Partie 2 : Types Abstraits de Données (TAD)

Définitions

- Dans un TAD, on s'intéresse à l'ensemble des opérations sur les données et aux propriétés des opérations.
  - sans dire comment ces opérations sont réalisées
  - on sépare le "quoi" (la spécification) du "comment" (l'implémentation)
- Un TAD est
  - un type de données
  - l'ensemble des opérations permettant de gérer ces données

Rq : Représentation des objets ; Un int est représenté par 2 ou 4 octets de mémoire.

Exemples

TAD entiers

- Propriétés
  - nombres
  - positifs / négatifs
- Opérations
  - addition
  - soustraction
  - multiplication
  - division

TAD Ensemble

- Propriétés
  - groupe d'éléments
- Opérations
  - intersection
  - union
  - différence

- Un TAD est caractérisé par
  - sa signature : définit la syntaxe du type et des opérations ;
  - sa sémantique : définit les propriétés des opérations.

Signature d'un TAD

- Comporte
  - Le nom du TAD ;
  - Les noms des types des objets utilisés par le TAD ;
  - Pour chaque opération, l'énoncé des types des objets qu'elle reçoit et qu'elle renvoie.
- Décrite par les paragraphes
  - Type
  - Utilise
  - Opérations

Signature d'un TAD

Exemple TA Booléan

Type Booléen

Utilise $\emptyset$

Opérations

vrai : $\rightarrow$ Booléen

faux : $\rightarrow$ Booléen

non : Booléen $\rightarrow$ Booléen

et : Booléen $\times$ Booléen $\rightarrow$ Booléen

ou : Booléen $\times$ Booléen $\rightarrow$ Booléen

Opérateurs

- Trois catégories d'opérateurs ou de primitives (fonctions en C) :
  - Constructeurs : type spécifié apparaît, uniquement, comme résultat ;
  - Observateurs : type spécifié apparaît, uniquement, comme argument ;
  - Transformateurs : type spécifié apparaît, à la fois, comme argument et comme résultat ;
- Constante : opérateur sans argument

Sémantique d'un TAD

- Précise
  - Les domaines de définition (ou d'application) des opérations ;
  - Les propriétés des opérations.
- Décrite par les paragraphes
  - Préconditions :
    Le champ "Pré-conditions" contient les conditions à respecter sur les arguments d'une fonction pour que celle-ci puisse avoir un comportement normal. On peut parler ici d'ensemble de définition de la fonction.
  - Axiomes :
    Le champ "Axiomes" décrit le comportement de chaque opération d'un type abstrait. Chaque axiome est une proposition logique vraie.

Sémantique d'un TAD

Nom

Nom du TAD

Utilise

Types utilisés par le TAD

Opérations :

nom1 : Type1 $\times$ Type2 $\times$ ... $\rightarrow$ Type'1 $\times$ Type'2$\times$Type'3$\times$ ...

nom2 : Type1$\times$ Type2$\times$ ...$\rightarrow$ Type'1$\times$ Type'2$\times$Type'3$\times$ ...

...

Préconditions :

Précondition1

Précondition2

...

Axiomes :

axiome1 qui décrit logiquement ce que fait une composition d'opérations

axiome2

...

TA Booléen

Type Booléen
Utilise $\emptyset$
Opérations
vrai : $\rightarrow$ Booléen
faux : $\rightarrow$ Booléen
non : Booléen $\rightarrow$ Booléen
et : Booléen $\times$ Booléen $\rightarrow$ Booléen
ou : Booléen $\times$ Booléen $\rightarrow$ Booléen
Préconditions $\emptyset$
Axiomes
Soit, a, b : Booléen
non(vrai) = faux
non(non(a)) = a
vrai et a = a
faux et a = faux
a ou b = non(non(a) et non(b))

TAD Vecteur

Type Vecteur
Utilise Entier, Élément
Opérations
vect : Entier $\rightarrow$ Vecteur
changer_ième : Vecteur $\times$ Entier $\times$ Élément $\rightarrow$ Vecteur
ième : Vecteur $\times$ Entier $\rightarrow$ Élément
taille : Vecteur $\rightarrow$ Entier
Préconditions
vect(i) est_défini_ssi i $\geqslant$ 0
ième(v,i) est_défini_ssi 0 $\leqslant$ i $<$ taille(v)
changer_ième(v,i,e) est_défini_ssi 0 $\leqslant$ i $<$ taille(v)
Axiomes
Soit, i, j : Entier, e : Élément, v : Vecteur
si 0 $\leqslant$ i $<$ taille(v) alors ième(changer_ième(v,i,e),i) = e
si 0 $\leqslant$ i $<$ taille(v) et 0 $\leqslant$ j $<$ taille(v) et i $\neq$ j
Alors
ième(changer_ième(v,i,e),j) = ième(v,j)
taille(vect(i)) = i
taille(changer_ième(v,i,e)) = taille(v)

Opérations Partielles

- Une opération peut ne pas être définie partout
- Cela dépend de son domaine de définition
- Ceci est traité dans le paragraphe Préconditions
- Exemple
  - Opérations ième et changer_ième du TAD Vecteur

Réutilisation des TADs

- Quand on définit un type, on peut réutiliser des types déjà définis
- La signature du type défini est l'union des signatures des types utilisés enrichie des nouvelles opérations
- Le type hérite des propriétés des types qui le constituent
- Exemples
  - Types Entier et Élément utilisés par le TAD Vecteur

Choix des Axiomes

- Deux questions existentielles :
  - y a-t-il des axiomes contradictoires ? $\rightarrow$ consistance
  - y a-t-il suffisamment d'axiomes ? $\rightarrow$ complétude
- Les axiomes doivent être consistants et complets

Type Abstrait de données : Récapitulation

- Un TAD spécifie
  - Le type de données contenues
  - Une description détaillée des opérations qui peuvent être effectuées sur les données
- Un TAD ne spécifie pas
  - La façon dont les données sont stockées
  - Comment les méthodes sont implémentées
- $\Longrightarrow$ Structures de données

Notion de Structure de Données

- On dit aussi structure de données concrète
- Correspond à l'implémentation d'un TAD
- Composée d'un algorithme pour chaque opération, plus éventuellement des données spécifiques à la structure pour sa gestion
- Un même TAD peut donner lieu à plusieurs structures de données, avec des performances différentes

Implémentation d'un TAD

- Pour implémenter un TAD
  - Déclarer la structure de données retenue pour représenter le TAD : L'interface
  - Définir les opérations primitives dans un langage particulier : La réalisation
- Exigences
  - Conforme à la spécification du TAD ;
  - Efficace en terme de complexité d'algorithme.
- Pour implémenter en C, on utilise
  - La programmation modulaire
  - Les types élémentaires (entiers, caractères, ...)
  - Les pointeurs ;
  - Les tableaux et les enregistrements ;
- Plusieurs implémentations possibles pour un même TAD

Programmation modulaire

- Un module est un ensemble de fonctions mis dans des fichiers à part : un programme sera donc un ensemble de fichiers qui seront réunis lors des différentes étapes de compilation et d'exécution.
- En C un module est défini par deux fichiers :
  - fichier truc.h : l'interface (la partie publique) ; contient la spécification de la structure ;
  - fichier truc.c : la définition (la partie privée) ; contient la réalisation des opérations fournies par la structure. Il contient au début l'inclusion du fichier truc.h
- Un module C implémente un TAD
  - L'encapsulation : détails d'implémentation cachés ; l'interface est la partie visible à un utilisateur
  - La réutilisation : placer les deux fichiers du module dans le répertoire où l'on développe l'application.

Programmation modulaire : Headers

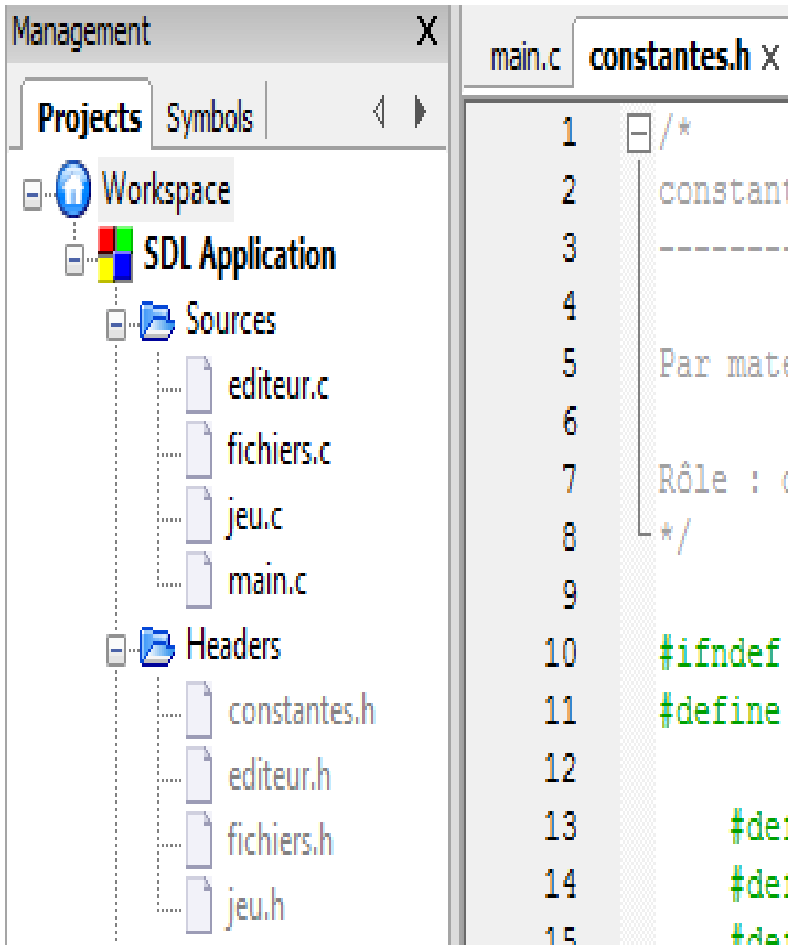
- Fichiers .h et .c
  - En général, on met rarement les prototypes dans les fichiers .c (sauf pour un programme tout petit)
  - Les .c : les fichiers source. Ces fichiers contiennent les fonctions elles-mêmes.
  - Les .h, appelés fichiers headers. Ces fichiers contiennent les prototypes des fonctions.
- Le nombre et la gestion des fichiers sources dépendent de la conception du développeur
- En général, on regroupe dans un même fichier des fonctions ayant le même thème.

Programmation modulaire : Headers

- Pour chaque fichier .c, il y a son équivalent .h qui contient les prototypes des fonctions
- Pour que l'ordinateur sache que les prototypes sont dans un autre fichier que le .c, il faut inclure le fichier .h
- L'inclusion des headers se fait grâce à une directive de préprocesseur (#include)
  - les chevrons < > pour inclure un fichier se trouvant dans le répertoire « include » de votre IDE
  - les guillemets " " pour inclure un fichier se trouvant dans le répertoire de votre projet (à côté des .c, généralement).
  - Exemples : #include<stdio.h>, #include"jeu.h" ...

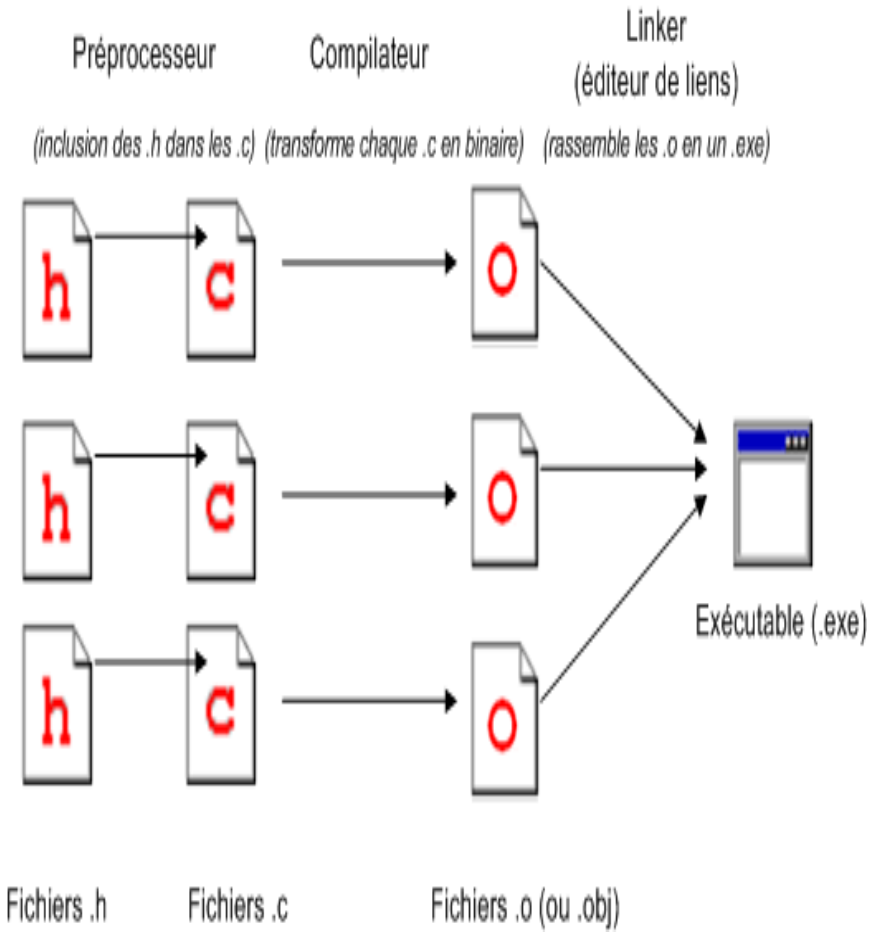
Programmation modulaire : projet

- Exemple



La compilation séparée

- Quand un projet comporte plus qu'un fichier, les fichiers seront compilés séparément. On parle donc de compilation séparée :

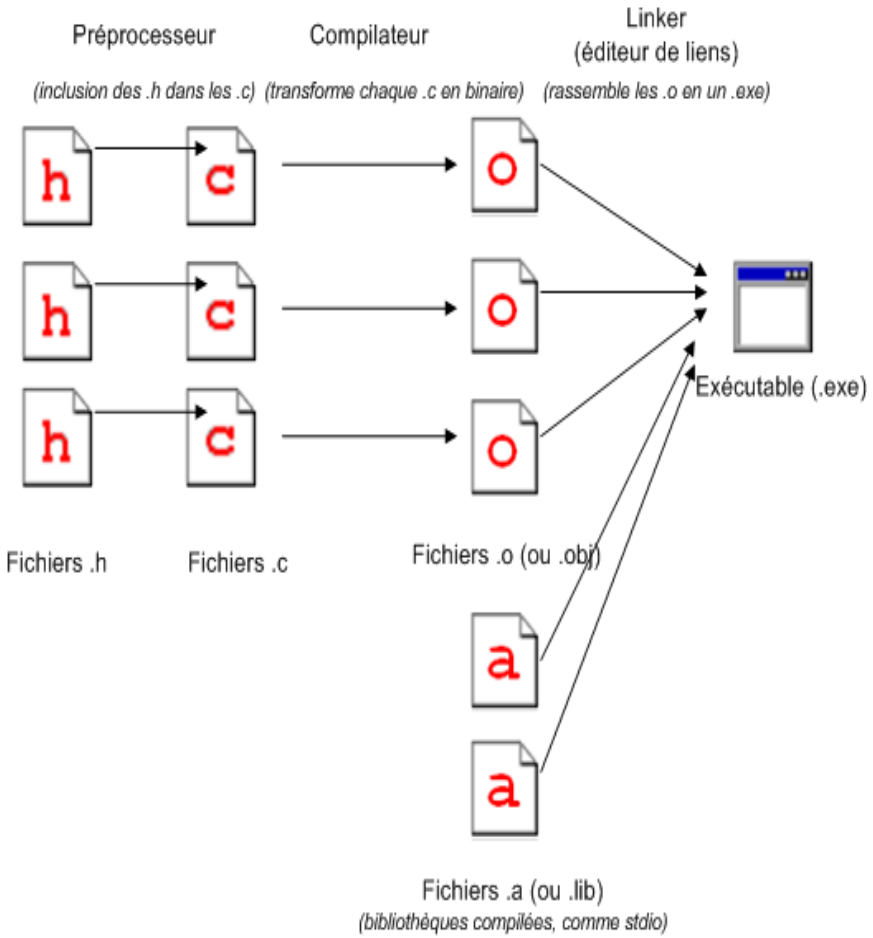


La compilation séparée

- Lors de la compilation séparée trois programmes interviennent :
  - Préprocesseur : le préprocesseur est un programme qui démarre avant la compilation. Son rôle est d'exécuter les instructions spéciales qu'on lui a données dans des directives de préprocesseur, ces lignes qui commencent par un #.
  - Compilation : cette étape très importante consiste à transformer vos fichiers source en code binaire compréhensible par l'ordinateur. Le compilateur compile chaque fichier .c un à un. Le compilateur génère un fichier .o (ou .obj, ça dépend du compilateur) par fichier .c compilé. Ce sont des fichiers binaires temporaires.
  - Édition de liens : le linker (ou « éditeur de liens » en français) est un programme dont le rôle est d'assembler les fichiers binaires .o. Il les assemble en un seul gros fichier : l'exécutable final. Cet exécutable a l'extension .exe sous Windows. Si vous êtes sous un autre OS, il devrait prendre l'extension adéquate.

La compilation séparée

- Le linker qui va devoir assembler les .o de votre programme avec les bibliothèques compilées dont vous avez besoin (.a ou .lib selon le compilateur) :



```
#ifdef symbole
partie-du-programme-1
#else condition-2
partie-du-programme-2
#endif
```

- La directive #else est évidemment facultative.
- De façon similaire, on peut tester la non-existence d'un symbole par :

```
#ifndef symbole
partie-du-programme-1
#else condition-2
partie-du-programme-2
#endif
```

Exemple

```
#define WINDOWS // pour windows, sinon #defind LINUX pour linux...

#ifdef WINDOWS
    /* Code source pour Windows */
#endif

#ifdef LINUX
    /* Code source pour Linux */
#endif

#ifdef MAC
    /* Code source pour Mac */
#endif
```

La compilation conditionnelle : les inclusions infinies

- #ifndef pour éviter les inclusions infinies
- cette directive est très utilisé dans les .h pour éviter les « inclusions infinies ».
- supposons que nous avons un fichier A.h et un fichier B.h. dans le fichier A.h nous avons un #include"B.h" et dans le fichier B.h nous avons un #include"A.h". On aura la situation suivante :
  - le préprocesseur lit A.h et voit qu'il faut inclure B.h ;
  - il lit B.h pour l'inclure, et là il voit qu'il faut inclure A.h ;
  - il inclut donc A.h dans B.h, mais dans A.h on lui indique qu'il doit inclure B.h !

  -> inclusion infinies
- Pour éviter qu'un fichier .h ne soit inclus un nombre infini de fois, on le protège à l'aide d'une combinaison de constantes de préprocesseur et de conditions

La compilation conditionnelle : les inclusions infinies

```
#ifndef DEF_NOMDUFICHIER // Si la constante n'a pas été définie le
                         // fichier n'a jamais été inclus}
#define DEF_NOMDUFICHIER // On définit la constante pour que la prochai
                         // fois le fichier ne soit plus inclus
/* Contenu de votre fichier .h (autres include, prototypes, define...)
#endif
```

- première inclusion
  - Le préprocesseur lit la condition "Si la constante DEF_NOMDUFICHIER n'a pas été définie"
  - Comme c'est la première fois que le fichier est lu, la constante n'est pas définie
  - le préprocesseur entre donc à l'intérieur du if
- deuxième inclusion
  - la constante DEF_NOMDUFICHIER est définie
  - le préprocesseur n'entre pas à l'intérieur du if
  - le fichier ne risque plus d'être inclus à nouveau

Implémentation du TA Booléen en C

/* fichier Booleen.h */

```
#ifndef _BOOLEEN
#define _BOOLEEN
typedef enum (faux,vrai) Booleen;
Booleen et(Booleen x, Booleen y);
Booleen ou(Booleen x, Booleen y);
Booleen non(Booleen x) ;
#endif
```

/* fichier Booleen.c */

```
#include "Booleen.h"
Booleen et(Booleen x, Booleen y) {
 if (x == faux)
   return faux;
 else return y;
}
Booleen ou(Booleen x, Booleen y) {
 if (x == vrai)
   return vrai;
 else return y;
}
Booleen non(Booleen x) {
 if (x == vrai)
   return faux;
 else return vrai;
}
```