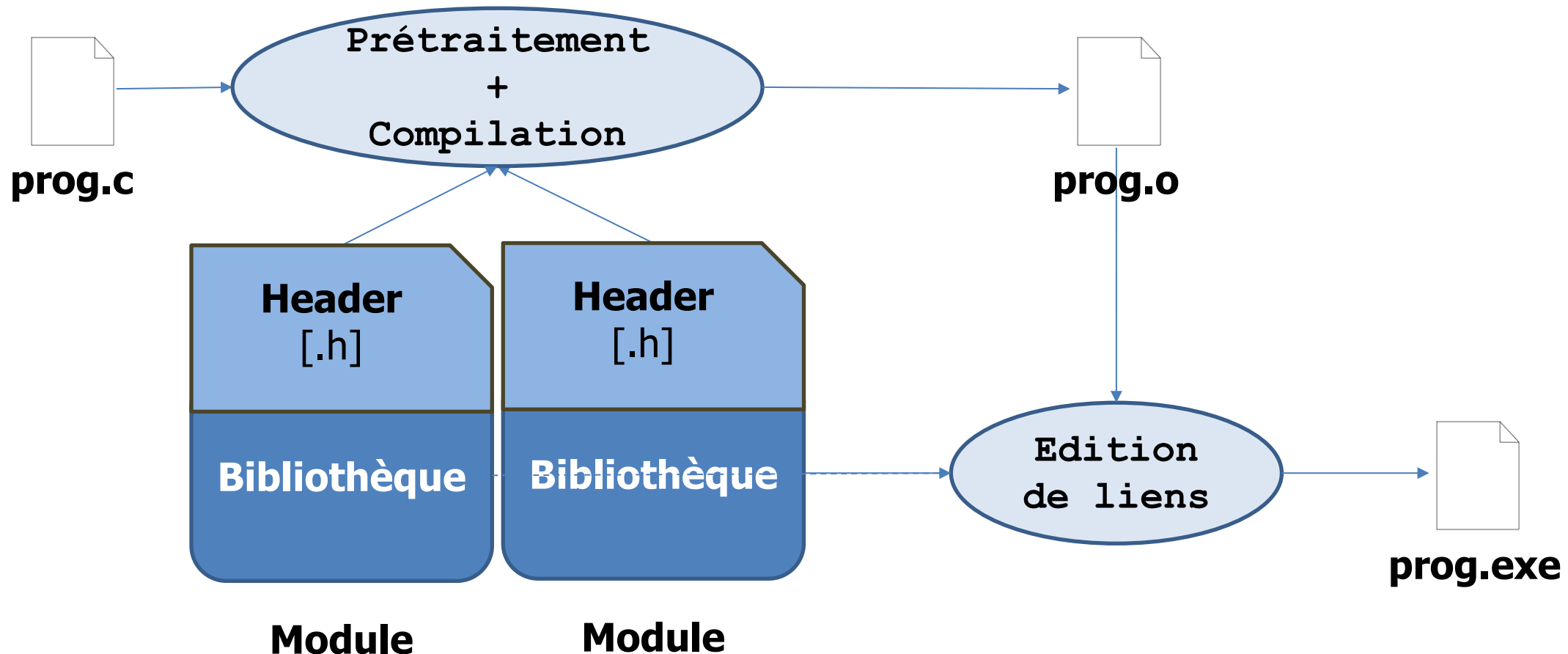
7.3 Modularisation : compilation directe



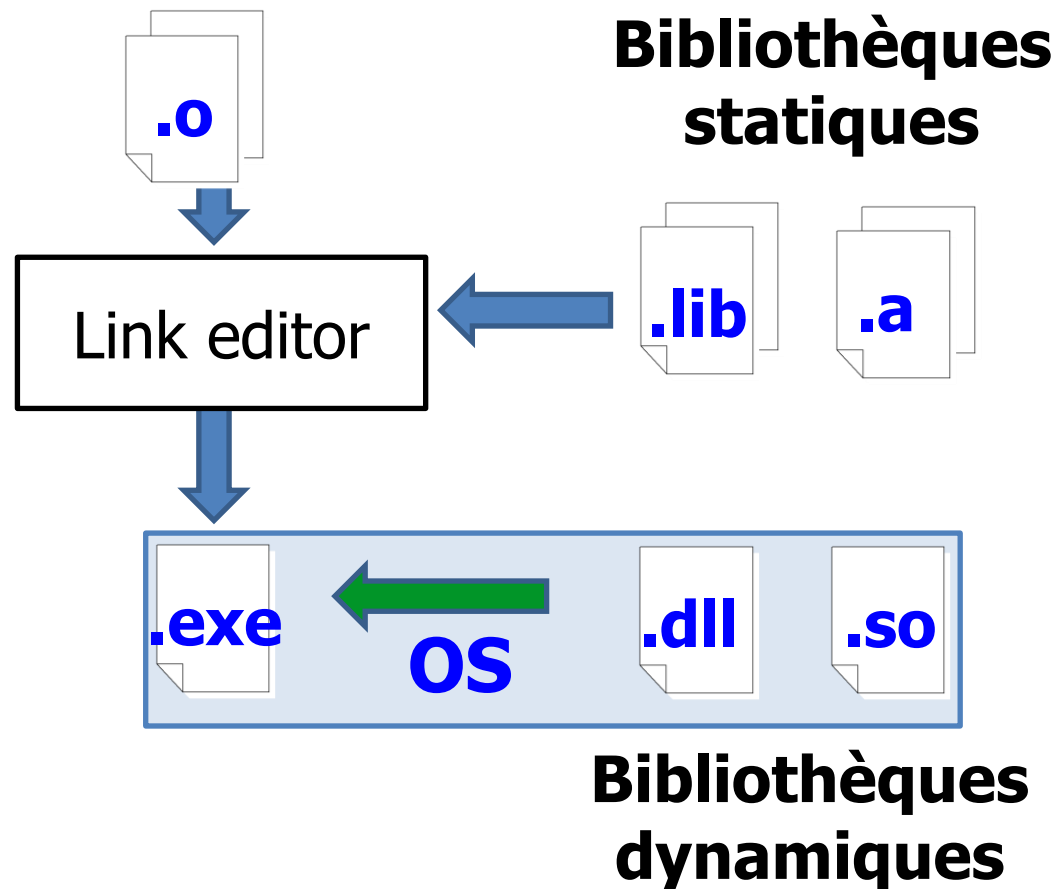
7.3 Modularisation : les bibliothèques



7.3 Modularisation : les bibliothèques



7.3 Modularisation : les bibliothèques



statique : le fichier objet et la bibliothèque sont liés dans le même fichier exécutable

dynamique : le fichier objet est lié avec la bibliothèque, mais les liens sont établis **lors du lancement de l'exécutable.**

7.3 Visibilité des variables "**extern**"

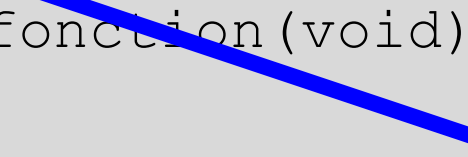
Lors de l'emploi d'une variable globale avec une compilation modulaire
Comment dire dans un fichier que l'on désire employer la même variable globale ?

main .c

```
void fonction(void);  
int n=3;  
int main(void)  
{  
    fonction();  
    n+=10;  
    return 0;  
}
```

extern int n;

```
void fonction(void)  
{  
    n++;  
}
```



fonction.c

Déclaration sans
définition de la
variable **n** qui
indique qu'elle est
définie ailleurs

7.3 Visibilité des fonctions

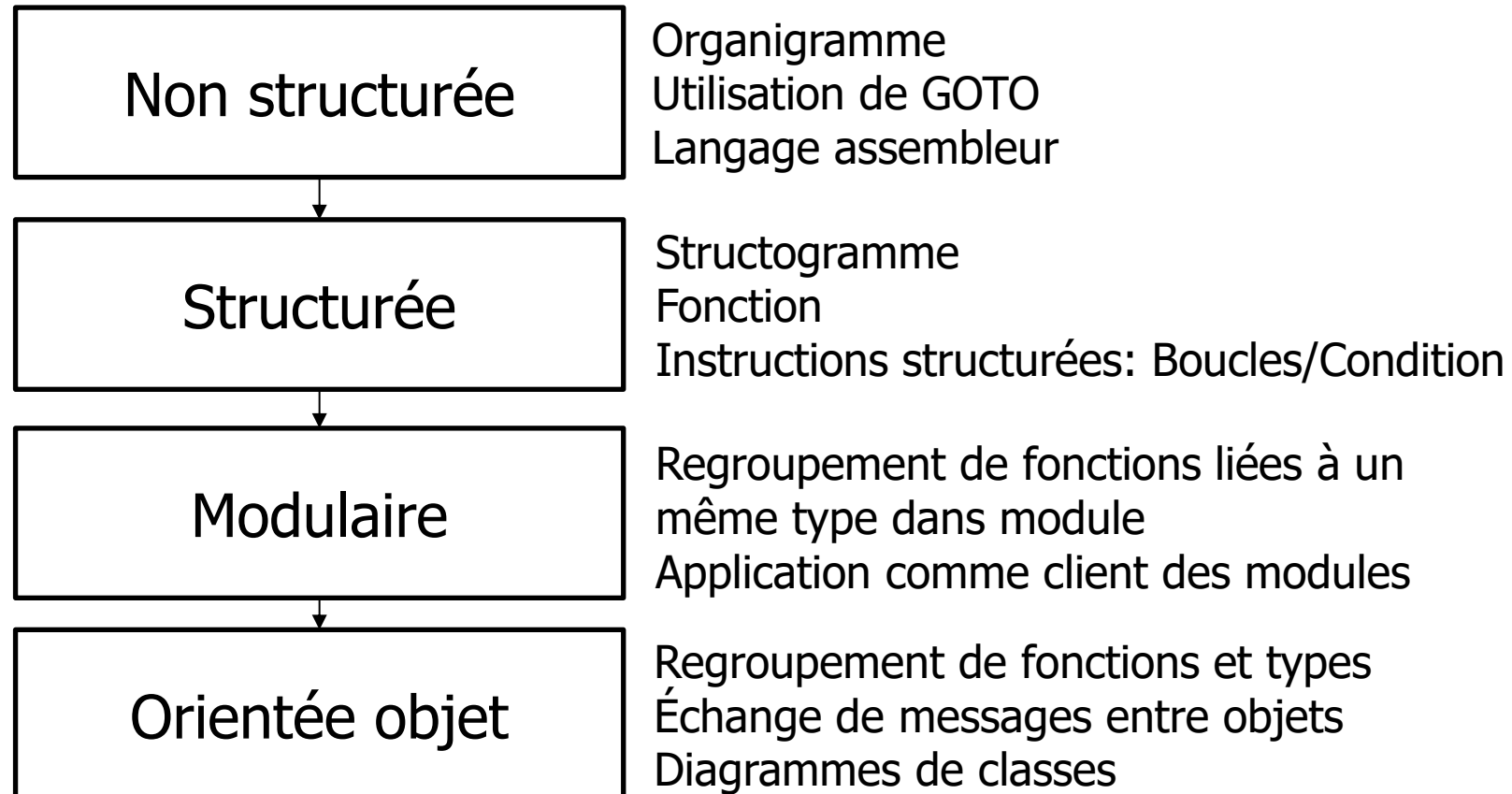
Une fonction déclarée avec le mot clé **static** n'est utilisable que dans le fichier où elle est déclarée

Par défaut, une fonction est **globale**. Cela signifie qu'elle est accessible de n'importe quelle fonction, et même depuis un autre module

Si la fonction doit être **privée** au module, on ajoutera donc le mot-clé **static** devant la fonction (pour cacher son implémentation par exemple)

7.3 Modularisation

Évolution des paradigmes de programmation



7.3 Modularisation

But : gérer des projets informatiques d'envergure impliquant plusieurs équipes de développeurs

Décomposition d'un code en modules

Organisation **hiérarchique** des modules

Regroupement par thèmes dans des modules

Encapsulation : cacher le code, donner accès par des headers

Type abstrait : cacher, si possible, les types de données

Développement et teste de chaque module indépendamment

Autorise la modification du cœur d'un module sans en toucher l'interface.

Difficile d'appliquer un module à d'autres données

Plan

1. Compilateur
 1. Prétraitement
 2. Compilation
 3. Assemblage
 4. Editeur de liens
2. L'outil `gcc`
3. Modularisation
- 4. L'outil `make`**

7.4 L'outil make

Make est un script basé sur des dépendances qui permet :

- de compiler automatiquement des programmes
- de n'exécuter que des commandes **nécessaires**

```
> make -f makefile
```

```
> make
```

Le fichier **makefile** décrit :

- les **dépendances** qu'il y a entre les fichiers intervenants dans la construction d'un exécutable (*graphe de dépendances*)
- les **commandes** (gcc, OS, ou autres) à lancer pour construire l'exécutable

7.4 Make : règles de dépendances explicites

Une règle de dépendance est composée de trois parties

1. une cible => *target*
2. une liste de dépendances => sous-buts
3. des commandes de mise à jour

La syntaxe

```
< cible > : < liste de dépendances >  
< Tabulation > < liste des commandes >
```

7.4 Make : exemple

makefile

```
all : prog.o module.o
      gcc prog.o module.o -o prog.exe
prog.o : prog.c
      gcc -c prog.c -o prog.o
module.o : module.c module.h
      gcc -c module.c -o module.o
```

> make module.o

1. Va à la cible **module.o**
2. Regarde si **module.c** / **module.h** ont été modifiés
 - Si oui : **gcc**
 - Si non : rien

> make all

1. Va à la cible **all**:
2. Regarde si **prog.o** / **module.o** ont été modifiés
 - Si oui : va à **prog.o** et/ou **module.o**
 - Si non : rien

Par défaut, le compilateur et le make se trouvent dans le dossier
`C:\Program Files (x86)\CodeBlocks\MinGW\bin\`

Make

`mingw32-make.exe` (minGW)
`make` (ailleurs)

Pour lancer `make` avec les valeurs par défaut, fichier `makefile`

```
>mingw32-make.exe
```

Pour lancer make et exécuter un autre fichier

```
>mingw32-make.exe -f fichier
```



Série 7.1



Tuto sur Make (PDF)