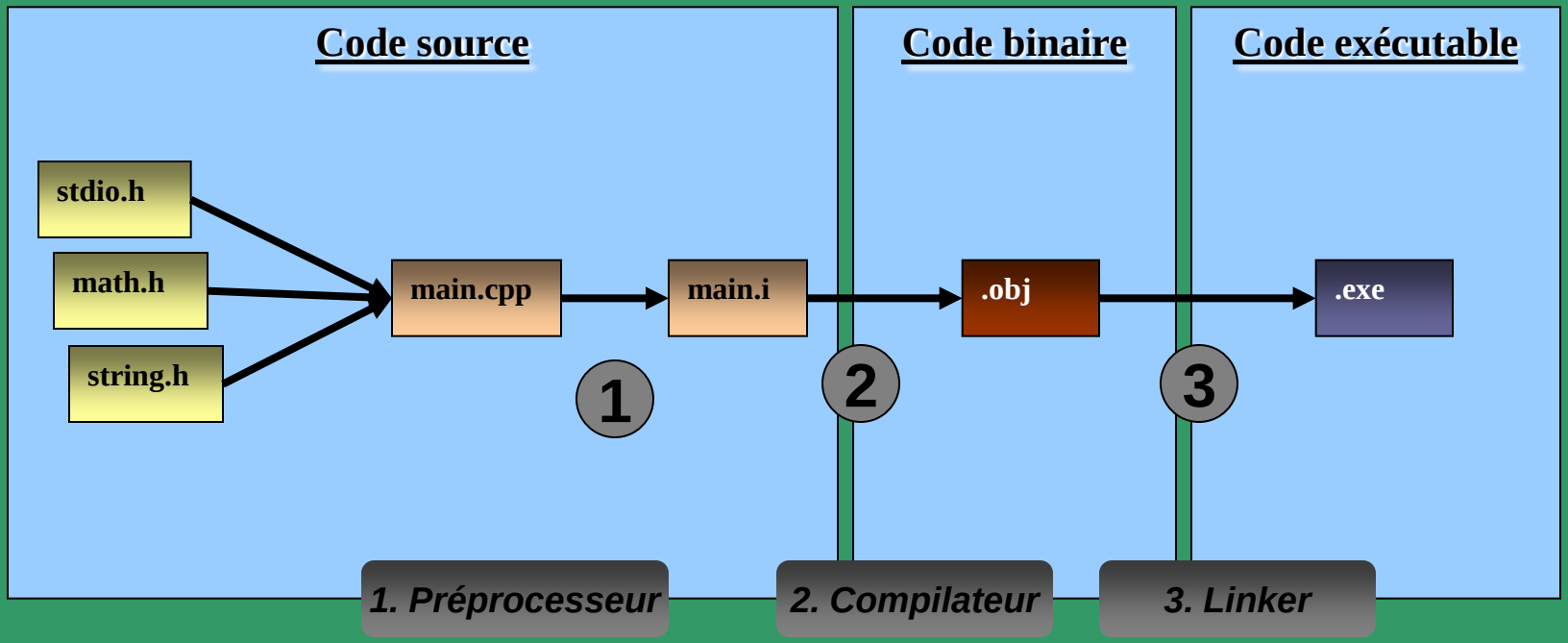
La modularisation d'un projet C/C++

Nicolas Gazères

Développeur Dassault Systèmes
ngs@3ds.com

Compilation d'un projet mono-source

Rappel du cours précédent



Limitations de l'approche mono-source

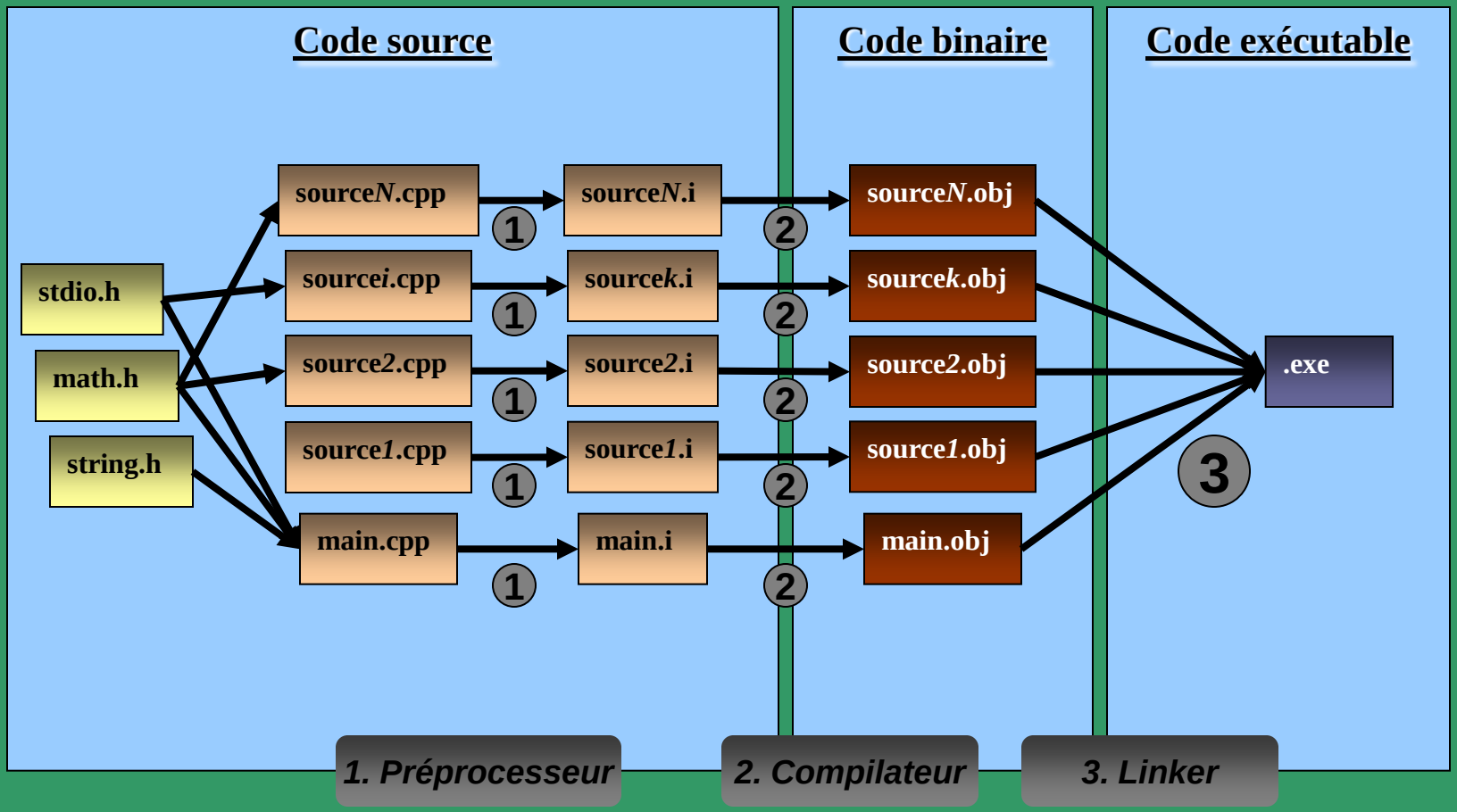
- Approche mono-source
  - Un seul fichier source (.c ou .cpp)
    - contenant le *main()*
    - et toutes les fonctions appelées par le *main()*

- Inconvénients (en contexte industriel)
  - *Un seul fichier énorme*
    → temps de compilation prohibitif
  - *Tout le monde travaille sur le même fichier*
    → problème pratique
  - *Tout le monde voit le code de tout le monde*
    → confidentialité inexistante

- La solution: la modularisation du code…

Compilation d'un projet multi-source

Schéma général



- Le fichier-source qui contient le *main()* est maintenant accompagné d'autres fichiers-sources, contenant les autres fonctions du projet.

La Modularisation du Code

- Principe général
  - On partage le code source en plusieurs fichiers (.c ou .cpp)
  - Chaque fichier-source est préprocessé et compilé *séparément* .
  - La compilation de chaque source doit être correcte:
    - du point de vue syntaxique
    - du point de vue du contrôle de type
  - Tous les fichiers-objets résultants sont ensuite *regroupés* pour constituer l'exécutable.
    - Il ne doit pas rester de « trou » dans l'exécutable.

- Les avantages:
  - Réduction des temps de compilation.
  - Commodité pour le développeur humain.
  - Aspect industriel:
    - protéger les secrets de fabrication en évitant de fournir le code sous forme lisible (voir cours DLL).

Conséquence:
Introduction des headers perso

- Quasiment tout projet utilise certaines des fonctions de la librairie standard (*printf*, *scanf*, *sin*, *malloc*, …)
  - Il doit donc inclure les headers standards qui *déclarent* ces fonctions.
    - *<stdio.h>*, *<math.h>*, *<stdlib.h>*, *<malloc.h>*

    … faute de quoi la compilation échoue.
  - Exemple: stdio.h → stdio.h
- Le même besoin se présente naturellement pour les fonctions spécifiques à un projet:
  - *Comment utiliser une fonction spécifique au projet dans plusieurs fichiers-sources du projet ?*
    Réponse: Un programmeur peut avoir besoin d'introduire son(ses) propre(s) header(s).
  - → Utilité des *déclarations de fonctions*, vues précédemment.

Introduction d'un header perso

Sans modularisation

main.cpp

```
int encadreChaine( char * p ) {
   // Affiche la chaine p
   // encadrée par des ****
   …
}

int main() {
   encadreChaine( "Debut \n" );
   encadreChaine( "Fin \n" );
   return 0;
}
```

Avec modularisation

monHeader.h

```
// Déclaration de la fonction
int encadreChaine( char * p );
```

encadreChaine.cpp

```
// Définition de la fonction
int encadreChaine( char * p ) {
   …   // Affiche la chaine p encadrée
}
```

main.cpp

```
#include "monHeader.h"

int main() {
   encadreChaine( "Debut \n" );
   encadreChaine( "Fin \n" );
   return 0;
}
```

Les directives conditionnelles
#ifdef, #ifndef, #else,#endif

- **Préprocessing conditionnel**
  - Seul un des deux blocs est conservé par le préprocesseur.

```
➡ #ifdef MON_SYMBOLE
        … bloc 1 …
  #else
        … bloc 2 …
  #endif
```

- **Au moment où il arrive à la flèche rouge**:
  - Si le préprocesseur a déjà rencontré MON_SYMBOLE, il garde le bloc 1.
  - Sinon, il garde le bloc 2.

- MON_SYMBOLE a pu être défini:
  - Par une directive #define, ie. *#define MON_SYMBOLE 1*
  - Par des options de compilation (cf. préprocessing sous Visual Studio).

- Il existe aussi la directive ***#ifndef*** (qui fait l'inverse de #ifdef).

Préprocessing d'un Fichier Source

Eviter les effets indésirables des includes multiples

- Il est très fréquent qu'un header soit inclus plusieurs fois dans le même fichier source (en général indirectement).

- Cela pose problème dans le cas d'une *définition* multiple
  - de variable globale,
  - de classe,
  - de template,
  - …

- En effet, le compilateur interdit la *définition* multiple de symbole
  - attention: la *déclaration* multiple est, elle, parfaitement admise.

- Pour éviter la définition multiple, tout en autorisant les inclusions multiples, deux solutions:
  - les includes guards (Windows, Unix…)
  - la directive "#pragma once" (Windows uniquement)

Préprocessing d'un Fichier Source

Les « include guards »

- Une technique basée sur les directives conditionnelles.
- On conditionne le corps du header par un symbole du préprocesseur.
- Chaque header du projet doit être gardé par un symbole unique.

mon_header.h

```
#ifndef MON_HEADER_H
#define MON_HEADER_H

// texte du header
…

#endif
```

Préprocessing d'un Fichier Source

Exemple d'« include guards »

mon_header.h

```
#ifndef MON_HEADER_H
#define MON_HEADER_H

// texte du header

#endif
```

mon_source.cpp

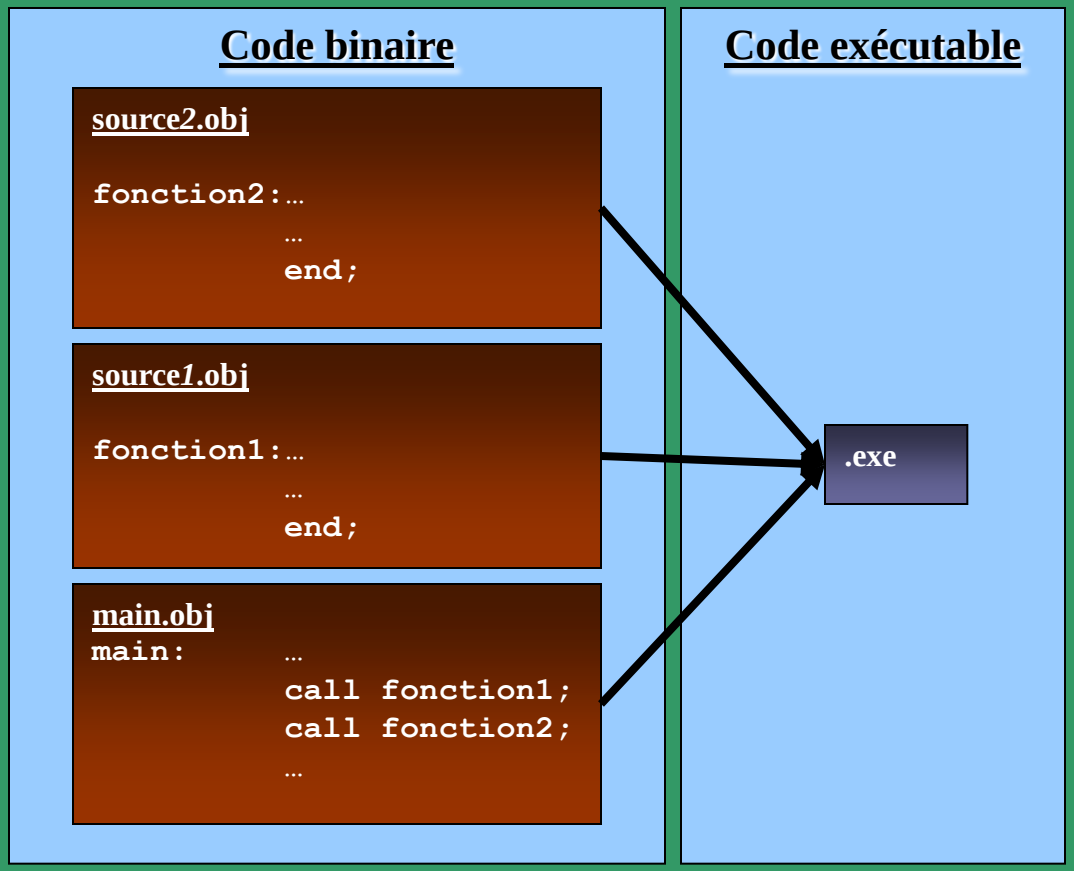
```
#include "mon_header.h"
#define MON_HEADER_H

// texte du header

#endif

#include "mon_header.h"
#define MON_HEADER_H

// texte du header

#endif

int main() {
    …
}
```

Préprocessing d'un Fichier Source
« *#pragma once* »

- Directive *#pragma*
  - Une directive *#pragma* est une consigne donnée au préprocesseur qui est plateforme-spécifique.
  - Si la plateforme ne reconnaît pas l'expression qui suit *#pragma*, la directive est tout simplement ignorée
    - sans erreur de compilation.
- ***#pragma once***
  - Lorsque la ligne « #pragma once » apparait dans un header, elle signifie « ce header ne peut être inclus qu'une seule fois dans une unité de traduction. »
  - *Avantage:* beaucoup plus simple à utiliser que les include guards
  - *Inconvénient:* spécifique Windows
- **Mode d'emploi**
  - il suffit de placer une directive « #pragma once » sur la première ligne de chaque header.

L'édition des liens

- Dans un projet *modularisé*, au moins un fichier-objet contient au moins un « trou » (*fonction*, *variable*, …)



L'édition des liens (en C)

- D'un fichier-objet qui ne définit pas tous les symboles dont il a besoin, on dit qu'il a une "*dépendance externe*".
  - Une dépendance externe peut être satisfaite par un autre fichier objet du même projet (ou par une DLL, cf. cours suivant).

- Une fonction (ou une variable) qui fait l'objet de dépendances externes:
  - doit être *définie exactement une fois* dans le projet .
  - doit être *exportée* par *le* fichier objet qui la définit.
  - En C, une fonction (ou une variable) est exportée par défaut.
    - Mais on peut en limiter la visibilité externe en la déclarant ***static***.
      (les symboles d'un source n'ont pas tous vocation à être exportés)

- Recommandation:
  - Dans chaque fichier source qui fait référence à une fonction (ou une variable) définie ailleurs, déclarer cette fonction ou cette variable à l'aide du qualificateur *extern*.

L'édition des liens (en C)

Exemple

source1.cpp

```
// par k, source1.obj a une dependance externe sur source2.obj
extern int k;

int somme( int a, int b ) {          // cette fonction n'est pas static,
 return a+b+k ;                      // elle est exportee par source1.obj
}

int i=4;                             // ceci est la définition (unique) de i
```

source2.cpp

```
int k=37;                            // ceci est la definition (unique) de k

// ces declarations extern de i et de somme() permettent de
// satisfaire la dependance externe que source2.obj a sur source1.obj
extern int i;
extern int somme( int a, int b ) ;

static int fonctionCachee() {        // fonctionCachee declaree static,
                                     // donc pas visible de source1.obj
  int j=9 ;
  return somme(i, j) ;               // utilise le i de source1.obj
}
```

Les dépendances de Build

- **Dès qu'un *header* est modifié** (même très peu), tous les fichiers source (.c ou .cpp) qui l'incluent
  - sont considérés comme non-à-jour
  - doivent être recompilés.

- **Dès qu'un *fichier-source* est modifié**, chaque fichier-objet (.obj/.o) généré à partir de lui
  - est considéré comme non-à-jour
  - doit être reconstruit.

- **Dès qu'un *fichier-objet* est modifié**, tout exécutable qui l'inclut:
  - est considéré comme non-à-jour
  - doit être relinkée.

- **Conclusion**
  - Dès qu'un header change, tous les exécutables qui en dépendent doivent être reconstruits.
    → coût élevé en temps de compilation et d'édition des liens.

Recommandations
Modularisation

- On utilise les headers pour représenter *physiquement* (ie. par un fichier) la structure *logique* du projet (ie. la modélisation):
  - Un header correspond à une classe et aux déclarations très apparentées à cette classe.
  - Grosso modo: une classe ←→ un header
- En général, un header qui *déclare* des symboles est #inclus par le source qui *définit* des symboles.
- Éviter absolument les *définitions* de fonctions dans les headers.
  - Un header est fait pour contenir des *déclarations*.
    - Seule exception: fonctions inline (usage avancé)
- Éviter absolument les *variables globales*
  - Les *constantes globales*, elles, sont tout-à-fait utiles et permises.

Recommandations
Modularisation

- Utiliser systématiquement les « include guards »
  - Une seule include guard par header
  - L' include guard doit contenir tout le header.
  - le symbole de l'include guard doit être calqué sur celui du header
    - Ex. pour le fichier *monHeader.h*, on peut utiliser l'include guard *MON_HEADER_H*.
  - éventuellement complétées par des « #pragma once » (Windows-only)

- Les headers doivent être bien-formés et minimaux:
  - Ne pas #inclure de fichiers d'extension *.cpp*
  - #inclure le minimum de headers.

- Les headers doivent être autosuffisants:
  - Quand un symbole *X* apparaît dans un header *Y.h*, il faut
    - tenter de déclarer *X* par une *forward declaration* si c'est possible
    - Si cela ne suffit pas, alors on #inclura le header qui déclare *X* (ie. *X.h*).

Fin