

Plan

Modules

- Notion de modularité

- Découpage d'un projet

- Fichiers sources / d'en-tête

- Création de modules

- Graphes d'inclusions

- Erreurs courantes et bonnes habitudes

Plan

Modules

Notion de modularité

Découpage d'un projet

Fichiers sources / d'en-tête

Création de modules

Graphes d'inclusions

Erreurs courantes et bonnes habitudes

Modules

Modulariser un projet signifie le découper de manière cohérente en plusieurs parties plus petites.

Modules

Modulariser un projet signifie le découper de manière cohérente en plusieurs parties plus petites.

Un **module** est un ensemble de données et d'instructions qui permettent de gérer une partie bien ciblée d'un projet.

Modules

Modulariser un projet signifie le découper de manière cohérente en plusieurs parties plus petites.

Un **module** est un ensemble de données et d'instructions qui permettent de gérer une partie bien ciblée d'un projet.

Il existe deux manières de concevoir un projet :

1. programmer dans un unique fichier contenant tout le code nécessaire;

Modules

Modulariser un projet signifie le découper de manière cohérente en plusieurs parties plus petites.

Un **module** est un ensemble de données et d'instructions qui permettent de gérer une partie bien ciblée d'un projet.

Il existe deux manières de concevoir un projet :

1. programmer dans un unique fichier contenant tout le code nécessaire;
2. programmer dans divers fichiers qui fractionnent le projet en plusieurs sous-parties.

Modules

Modulariser un projet signifie le découper de manière cohérente en plusieurs parties plus petites.

Un **module** est un ensemble de données et d'instructions qui permettent de gérer une partie bien ciblée d'un projet.

Il existe deux manières de concevoir un projet :

1. programmer dans un unique fichier contenant tout le code nécessaire;
2. programmer dans divers fichiers qui fractionnent le projet en plusieurs sous-parties.

À partir de maintenant, on adoptera la 2^e manière.

Modules

Modulariser un projet signifie le découper de manière cohérente en plusieurs parties plus petites.

Un **module** est un ensemble de données et d'instructions qui permettent de gérer une partie bien ciblée d'un projet.

Il existe deux manières de concevoir un projet :

1. programmer dans un unique fichier contenant tout le code nécessaire;
2. programmer dans divers fichiers qui fractionnent le projet en plusieurs sous-parties.

À partir de maintenant, on adoptera la 2^e manière.

Il reste à savoir comment **découper un projet de manière cohérente** et comment **utiliser les outils offerts par le langage** pour gérer ce découpage.

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

1. La **lisibilité** du code est accrue, ainsi que la facilité de son **entretien**.

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

1. La **lisibilité** du code est accrue, ainsi que la facilité de son **entretien**.
2. Permet de **regrouper** les types et les fonctions selon leurs objectifs.

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

1. La **lisibilité** du code est accrue, ainsi que la facilité de son **entretien**.
2. Permet de **regrouper** les types et les fonctions selon leurs objectifs.
3. Il devient possible de **réutiliser** dans un nouveau projet un module créé dans un projet antérieur.

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

1. La **lisibilité** du code est accrue, ainsi que la facilité de son **entretien**.
2. Permet de **regrouper** les types et les fonctions selon leurs objectifs.
3. Il devient possible de **réutiliser** dans un nouveau projet un module créé dans un projet antérieur.
4. Permet de **cacher des fonctions** (notion de fonctions privées).

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

1. La **lisibilité** du code est accrue, ainsi que la facilité de son **entretien**.
2. Permet de **regrouper** les types et les fonctions selon leurs objectifs.
3. Il devient possible de **réutiliser** dans un nouveau projet un module créé dans un projet antérieur.
4. Permet de **cache des fonctions** (notion de fonctions privées).
5. Facilite le **travail par équipe**.

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

1. La **lisibilité** du code est accrue, ainsi que la facilité de son **entretien**.
2. Permet de **regrouper** les types et les fonctions selon leurs objectifs.
3. Il devient possible de **réutiliser** dans un nouveau projet un module créé dans un projet antérieur.
4. Permet de **cacher des fonctions** (notion de fonctions privées).
5. Facilite le **travail par équipe**.
6. Permet de rendre la **compilation localisée** (compilation module par module).

Plan

Modules

Notion de modularité

Découpage d'un projet

Fichiers sources / d'en-tête

Création de modules

Graphes d'inclusions

Erreurs courantes et bonnes habitudes

Spécification d'un projet

Considérons le **projet spécifié** de la manière suivante :

Spécification d'un projet

Considérons le **projet spécifié** de la manière suivante :

- ▶ le but est de fournir un programme qui permet de décider si des formules logiques sans quantificateur sont valides ou contradictoires.

Spécification d'un projet

Considérons le **projet spécifié** de la manière suivante :

- ▶ le but est de fournir un programme qui permet de décider si des formules logiques sans quantificateur sont valides ou contradictoires.
- ▶ La syntaxe d'une formule est la suivante : on dispose du jeu de formules atomiques a, b, \dots, z et on écrit les formules de manière infixe et totalement parenthésée. Par exemple, la formule $(A \rightarrow (B \vee \neg C)) \wedge A$ s'écrit $((a \text{ IMP } (b \text{ OU } (\text{NON } c))) \text{ ET } a)$.

Spécification d'un projet

Considérons le **projet spécifié** de la manière suivante :

- ▶ le but est de fournir un programme qui permet de décider si des formules logiques sans quantificateur sont valides ou contradictoires.
- ▶ La syntaxe d'une formule est la suivante : on dispose du jeu de formules atomiques a, b, \dots, z et on écrit les formules de manière infixe et totalement parenthésée. Par exemple, la formule $(A \rightarrow (B \vee \neg C)) \wedge A$ s'écrit ((a IMP (b OU (NON c))) ET a).
- ▶ L'interaction utilisateur / programme se fait de la manière suivante :
 1. l'utilisateur fournit un fichier en entrée contenant une formule par ligne;
 2. le programme produit un fichier en sortie contenant, ligne par ligne, la réponse **erreur** si la formule correspondante est syntaxiquement erronée ou bien **valide**, **contradictoire** ou **rien** selon la nature de la formule.

Découpage du projet

Il y a deux parties bien distinctes dans ce projet :

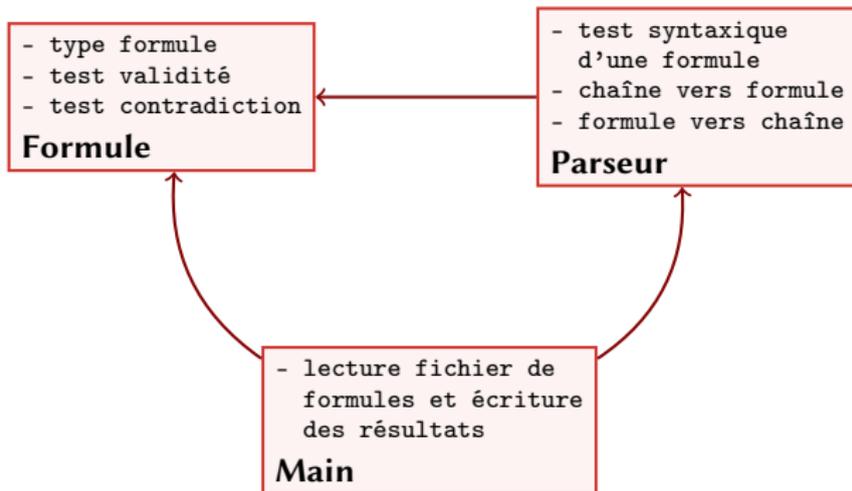
1. la **représentation des formules** et le test de validité / contradiction ;
2. la **gestion syntaxique des formules** (lecture / écriture d'une formule dans un fichier).

Découpage du projet

Il y a deux parties bien distinctes dans ce projet :

1. la **représentation des formules** et le test de validité / contradiction;
2. la **gestion syntaxique des formules** (lecture / écriture d'une formule dans un fichier).

Ces deux parties dictent le découpage suivant :



Toute flèche $A \rightarrow B$ signifie que le module A **dépend** du module B .

Plan

Modules

Notion de modularité

Découpage d'un projet

Fichiers sources / d'en-tête

Création de modules

Graphes d'inclusions

Erreurs courantes et bonnes habitudes

Composition d'un module

Un module est composé de deux fichiers :

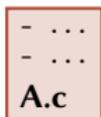
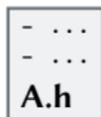
1. un fichier d'en-tête d'extension `.h`;
2. un fichier source d'extension `.c`.

Composition d'un module

Un module est composé de deux fichiers :

1. un **fichier d'en-tête** d'extension `.h`;
2. un **fichier source** d'extension `.c`.

Les noms de ces deux fichiers sont les mêmes (extension mise à part).

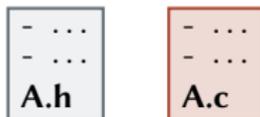


Composition d'un module

Un module est composé de deux fichiers :

1. un **fichier d'en-tête** d'extension `.h`;
2. un **fichier source** d'extension `.c`.

Les noms de ces deux fichiers sont les mêmes (extension mise à part).



Seul le module principal est constitué d'un seul fichier source `Main.c`.

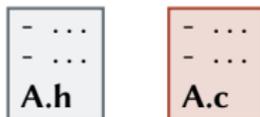


Composition d'un module

Un module est composé de deux fichiers :

1. un **fichier d'en-tête** d'extension `.h`;
2. un **fichier source** d'extension `.c`.

Les noms de ces deux fichiers sont les mêmes (extension mise à part).



Seul le module principal est constitué d'un seul fichier source `Main.c`.



Par exemple, le projet précédent est constitué des fichiers `Formule.h`, `Formule.c`, `Parseur.h`, `Parseur.c` et `Main.c`.

Fichiers d'en-tête

Un fichier d'en-tête contient des **déclarations** de types et de fonctions (prototypes).

Fichiers d'en-tête

Un fichier d'en-tête contient des **déclarations** de types et de fonctions (prototypes).

Les prototypes qui y figurent sont ceux des fonctions que l'on souhaite rendre visibles (utilisables) par d'autres modules.

Fichiers d'en-tête

Un fichier d'en-tête contient des **déclarations** de types et de fonctions (prototypes).

Les prototypes qui y figurent sont ceux des fonctions que l'on souhaite rendre visibles (utilisables) par d'autres modules.

Par exemple, un en-tête possible du module `Formule` est

```
/* Formule.h */

typedef struct {
    ...
} Form;

int est_valide(Form *f);
int est_contra(Form *f);
```

Fichiers d'en-tête

Les fichiers d'en-tête sont ceux que le programmeur regarde en premier pour connaître le rôle d'un type ou d'une fonction.

Fichiers d'en-tête

Les fichiers d'en-tête sont ceux que le programmeur regarde en premier pour connaître le rôle d'un type ou d'une fonction.

De ce fait, c'est dans les fichiers d'en-tête que l'on commente chaque **type et fonction** pour préciser leur rôle.

Fichiers d'en-tête

Les fichiers d'en-tête sont ceux que le programmeur regarde en premier pour connaître le rôle d'un type ou d'une fonction.

De ce fait, c'est dans les fichiers d'en-tête que l'on commente chaque type et fonction pour préciser leur rôle.

Par exemple, l'en-tête du module `Formule` devrait être de la forme

```
/* Formule.h */

/* Representation des formules logiques sans
 * quantificateur. */
typedef struct {
    ...
} Form;

/* Renvoie '1' si la formule pointee par 'f'
 * est valide et '0' sinon. */
int est_valide(Form *f);
...
```

Fichiers sources

Un fichier source contient une **implantation** de l'en-tête du module auquel il appartient.

Fichiers sources

Un fichier source contient une **implantation** de l'en-tête du module auquel il appartient.

Il contient de ce fait les définitions de fonctions déclarées dans l'en-tête.

Fichiers sources

Un fichier source contient une **implantation** de l'en-tête du module auquel il appartient.

Il contient de ce fait les définitions de fonctions déclarées dans l'en-tête.

Par exemple, une implantation possible du module `Formule` est

```
/* Formule.c */  
  
...  
int est_valide(Form *f) {  
    int i, j;  
    assert(f != NULL);  
    ...  
}  
  
int est_contra(Form *f) {  
    ...  
}
```

L'ordre de définition des fonctions n'est pas important. Nous en verrons la raison dans la suite.

Fichiers sources

Il faut impérativement que toutes les fonctions déclarées dans le fichier d'en-tête du module soient **définies** dans le fichier source correspondant.

Fichiers sources

Il faut impérativement que toutes les fonctions déclarées dans le fichier d'en-tête du module soient **définies** dans le fichier source correspondant.

Il est en revanche possible de définir dans un fichier source des fonctions qui ne sont pas déclarées dans l'en-tête correspondant.

Fichiers sources

Il faut impérativement que toutes les fonctions déclarées dans le fichier d'en-tête du module soient **définies** dans le fichier source correspondant.

Il est en revanche possible de définir dans un fichier source des fonctions qui ne sont pas déclarées dans l'en-tête correspondant.

Ainsi,

- ▶ une fonction définie dans un fichier source mais pas dans l'en-tête correspondant s'appelle **fonction privée**;

Fichiers sources

Il faut impérativement que toutes les fonctions déclarées dans le fichier d'en-tête du module soient **définies** dans le fichier source correspondant.

Il est en revanche possible de définir dans un fichier source des fonctions qui ne sont pas déclarées dans l'en-tête correspondant.

Ainsi,

- ▶ une fonction définie dans un fichier source mais pas dans l'en-tête correspondant s'appelle **fonction privée**;
- ▶ l'intérêt des fonctions privées est d'être des **fonctions outils** dont le champ d'application est local au module dans lequel elles sont définies.

On ne souhaite pas les rendre utilisables en dehors.

Fichiers sources et fonctions privées

Une fonction privée se définit avec le mot clé `static` (à ne pas confondre avec le `static` pour la déclaration de variables).

Fichiers sources et fonctions privées

Une fonction privée se définit avec le mot clé `static` (à ne pas confondre avec le `static` pour la déclaration de variables).

Par exemple, on peut avoir besoin d'une fonction privée appartenant au module `Formule` qui permet de compter le nombre d'occurrences d'un atome dans une formule.

Fichiers sources et fonctions privées

Une fonction privée se définit avec le mot clé `static` (à ne pas confondre avec le `static` pour la déclaration de variables).

Par exemple, on peut avoir besoin d'une fonction privée appartenant au module `Formule` qui permet de compter le nombre d'occurrences d'un atome dans une formule.

On la définit alors dans `Formule.c` par

```
static int nb_occ(Form *f, char atome) {  
    ...  
    assert(f != NULL);  
    assert(('a' <= atome) && (atome <= 'z'));  
    ...  
}
```

Fichiers sources et fonctions privées

Une fonction privée se définit avec le mot clé `static` (à ne pas confondre avec le `static` pour la déclaration de variables).

Par exemple, on peut avoir besoin d'une fonction privée appartenant au module `Formule` qui permet de compter le nombre d'occurrences d'un atome dans une formule.

On la définit alors dans `Formule.c` par

```
static int nb_occ(Form *f, char atome) {  
    ...  
    assert(f != NULL);  
    assert(('a' <= atome) && (atome <= 'z'));  
    ...  
}
```

La **portée lexicale** de cette fonction s'étend à tout ce qui suit sa définition dans le fichier `Formule.c`. Elle est invisible ailleurs.

Fichiers sources et fonctions privées

C'est un contresens que de définir une fonction dans un fichier source sans le mot clé `static` et sans l'avoir déclarée dans le fichier d'en-tête.

Fichiers sources et fonctions privées

C'est un contresens que de définir une fonction dans un fichier source sans le mot clé `static` et sans l'avoir déclarée dans le fichier d'en-tête.

C'est aussi un contresens que de définir dans un fichier source une fonction déclarée dans le fichier d'en-tête avec `static`.

Fichiers sources et fonctions privées

C'est un contresens que de définir une fonction dans un fichier source sans le mot clé `static` et sans l'avoir déclarée dans le fichier d'en-tête.

C'est aussi un contresens que de définir dans un fichier source une fonction déclarée dans le fichier d'en-tête avec `static`.

Ainsi, pour résumer, toute fonction définie dans un fichier source est

1. soit déclarée dans le fichier d'en-tête;

Fichiers sources et fonctions privées

C'est un contresens que de définir une fonction dans un fichier source sans le mot clé `static` et sans l'avoir déclarée dans le fichier d'en-tête.

C'est aussi un contresens que de définir dans un fichier source une fonction déclarée dans le fichier d'en-tête avec `static`.

Ainsi, pour résumer, toute fonction définie dans un fichier source est

1. soit déclarée dans le fichier d'en-tête;
2. soit non déclarée dans le fichier d'en-tête mais définie par `static`.

Allure d'un projet

Il y a deux manières d'organiser un projet en termes de fichiers et de répertoires :

Allure d'un projet

Il y a deux manières d'organiser un projet en termes de fichiers et de répertoires :

1. la 1^{re} consiste à regrouper l'ensemble des fichiers d'en-tête et des fichiers sources dans un même répertoire. Il y figure donc un nombre impair de fichiers (le module principal et les paires en-tête / source pour chaque module);

Allure d'un projet

Il y a deux manières d'organiser un projet en termes de fichiers et de répertoires :

1. la 1^{re} consiste à regrouper l'ensemble des fichiers d'en-tête et des fichiers sources dans un même répertoire. Il y figure donc un nombre impair de fichiers (le module principal et les paires en-tête / source pour chaque module);
2. la 2^e consiste à séparer les fichiers du projet en deux répertoires frères, `include` et `src`, le premier contenant les fichiers d'en-tête et l'autre, les fichiers sources et le module principal du projet.

Plan

Modules

Notion de modularité

Découpage d'un projet

Fichiers sources / d'en-tête

Création de modules

Graphes d'inclusions

Erreurs courantes et bonnes habitudes

Inclusion de modules

Pour utiliser un module `Module` dans un fichier `F`, on doit l'y **inclure**.

Inclusion de modules

Pour utiliser un module `Module` dans un fichier `F`, on doit l'y **inclure**.

On utilise pour cela dans `F` la commande pré-processeur

```
#include "Module.h"
```

Inclusion de modules

Pour utiliser un module `Module` dans un fichier `F`, on doit l'y **inclure**.

On utilise pour cela dans `F` la commande pré-processeur

```
#include "Module.h"
```

Celle-ci sera remplacée par le pré-processeur par le contenu de `Module.h`.

Inclusion de modules

Pour utiliser un module `Module` dans un fichier `F`, on doit l'y **inclure**.

On utilise pour cela dans `F` la commande pré-processeur

```
#include "Module.h"
```

Celle-ci sera remplacée par le pré-processeur par le contenu de `Module.h`.

Cette commande peut se trouver

- ▶ dans un fichier source pour bénéficier des fonctions définies et des types déclarés par le module;

Inclusion de modules

Pour utiliser un module `Module` dans un fichier `F`, on doit l'y **inclure**.

On utilise pour cela dans `F` la commande pré-processeur

```
#include "Module.h"
```

Celle-ci sera remplacée par le pré-processeur par le contenu de `Module.h`.

Cette commande peut se trouver

- ▶ dans un fichier source pour bénéficier des fonctions définies et des types déclarés par le module;
- ▶ dans un fichier d'en-tête pour bénéficier des types déclarés par le module.

Inclusion de modules

Par exemple, si `A` est un module définissant une fonction `f`, pour utiliser `f` dans un fichier `B.c` situé dans le même répertoire que `A.c` et `A.h`, on écrit

```
/* B.c */  
  
#include "A.h"  
...
```

Inclusion de modules

Par exemple, si `A` est un module définissant une fonction `f`, pour utiliser `f` dans un fichier `B.c` situé dans le même répertoire que `A.c` et `A.h`, on écrit

```
/* B.c */  
  
#include "A.h"  
...
```

Il est possible d'inclure à la suite plusieurs modules dans un même fichier :

```
/* Fichier.c ou Fichier.h */  
  
#include "A.h"  
#include "B.h"  
#include "C.h"  
...
```

Inclusion de modules

Par exemple, si A est un module définissant une fonction `f`, pour utiliser `f` dans un fichier `B.c` situé dans le même répertoire que `A.c` et `A.h`, on écrit

```
/* B.c */  
  
#include "A.h"  
...
```

Il est possible d'inclure à la suite plusieurs modules dans un même fichier :

```
/* Fichier.c ou Fichier.h */  
  
#include "A.h"  
#include "B.h"  
#include "C.h"  
...
```

De cette manière, `Fichier.c` ou `Fichier.h` bénéficie de tout ce qui est déclaré et défini dans les modules A, B et C.

Attention : les modules inclus ne doivent pas déclarer / définir des éléments d'un identificateur commun.

Inclusion de modules

Le fichier incluant n'a **accès** qu'au fichier d'en-tête du module, et donc qu'aux **déclarations effectuées**.

Inclusion de modules

Le fichier incluant n'a **accès** qu'au fichier d'en-tête du module, et donc qu'aux **déclarations effectuées**.

Le fichier incluant n'a pas besoin de connaître l'implantation du module.

Inclusion de modules

Le fichier incluant n'a **accès** qu'au fichier d'en-tête du module, et donc qu'aux **déclarations effectuées**.

Le fichier incluant n'a pas besoin de connaître l'implantation du module.

Considérons par exemple le module `Couple` défini par

```
/* Couple.h */

typedef int Couple[2];

int est_zero(Couple c);
void afficher(Couple c);
```

```
/* Couple.c */

...

int est_zero(Couple c) {
    return (c[0] == 0) && (c[1]== 0);
}

void afficher(Couple c) {
    printf("(%d, %d)", c[0], c[1]);
}
```

Inclusion de modules

Supposons que l'on ait besoin d'inclure le module `Couple` dans un fichier `Fichier.c`.

Inclusion de modules

Supposons que l'on ait besoin d'inclure le module `Couple` dans un fichier `Fichier.c`.

Le pré-processeur aura donc l'effet suivant sur `Fichier.c` :

```
/* Fichier.c avant
 * la passe du
 * pre-processeur */

#include "Couple.h"
...

if (!est_zero(c))
    afficher(c);
...
```

Inclusion de modules

Supposons que l'on ait besoin d'inclure le module `Couple` dans un fichier `Fichier.c`.

Le pré-processeur aura donc l'effet suivant sur `Fichier.c` :

```
/* Fichier.c avant
 * la passe du
 * pre-processeur */

#include "Couple.h"
...

if (!est_zero(c))
    afficher(c);
...
```

```
/* Fichier.c apres la passe
 * du pre-processeur */

typedef int Couple[2];
int est_zero(Couple c);
void afficher(Couple c);
...

if (!est_zero(c))
    afficher(c);
...
```

Inclusion de modules

Supposons que l'on ait besoin d'inclure le module `Couple` dans un fichier `Fichier.c`.

Le pré-processeur aura donc l'effet suivant sur `Fichier.c` :

```
/* Fichier.c avant
 * la passe du
 * pre-processeur */

#include "Couple.h"
...

if (!est_zero(c))
    afficher(c);
...
```

```
/* Fichier.c apres la passe
 * du pre-processeur */

typedef int Couple[2];
int est_zero(Couple c);
void afficher(Couple c);
...

if (!est_zero(c))
    afficher(c);
...
```

`Fichier.c` a besoin uniquement de connaître les types de retour et les signatures des fonctions qu'il invoque (connus à leur déclaration). Il n'a à ce stade pas besoin de connaître les définitions de ces fonctions.

Création complète d'un module

Pour créer un module `A`, il faut inclure son fichier d'en-tête `A.h` dans son fichier source `A.c`.

```
/* A.h */
```

```
...
```

```
/* A.c */
```

```
#include "A.h"
```

```
...
```

Création complète d'un module

Pour créer un module `A`, il faut inclure son fichier d'en-tête `A.h` dans son fichier source `A.c`.

```
/* A.h */
```

```
...
```

```
/* A.c */
```

```
#include "A.h"
```

```
...
```

De cette manière,

- ▶ d'une part, `A.c` a accès aux types et aux prototypes de fonctions déclarés dans `A.h`;

Création complète d'un module

Pour créer un module `A`, il faut inclure son fichier d'en-tête `A.h` dans son fichier source `A.c`.

```
/* A.h */
```

```
...
```

```
/* A.c */
```

```
#include "A.h"
```

```
...
```

De cette manière,

- ▶ d'une part, `A.c` a accès aux types et aux prototypes de fonctions déclarés dans `A.h`;
- ▶ d'autre part, cela permet d'implanter les fonctions déclarées dans `A.h` sans contrainte d'ordre.

Création complète d'un module

Considérons la situation suivante :

<pre>/* A.h */ ... #include "C.h" ...</pre>	<pre>/* A.c */ #include "A.h" ...</pre>	<pre>/* B.h */ ... #include "C.h" ...</pre>	<pre>/* B.c */ #include "B.h" ...</pre>
<pre>/* C.h */ ... int f(); ...</pre>	<pre>/* C.c */ #include "C.h" ...</pre>	<pre>/* D.c */ #include "A.h" #include "B.h" ...</pre>	

Création complète d'un module

Le pré-processeur transforme ces fichiers en

<pre>/* A.h */ ... int f(); ...</pre>	<pre>/* A.c */ /* Copie A.h */ ...</pre>	<pre>/* B.h */ ... int f(); ...</pre>	<pre>/* B.c */ /* Copie B.h */ ...</pre>
<pre>/* C.h */ ... int f(); ...</pre>	<pre>/* C.c */ /* Copie C.h */ ...</pre>	<pre>/* D.c */ int f(); int f(); ...</pre>	

Problème : le contenu de `C.h` est copié deux fois dans `D.c`. Ceci n'est pas accepté par le compilateur car il y a **multiple déclaration** d'un même symbole (`f` ici).

Création complète d'un module

La parade consiste à **inclure** un fichier d'en-tête de **manière conditionnelle** : on procède à l'inclusion que s'il n'a pas déjà été inclus.

On utilise pour cela les macro-instructions de contrôle de compilation `#ifndef` et `#endif` ainsi que `#define`.

Le schéma général est

```
/* A.h */
#ifndef __A__
#define __A__

    /* Declaration de types */

    /* Declaration de fonctions */

#endif
```

Ainsi, lors d'une inclusion de `A.h`, le pré-processeur vérifie si la macro `__A__` n'existe pas.

- ▶ Si elle n'existe pas, alors on la définit (`#define __A__`) et le contenu du module est pris en compte;
- ▶ sinon, cela signifie que le contenu a déjà été pris en compte. Celui-ci n'est pas repris en compte une 2^e fois.

Squelette d'un module

Pour résumer, tous les modules doivent avoir le squelette suivant :

```
/* A.h */
#ifdef __A__
#define __A__

/* Inclusions eventuelles
 * de modules */

/* Definitions eventuelles
 * de macros */

/* Declarations eventuelles
 * de types */

/* Declarations eventuelles
 * de fonctions */

#endif
```

```
/* A.c */
#include "A.h"

/* Inclusions eventuelles
 * de modules */

/* Definitions eventuelles
 * de fonctions privees */

/* Definitions de toutes les
 * fonctions declarees dans
 * le fichier d'en-tete */
```

Exemple du module Parseur

```
/* Parseur.h */
#ifndef __PARSEUR__
#define __PARSEUR__

#include "Formule.h"

/* Convertit la chaine de caracteres 'ch'ensee representer une formule
 * en une variable de type 'Form', qui va etre ecrite dans 'f'.
 * Renvoie '1' si 'ch' represente bien une formule et '0' sinon. */
int chaine_vers_form(Form *f, char *ch);

#endif
```

```
/* Parseur.c */
#include "Parseur.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int chaine_vers_form(Form *f, char *ch) {
    ...
    assert(f != NULL);
    ...
}
```

Plan

Modules

Notion de modularité

Découpage d'un projet

Fichiers sources / d'en-tête

Création de modules

Graphes d'inclusions

Erreurs courantes et bonnes habitudes

Graphes d'inclusions

On rappelle qu'un module **A** **dépend** d'un module **B** si le **fichier d'en-tête** de **A** inclut **B**.

Graphes d'inclusions

On rappelle qu'un module **A** **dépend** d'un module **B** si le **fichier d'en-tête** de **A** inclut **B**.

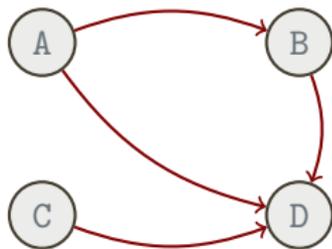
Pour visualiser l'allure d'un projet, on trace son **graphe d'inclusions**. On représente pour cela chacun des modules qui le composent dans des cercles (**sommets**) et on trace des flèches (**arcs**) de **A** vers **B** pour tout module **A** dépendant de **B**.

Graphes d'inclusions

On rappelle qu'un module A **dépend** d'un module B si le **fichier d'en-tête** de A inclut B.

Pour visualiser l'allure d'un projet, on trace son **graphe d'inclusions**. On représente pour cela chacun des modules qui le composent dans des cercles (**sommets**) et on trace des flèches (**arcs**) de A vers B pour tout module A dépendant de B.

Par exemple, le graphe d'inclusions



signifie que

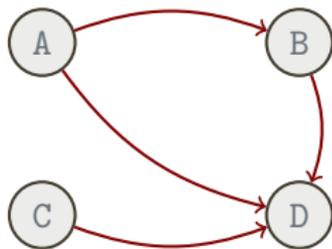
- ▶ A.h inclut B.h et D.h;
- ▶ B.h inclut D.h;
- ▶ C.h inclut D.h;
- ▶ D.h n'inclut rien.

Graphes d'inclusions

On rappelle qu'un module A **dépend** d'un module B si le **fichier d'en-tête** de A inclut B.

Pour visualiser l'allure d'un projet, on trace son **graphe d'inclusions**. On représente pour cela chacun des modules qui le composent dans des cercles (**sommets**) et on trace des flèches (**arcs**) de A vers B pour tout module A dépendant de B.

Par exemple, le graphe d'inclusions



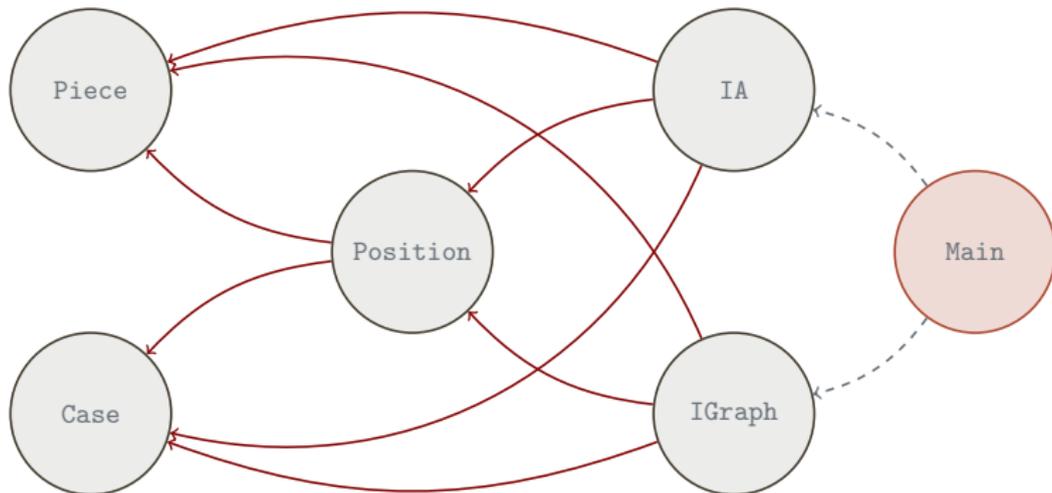
signifie que

- ▶ A.h inclut B.h et D.h;
- ▶ B.h inclut D.h;
- ▶ C.h inclut D.h;
- ▶ D.h n'inclut rien.

On ne mentionne pas dans les graphes d'inclusions les inclusions aux fichiers d'en-tête standards (`stdio.h`, `stdlib.h`, `assert.h`, *etc.*).

Graphe d'inclusions et fichier principal

Le graphe d'inclusions d'un projet consistant à faire jouer l'ordinateur aux échecs contre un humain peut être le suivant :



Tout projet contient un fichier source `Main.c`, le **fichier principal** du projet, où figure la fonction `main` (le **point d'entrée** de l'exécution du programme). Celui-ci apparaît dans le graphe d'inclusions.

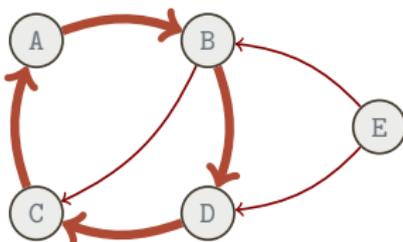
Inclusions circulaires

Les graphes d'inclusions permettent d'avoir une vision globale de l'architecture d'un projet.

Inclusions circulaires

Les graphes d'inclusions permettent d'avoir une vision globale de l'architecture d'un projet.

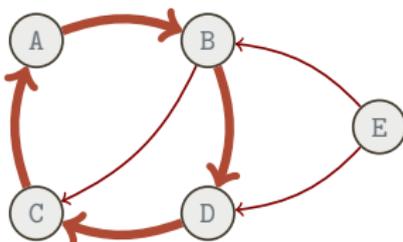
Ils permettent aussi de mettre en évidence des problèmes de conception et notamment les problèmes d'**inclusion circulaire**. Ce type de problème s'observe par la présence d'un **cycle** dans le graphe d'inclusions :



Inclusions circulaires

Les graphes d'inclusions permettent d'avoir une vision globale de l'architecture d'un projet.

Ils permettent aussi de mettre en évidence des problèmes de conception et notamment les problèmes d'**inclusion circulaire**. Ce type de problème s'observe par la présence d'un **cycle** dans le graphe d'inclusions :



Règle importante : il ne doit jamais y avoir de cycle dans le graphe d'inclusions d'un projet. S'il y a un cycle, c'est que le projet est mal découpé en modules.

Limiter les inclusions

La plupart des inclusions circulaires peuvent être évitées en **réduisant au maximum les inclusions** de modules dans les **fichiers d'en-tête** en les faisant plutôt si possible dans les fichiers sources.

Par exemple, supposons que l'on dispose d'un module `Tri` qui permet de trier des tableaux génériques (nous aborderons plus loin ce concept de généricité). Il est de la forme :

```
/* Tri.h */
#ifndef __TRI__
#define __TRI__

void trier_tab(void **t, int n,
               int (*est_inf)(void *, void *));
...
#endif
```

```
/* Tri.c */
#include "Tri.h"
...
void trier_tab(void **t, int n,
               int (*est_inf)(void *, void *)) {
    ...
}
...
```

Limiter les inclusions

On souhaite maintenant écrire un module `TabInt` pour gérer des tableaux d'entiers.

On n'écrira pas

```
/* TabInt.h */
#ifndef __TAB_INT__
#define __TAB_INT__

#include "Tri.h"

    typedef struct {int n; int *tab;} TabInt;
    int trier_tab_int(TabInt *t);
#endif
```

```
/* TabInt.c */
#include "TabInt.h"

int trier_tab_int(TabInt *t)
{
    /* Util. de 'trier_tab' */
}
```

mais plutôt

```
/* TabInt.h */
#ifndef __TAB_INT__
#define __TAB_INT__

    typedef struct {int n; int *tab;} TabInt;
    int trier_tab_int(TabInt *t);
#endif
```

```
/* TabInt.c */
#include "TabInt.h"

#include "Tri.h"

int trier_tab_int(TabInt *t)
{
    /* Util. de 'trier_tab' */
}
```