Plan

2 Habitudes
- Mise en page
- Gestion d'erreurs
- Assertions d'entrée

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (p.ex. lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (p.ex. lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Il est nécessaire de

1. pouvoir détecter les erreurs ;

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une erreur provenant principalement

- du **système** (p.ex. lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (p.ex. lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (p.ex. lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Il est nécessaire de

1. pouvoir détecter les erreurs ;
2. pouvoir remédier aux erreurs quand elles surviennent.

Pour cela, nous allons augmenter l'écriture de fonctions de mécanismes de gestion d'erreur.

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Schématiquement, une telle fonction admet ainsi le prototype

```
int FCT(T1 E1, ..., TN EN, U1 S1, ..., UK SK);
```

dit prototype standard où

- les `Ei` sont les paramètres d'entrée ;
- les `Ti` sont des types (potentiellement des adresses) ;
- les `Sj` sont les paramètres de sortie ;
- les `Uj` sont des types (potentiellement des adresses).

Mécanisme de gestion d'erreurs

Le code d'erreur d'une fonction ayant un prototype standard suit la spécification suivante :

- une **valeur négative ou nulle** renseigne qu'une erreur s'est produite. En général, la fonction renvoie `0` dans ce cas. On peut être plus précis et exprimer une erreur particulière en renvoyant des autres valeurs négatives.
- une **valeur strictement positive signifie** que l'exécution de la fonction s'est bien déroulée. En général, la fonction renvoie `1` dans ce cas. On peut être plus précis et exprimer une information particulière en renvoyant des autres valeurs strictement positives.

Attention : il y a des fonctions de la librairie standard qui ne suivent pas cette spécification.

De notre côté, nous allons la suivre à la lettre dans les fonctions que nous écrirons.

Remarque : le fonctionnement des codes d'erreurs de chaque fonction écrite doit être spécifié dans sa documentation.

Exemple 1 de gestion d'erreur

Considérons la fonction

```
1  int division(float x, float y, float *res) {
2      if (y == 0)
3          return 0;
4      *res = x / y;
5      return 1;
6  }
```

Les entrées sont les flottants `x` et `y`. La sortie est `res` ; c'est une adresse qui pointera sur le résultat de la division de `x` par `y`.

La valeur de retour est un code d'erreur : il vaut `0` lorsque la division ne peut pas être calculée (`y` nul) et vaut `1` sinon.

On remarque que l'on ne modifie pas `*res` lorsque le calcul ne peut pas être réalisé.

Exemple 2 de gestion d'erreur

Considérons la fonction

```
1  int nb_min_maj(char *chaine, int *res_min, int *res_maj)
2      int i;
3      *res_min = 0;
4      *res_maj = 0;
5      i = 0;
6      while (chaine[i] != '\0') {
7          if (('a' <= chaine[i]) && (chaine[i] <= 'z'))
8              *res_min += 1;
9          else if (('A' <= chaine[i]) && (chaine[i] <= 'Z'))
10             *res_maj += 1;
11         else
12             return 0;
13         i += 1;
14     }
15     return i;
16 }
```

L'entrée est la chaîne de caractères `chaine`. Les sorties sont `res_min` et `res_maj` ; ces adresses pointeront sur le nombre de minuscules et de majuscules dans `chaine`.

La valeur de retour est un code d'erreur : il vaut `0` si un caractère non alphabétique apparaît dans `chaine` et vaut la longueur de `chaine` sinon.

Fonctions classiques à gestion d'erreurs

La librairie standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`) ;
- `scanf` renvoie le nombre d'affectations réalisées lors de la lecture. La valeur `EOF` est renvoyée si une erreur de lecture a lieu ;
- `malloc` renvoie un pointeur vers la zone de la mémoire allouée. Lorsque l'allocation échoue, la valeur `NULL` est renvoyée ;
- `fopen` renvoie un pointeur sur le fichier ouvert. Lorsque l'ouverture échoue, la valeur `NULL` est renvoyée ;
- `fclose` renvoie `0` si la fermeture du fichier s'est bien déroulée (attention à ce cas particulier). Lorsque la fermeture échoue, la valeur `EOF` est renvoyée.

Remarque : certaines de ces fonctions ont une gestion d'erreurs encore plus sophistiquée et modifient des variables globales comme `errno` (de l'en-tête `errno.h`) pour renseigner précisément sur l'erreur survenue.

Emploi des fonctions à gestion d'erreurs

Les fonctions à gestion d'erreur renvoient des entiers. De ce fait, leur valeur de retour est une expression booléenne.

Nous pouvons donc combiner l'appel d'une fonction à gestion d'erreurs avec un test pour traiter l'erreur éventuelle.

Voici quelques exemples avec les deux fonctions précédentes :

```
1  if (division(8, a, &b) == 0) {
2      /* Traitement de l'erreur lors
3       * de la division par zero. */
4  }
5  /* Instructions suivantes. */
```

```
1  if (nb_min_maj("UnDeuxTrois", &a, &b) == 0) {
2      /* Traitement de l'erreur lorsque
3       * la chaine de caracteres contient des
4       * caracteres non alphabetiques. */
5  }
6  /* Instructions suivantes. */
```

Interruption de l'exécution

Dans certains cas où une erreur survient, celle-ci peut être irrécupérable. Il faut donc interrompre l'exécution du programme. On utilise pour cela la fonction

```
1     void exit(int status);
```

de `stdlib.h`, appelée avec l'argument `EXIT_FAILURE`.

```
1  /* Allocation dynamique. */
2  tab = (int *) malloc(sizeof(int) * 1024);
3
4  /* Verification de son succes. */
5  if (NULL == tab)
6      /* Sur son echec, on interrompt
7       * l'execution immediatement. */
8      exit(EXIT_FAILURE);
9  /* Instructions suivantes. */
```

Plan

2 Habitudes
- Mise en page
- Gestion d'erreurs
- Assertions d'entrée

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Au lieu de gérer ces cas de figure par l'usage de codes d'erreur, il est possible de tester l'état des arguments.

Une pré-assertion (ou assertion d'entrée) est un test réalisé dans une fonction pour vérifier si elle appelée avec des arguments adéquats.

On utilise la fonction

```
1 void assert(int a);
```

du fichier d'en-tête `assert.h`. Elle fonction de la manière suivante :

- lorsque l'assertion `a` est fausse, l'exécution du programme est interrompue et diverses informations utiles sont affichées ;
- lorsque `a` est vraie, l'exécution continue.

Exemple 1 de fonction avec pré-assertions

Considérons la fonction

```
1  void afficher_tab(int tab[], int nb) {
2      int i;
3      assert(tab != NULL);
4      assert(nb >= 0);
5      for (i = 0 ; i < nb ; ++i)
6          printf("%d\n", tab[i]);
7  }
```

Elle possède deux pré-assertions :

1. la première teste si le tableau `tab` est bien un pointeur valide (différent de `NULL`) ;
2. la seconde teste si la taille `nb` donnée est bien positive.

Conception de pré-assertions

Il est important de munir ses fonctions de pré-assertions les plus précises et complètes possibles. Quelques règles :

- la condition testée ne doit dépendre que des arguments d'une fonction (elle ne dépend pas de données apprises à l'exécution) ;
- la condition testée doit être la plus atomique possible.

```
1  /* Correct. */
2  assert(nb >= 0);
3  assert(nb <= 1024);
```

```
1  /* Incorrect. */
2  assert((0 <=nb) && (nb <=1024));
```

- Pour les concevoir, il faut imaginer les pires cas possibles à capturer qui peuvent survenir (p.ex., pointeurs nuls, quantités négatives, chaînes de caractères vides, *etc.*).
- Elles sont situées juste après les déclarations de variables dans le corps d'une fonction.

Redondance nécessaire des pré-assertions

Considérons les fonctions

```
1  int div(int a, int b) {
2      assert(b != 0);
3      return a / b;
4  }
```

```
1  int somme_div(int a, int b) {
2      return div(a, b)
3          + div(b, a + 1);
4  }
```

Raisonnement : il n'y a pas de pré-assertion dans `somme_div` mais cela n'est pas grave car les cas problématiques sont capturés par `div` qui contient une pré-assertion.

Ceci est une **fausse bonne idée** : chaque fonction doit faire ses propres pré-assertions. Toute erreur doit être capturée le plus en amont possible.

La bonne version de `somme_div` consiste à capturer les mauvaises valeurs possibles de ses arguments de la manière suivante :

```
1  int somme_div(int a, int b) {
2      assert(b != 0);
3      assert(a + 1 != 0);
4      return div(a, b)
5          + div(b, a + 1);
6  }
```

Exemple 2 de fonction avec pré-assertion

La fonction `nb_min_maj`, à code d'erreur, doit être pourvue de pré-assertions :

```
1  int nb_min_maj(char *chaine, int *res_min, int *res_maj)
2      int i;
3
4      assert(chaine != NULL);
5      assert(res_min != NULL);
6      assert(res_maj != NULL);
7
8      /* Suite inchangee. */
9  }
```

On observe que le mécanisme de gestion d'erreurs par valeur de retour teste des comportements incohérents complexes qui se déroulent à l'exécution, tandis que le mécanisme de pré-assertion permet de capturer des erreurs évidentes lisibles directement sur les arguments.

Les pré-assertions pour corriger un programme

Considérons le programme

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int div(int a, int b) {
5      assert(b != 0);
6
7      return a / b;
8  }
9
10 int main() {
11     int a;
12
13     a = div(17, 0);
14     printf("%d\n", a);
15
16     return 0;
17 }
```

La compilation `gcc -ansi -pedantic -Wall Prgm.c` donne l'exécutable `a.out`.

Son exécution `./a.out` est interrompue en l.5. Elle produit la réponse

```
a.out: Prgm.c:5: div: Assertion
‘b != 0’ failed.
Aborted (core dumped)
```

On récolte la précieuse information sur le numéro de ligne de la pré-assertion non satisfaite qui produit l'arrêt précipité de l'exécution.

Plan

3 Modules
- Notion de modularité
- Découpage d'un projet
- Fichiers sources/d'en-tête
- Création de modules
- Graphes d'inclusions
- Erreurs courantes

Plan

3 Modules
- Notion de modularité
- Découpage d'un projet
- Fichiers sources/d'en-tête
- Création de modules
- Graphes d'inclusions
- Erreurs courantes

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.

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

3 Modules
- Notion de modularité
- Découpage d'un projet
- Fichiers sources/d'en-tête
- Création de modules
- Graphes d'inclusions
- Erreurs courantes

Spécification d'un projet

Considérons le projet spécifié de la manière suivante :

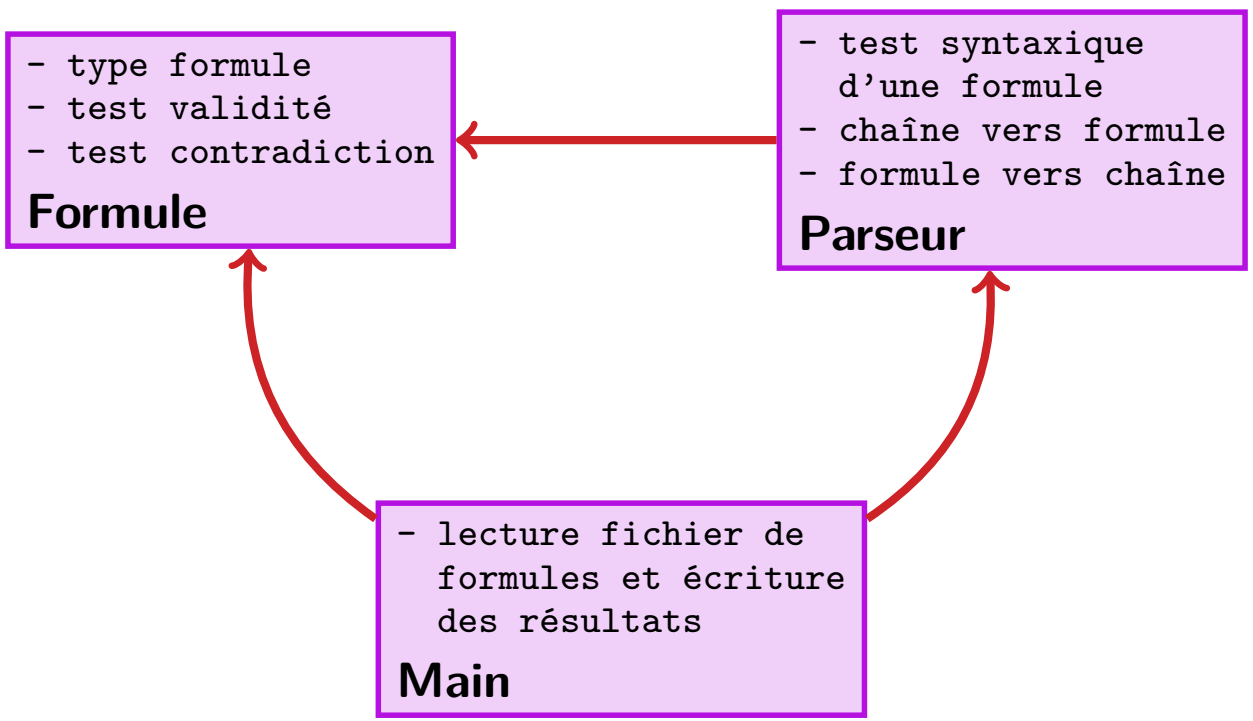
- le but est de fournir un programme qui permet de décider si des formules logiques sans quantificateur sont valides ou contradictoires.
- La syntaxe d'une formule est la suivante : on dispose du jeu de formules atomiques a, b, ..., z et on écrit les formules de manière infixe et totalement parenthésée.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

Ces deux parties dictent le découpage suivant :



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

Plan

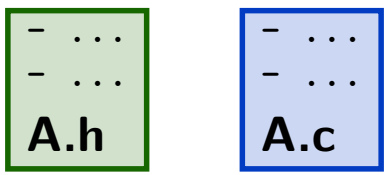
3 Modules
- Notion de modularité
- Découpage d'un projet
- Fichiers sources/d'en-tête
- Création de modules
- Graphes d'inclusions
- Erreurs courantes

Composition d'un module

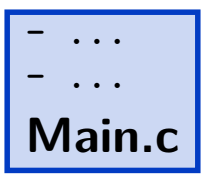
Un module est composé de deux fichiers :

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).

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.

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.

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) {
    ...
}
```

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.

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).

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

```
1  static int nb_occ(Form *f, char atome) {
2      ...
3      assert(f != NULL);
4      assert(('a' <= atome) && (atome <= 'z'));
5      ...
6  }
```

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 source 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 :

1. la 1re 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 2e 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

3 Modules
- Notion de modularité
- Découpage d'un projet
- Fichiers sources/d'en-tête
- Création de modules
- Graphes d'inclusions
- Erreurs courantes

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"
...
```

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.

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

Considérons par exemple le module `Couple` défini par

```c
/* Couple.h */

typedef int Couple[2];

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

```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`.

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

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

#include "Couple.h"
...

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

```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"
...
```

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.