Université de Liège

INFO0947

Compléments de Programmation

Un exercice dont vous êtes le Héros · l'Héroïne.

TAD – Nombres Complexes

Simon Liénardy Benoit Donnet

5 avril 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le héros », nous vous proposons de suivre pas à pas la résolution de deux exercices. L'un sur un problème concernant la définition d'un TAD représentant les Nombres Complexes (c'est un exercice complet, de la spécification abstraite du TAD à l'implémentation concrète) et l'autre sur la spécification abstraite d'un Ensemble d'Entiers.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

5.1 Énoncé

On désire définir un type abstrait permettant de manipuler un *nombre complexe*. Ce type abstrait sera appelé `Complex`.

Le type défini devra comporter des opérations pour additionner, soustraire, multiplier, et diviser deux nombres complexes, pour calculer le module et le conjugué d'un nombre complexe, pour indiquer la partie réelle et la partie imaginaire d'un nombre complexe ainsi qu'un constructeur donnant, à partir de deux réels a et b, le nombre complexe $a + bi$.

▷ **Exercice** Spécifiez complètement le type abstrait `Complex`.

▷ **Exercice** Proposez une structure de données permettant de représenter le type abstrait `Complexe`.

▷ **Exercice** Écrivez le *header* C pour le type abstrait `Complex`. N'oubliez pas de spécifier, formellement, chaque fonction/procédure.

▷ **Exercice** Écrivez le module C implémentant le header défini à l'exercice 3.

▷ **Exercice** Écrivez un programme utilisant le type abstrait `Complex`.

5.1.1 Méthode de résolution

Nous allons procéder de la façon suivante :

1. Spécification abstraite (Sec. 5.2) ;
2. Structure de données (Sec. 5.3) ;
3. Interface pour le type abstrait (Sec. 5.4) ;
4. Implémentation du module pour le type abstrait (Sec. 5.5) ;
5. Implémentation d'un programme utilisant le type abstrait (Sec. 5.6)

5.2 Spécification Abstraite

Il faut fournir la spécification abstraite de la structure de données.

Pour un rappel sur la spécification abstraite, voyez la section 5.2.1
La correction de la spécification est disponible à la section 5.2.3

Conseil

Si vous ne vous souvenez plus des propriétés fondamentales des nombres complexes, allez relire votre cours de mathématique (cfr. MATH2007, Chapitre 1 (Sec. 1.8, pg. 46 → 55)).

5.2.1 Rappel sur les spécifications abstraites

Voici un rappel à propos de ce qui est attendu :

5.2.1.1 Structure de la spécification d'un TAD

Un *Type Abstrait de Données* (TAD), c'est la description d'un ensemble de valeurs ainsi que les opérations permises sur ces valeurs. Passons maintenant aux exercices ! Ou peut-être pas... Précisons qu'on s'intéresse aux propriétés des opérations et non à la manière dont elles sont implémentées. On sépare le *quoi* (spécifications) du *comment* (implémentation).

La spécification d'un TAD contient de nombreux éléments qui sont rappelés ici :

Type C'est le nom du TAD. On utilise le plus souvent un nom anglais qui débute par une lettre capitale. On privilégie un nom en rapport avec ce que l'on est en train de décrire, évidemment. Attention : Un nom ne confère pas *magiquement* des propriétés au type défini : c'est la sémantique qui se charge de décrire les propriétés du TAD ! Par contre, trouver un nom adéquat permet davantage de **lisibilité**.

Utilise On liste ici tous les autres TAD nécessaires à la spécification de celui qui est en train d'être décrit. C'est important : on pourra alors *utiliser* les opérations des TAD mentionnés dans la liste lors de l'écriture des axiomes. Certains types sont considérés comme **bien connus** et peuvent être utilisés *ad libitum* [a]. C'est le cas par exemple de Natural, Integer et Boolean. La généricité d'un TAD s'exprime avec l'utilisation du type Element.

Opérations On liste ici les opérations du TAD, en utilisant une *notation fonctionelle* qui indique le nom de chaque opération ainsi que les objets qu'elle reçoit et qu'elle renvoie. Selon le type de ces objets, on peut classer les opérations en trois catégories : les **constructeurs**, les **observateurs** et les **transformateurs**. Pour savoir exactement comment procéder pour décrire une opération, reportez-vous à la section 5.2.1.2.

Préconditions Parfois, certaines opérations n'ont aucun sens si elles sont appliquées à certaines valeurs particulières du TAD. Dans ce cas, on peut restreindre le domaine de définition de ces opérations à l'aide de préconditions. On obtient alors ce que l'on appelle des **opérations partielles**, parce qu'elles ne sont que partiellement définies sur le domaine de définition.

Axiomes C'est la partie la plus importante du TAD, le plat de résistance ! Grâce aux axiomes, on peut enfin comprendre la signification des différentes opérations. Il faut surtout faire attention à avoir assez d'Axiomes (on parle de *complétude*) et que ceux-ci n'entrent pas en contradiction (on parle de *consistance*). Comment être complet et consistant ? Le sujet est abordé à la section 5.2.1.3.

→ **Type**, **Utilise** et **Operation** forment ce que l'on appelle la **signature** du TAD (ou syntaxe).

→ Les **Préconditions** et les **Axiomes** forment la **sémantique** du TAD.

a. « À volonté », en latin.

Message à caractère informatif

Alerte : Rupture d'Abstraction

Aviez-vous remarqué que le « A » de TAD signifie « Abstrait » ? Vous pensez que je vous prends pour un jambon à mentionner cela dans un cadre mis en évidence. Et pourtant...

Il faut faire très attention à **ne pas** mentionner d'informations au sujet d'une quelconque implémentation du TAD (cela arrive *hélas* chaque année !). Cela constitue une *Rupture d'Abstraction* qui est éliminatoire tant à l'écrit qu'à l'oral !

5.2.1.2 Comment décrire une opération

Voici comment se présente la description d'une opération. Il faut suivre ce schéma :

nom de l'Opération : type $\text{arg}_1 \times$ type $\text{arg}_2 \times \cdots \times$ type $\text{arg}_n \rightarrow$ type

On commence par le nom de l'opération, suivi de deux points. Les type arg_i représentent les types des arguments de l'opération. On ajoute ensuite une flèche et on indique alors le type de l'objet renvoyé par l'opération. Par exemple, pour le TAD Vector, on avait l'opération *set* :

set : Vector $\times$ Integer $\times$ Element $\rightarrow$ Vector

L'opération *set* prend en argument un $Vector$ (qui sera modifié), un entier ($Integer$ – qui représente une position dans le $Vector$), ainsi qu'un élément ($Element$ – qui sera inséré par set à l'endroit spécifié par l'entier). Les précisions entre parenthèses dans la phrase précédente ne sont disponibles qu'une fois les axiomes correctement spécifié : c'est là leur but !

Typologie des opérations

On peut regrouper les opérations en plusieurs catégories :

Les Constructeurs Ce sont les opérations pour lesquelles le TAD n'apparait que du côté des résultats. Comme le nom l'indique, ce sont des opérations qui permettent de créer une valeur du type du TAD.

Les Observateurs Ce sont des opérations pour lesquelles le TAD n'apparait que dans la liste des arguments. Un autre type de valeur est renvoyé. C'est une valeur observée.

Les Transformateurs Pour ces opérations, le TAD apparait tant du côté des arguments que du côté du résultat. La (ou les) valeurs·s de type TAD passée·s en arguments sont transformée·s en la valeur renvoyée en résultat.

Les Constantes Ce sont des opérations sans arguments. Incidemment, ce sont donc des cas particuliers de constructeurs.

On regroupe parfois les constructeurs et les transformateurs en **opérations internes** que l'on distingue donc des observateurs. Cette classification est intéressante lorsque l'on vérifie que les axiomes correspondants aux opérations sont consistants et complets.

Conseils pour procéder

Il faut d'abord choisir un nom qui représente correctement l'opération que l'on souhaite décrire. Ensuite, il est sans doute plus facile de réfléchir à la catégorie de l'opération : est-ce un constructeur ? Un observateur ? Un transformateur ? Cela donne déjà quelques indices sur les types des paramètres et des résultats. Ensuite, il faut se demander : de quoi ai-je besoin pour réaliser l'opération ? Il suffit alors de lister les paramètres et de les séparer avec le caractère $\times$ (`\times` en latex). Il faut ensuite bien vérifier que chaque type d'argument, s'il n'est pas le TAD lui-même, est bien listé dans la section **Utilise** de la signature.

5.2.1.3 Comment écrire les Axiomes, être complet et consistant

Écrire des axiomes demande de l'abstraction (en même temps, la signification de TAD devrait vous avoir mis la puce à l'oreille [2]). Voici une série de conseils à mettre en pratique [3] :

1. Réfléchir à la question : mon opération est-elle pertinente pour toutes les valeurs possibles et imaginables de ses arguments ? Si ce n'est pas le cas, il faut écrire des **Préconditions**. Il y a deux manières de les écrire.

   Soit :

   $$\forall i, Cond(i), nomOperation(\text{arg}_1, \text{arg}_2, \ldots, i, \ldots, \text{arg}_\text{n})$$

   On souhaite limiter l'opération pour l'un de ses arguments, on écrit une quantification universelle (cfr. Chapitre 1, Slides 12 → 16) introduisant une variable liée (cfr. Chapitre 1, Slides 22 → 24) qui sera du même type que l'argument à limiter. $Cond(i)$ est une condition que doit satisfaire toutes les valeurs de l'argument à limiter pour que l'opération soit permise. Exemple :

   $$\forall \text{i}, 0 \leq \text{i} < \text{size(v)}, \text{get(i, v)}$$

   signifie que l'opération *get* n'est permise que si l'entier i est compris entre 0 et $size(v) - 1$. Savoir à quoi exactement servent i et v n'est possible qu'en lisant la suite des axiomes.

   Ou Bien :

   $$nomOperation(\text{arg}_1, \text{arg}_2, \ldots, i, \ldots, \text{arg}_\text{n}) \text{ is defined iff } Cond(i)$$

   Les plus sagaces auront remarqué que cela consiste juste à déplacer la condition $Cond(i)$. La signification est (forcément) la même que ci-dessus. Exemple identique au précédent :

   $$\text{get(i, v) is defined iff } 0 \leq \text{i} < \text{size}$$

   Quelle variante choisir ? J'ai une préférence pour la 2e formulation qui a l'heur [4] d'être explicite, qualité rare quand on parle de TAD.

   Note : La première ligne des préconditions sera constituées de l'information que tous les autres arguments $\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_\text{n}$ peuvent prendre n'importe quelle valeur. Cela se note ainsi :

   $$\forall \text{arg}_1 \in \text{Type arg}_1, \forall \text{arg}_2 \in \text{Type arg}_2, \ldots, \forall \text{arg}_\text{n} \in \text{Type arg}_\text{n}$$

2. Passer aux axiomes proprement-dits. Tous les axiomes présentent la forme suivante :

   Terme de gauche = Terme de droite

   Commençons par regarder le **terme de gauche**. Celui-ci est le plus souvent constitué par la combinaison d'une **opération interne** avec un **observateur**. Ce n'est pas tout le temps le cas (voir plus bas) mais la plupart du temps, on commence par cette combinaison. On sait déjà qu'un observateur prend en argument une valeur de type TAD alors qu'une opération interne produit une valeur de type TAD. Donc, logiquement, c'est l'observateur qui va prendre en argument le résultat de l'opération interne. Attention à ne pas écrire des axiomes trop spécifiques : les arguments de l'opération interne peuvent prendre **n'importe quelle valeur permise par les préconditions**. Et on se limite à cela pour le terme de gauche ! En conséquence, toute la difficulté est d'écrire le terme de droite. Exemple, dans :

2. Déso si c'est pas le cas...
3. Les exemples sur lesquels nous nous appuyons dans la suite sont tirés du cours théorique. Voir Chapitre 5, Slides 23 → 32
4. Nope, y'a pas d'erreur.

size(set(v, i, e)) = ...

On a bien mis un v, un i et un e quelconque. On n'a pas mis, que sais-je ? 0 à la place du i : ce serait trop spécifique. D'ailleurs, on rappelle en début des axiomes que ces arguments peuvent prendre n'importe quelle valeur permise :

$$\forall \text{arg}_1 \in \text{Type arg}_1, \forall \text{arg}_2 \in \text{Type arg}_2, \ldots, \forall \text{arg}_\text{n} \in \text{Type arg}_\text{n}$$

en l'occurrence :

$$\forall v \in vector, \forall i \in Integer, \forall e \in Element$$

3. Décrire le **terme de droite** des axiomes à l'aide de la **récursion**. Il faut maintenant décrire le résultat de l'opération décrite dans le terme de gauche de manière récursive. Comme d'habitude, il y a un ou plusieurs cas de base ainsi que un ou plusieurs cas récursifs. Si l'opération doit avoir un comportement déterminé pour une valeur particulière d'un de ses argument, c'est dans la définition du terme de droite que l'on discute ce cas précis.

   Reprenons l'exemple du TAD Vector :

   size(set(v, i, e)) = ...

   Il faut exprimer la taille d'un Vector v après modification de l'élément e à la position i. Dans notre esprit, il est clair que v est le $Vector$, que i est un index, que e est une élément de remplacement et que *set* est une opération de transformation. Il faut bien se rappeler qu'un lecteur quelconque ne comprendra tout cela qu'après avoir lu le reste des axiomes ! Puisque l'axiome a toute les chances d'être un cas récursif, on peut déjà recopier l'appel à l'observateur *size*. Par contre, on essaie de simplifier l'argument de cet observateur : c'est l'idée de la récursion, de converger vers un cas de base. Le plus souvent, on essaie de faire disparaitre l'occurrence de l'opération interne présente dans le terme de gauche (ici, c'est *set*). Notez que ce n'est pas toujours possible. En ce qui concerne cet axiome particulier, on veut exprimer que modifier un élément à une position donnée *ne modifie pas* la taille du $Vector$. On aura donc l'axiome :

   size(set(v, i, e)) = size(v)

   On voit bien que $size(v)$ est plus simple que $size(set(v, i, e))$. L'axiome exprime en outre que l'opération est indépendante des valeurs de i et de e, puisqu'ils ne sont pas présents dans le terme de droite.

4. Vérifier que l'on est bien **consistant** et **complet**

   Le cours mentionne deux méthodes :

   — S'assurer que le comportement du TAD est bien décrit. C'est une méthode intuitive. Il faut avoir l'habitude. *Passez votre chemin en première lecture.*

   — Écrire les axiomes qui définissent le résultat de la composition de tous les observateurs avec toutes les opérations internes. C'est la méthode recommandée. Il est recommandé de travailler avec un tableau à double entrée [5] : les lignes (resp. les colonnes) listent les observateurs (resp. les opérations internes). Dès que **tous les cas** [6] de la combinaison d'un observateur et d'une opération interne sont décrits, vous cochez la case correspondante. Il faut avoir coché toutes les cases du tableau. Simple.

   Voici une variante de la deuxième méthode. Elle se base sur une autre classification des opérations. Elle distingue :

   — Les constructeurs universels : c'est un ensemble minimum d'opérations qui permettent de créer toutes les valeurs possibles de TAD. Exemple pour le TAD *List* (cfr. Chapitre 6, Slides 13 → 17) : *empty_list* et *add_at*. N'importe quelle liste peut être obtenue par la combinaison de ces deux opérations ;

5. C'est d'ailleurs comme cela que les TADs des projets sont corrigés
6. Ne vous contentez pas des cas de base !

— Les autres opérations.

Il faut combiner chacune des autres opérations avec chacun des constructeurs universels. Dans l'exemple des listes, on aurait par exemple plusieurs axiomes qui combinent *remove* et *add_at*. Il se présenteraient comme ceci :

$$\begin{array}{lll} \text{remove}(\text{add\_at}(L, i, e), j) & = L & \text{si } i = j \\ & = \text{add\_at}(\text{remove}(L, j-1), i, e) & \text{si } i < j \\ & = \text{add\_at}(\text{remove}(L, j), i-1, e) & \text{si } i > j \end{array}$$

Attention, dans le terme de gauche, il y a bien deux positions. On utilise alors deux variables distinctes : i et j. Les relations possibles entre i et j sont discutées dans trois axiomes séparés. On ne se contente pas du cas où $(i = j)$ qui est le plus facile à écrire. La première ligne signifie que supprimer l'élément que l'on vient d'ajouter revient à ne rien faire du tout. Les deux autres sont des cas récursifs. Les indices i et j sont modifiés dans l'appel récursif puisque dans le terme de gauche, on retire après un ajout alors qu'on ajoute après un retrait dans le terme de droite. Il faut donc gérer le décalage des positions [7]. Vu les conditions des deux derniers cas et la mise à jour des indices, on voit qu'on converge bien vers le cas de base $i = j$. En quoi est-ce que le cas récursif est « plus simple » ? Dans la mesure où, dans le terme de gauche, *remove* est appliquée au résultat de *add_at* et dans le terme de droite, elle n'est plus appliquée que sur la liste L, on peut dire que c'est plus simple.

5. Réduire le nombre d'axiomes. On peut parfois parvenir à réduire le nombre d'axiome. Par exemple, si une opération C peut s'exprimer en fonction de deux autres opérations A et B (se sont soit tous des observateurs, soit tous des transformateurs), on peut avoir un axiome du type :

$$C(args) = f(A(args), B(args))$$

Où f est la fonction qui combine le résultat de A et B. Dans le tableau à double entrée présenté précédemment, la ligne (resp. la colonne) de C peut être cochée dès que les lignes (resp. colonnes) de A et de B sont remplies. On peut donc réduire le nombre d'axiomes. Exemple pour le TAD *List* :

add_first(L, e) = add_at(L, 0, e)
add_last(L, e) = add_at(L, length(L), e)

Le TAD *List* possède 4 observateurs. Cex deux axiomes permettent donc de ne pas en écrire 6 (4 *obs* × 2 *op* − 2 *axiomes*).

Alerte : Allergie à la récursivité

Une erreur courante est de ne pas assez être récursif dans l'écriture des axiomes. il faut se dire que la récursivité, c'est plutôt **la norme**. Il faut s'inquiéter si les principales opérations du TAD ne sont pas axiomatisées récursivement !

Un cas particulier consiste à sur-spécifier le terme de gauche de l'axiome, en se focalisant sur des valeurs particulières des arguments de l'opération interne. L'axiome ne documente pas bien l'opération car il se focalise sur un sous-ensemble de valeurs possible. Le risque d'être incomplet est alors très grand !

5.2.2 Suite de l'exercice

Spécifiez le TAD. rendez-vous à la Sec. 5.2.3 pour la correction !

7. Un petit schéma a été réalisé pour ne pas nous tromper...

5.2.3 Spécification

Puisque la spécification d'un TAD est constituée de deux parties, procédons en deux étapes :

5.2.3.1 Syntaxe

```
Type:
  Complex
Utilise:
  Real
Opérations:
  add: Complex × Complex → Complex
  sub: Complex × Complex → Complex
  mul: Complex × Complex → Complex
  div: Complex× Complex → Complex
  modulus: Complex → Real
  conj: Complex → Complex
  real: Complex → Real
  imag: Complex → Real
  complex: Real × Real → Complex
```

On choisit d'appeler notre type `Complex`.
Pour les opérations, on reprend l'énoncé et on liste les opérations dans l'ordre :

> Le type défini devra comporter des opérations pour additionner, soustraire, multiplier, et diviser deux nombres complexes, pour calculer le module et le conjugué d'un nombre complexe, pour indiquer la partie réelle et la partie imaginaire d'un nombre complexe ainsi qu'un constructeur donnant, à partir de deux réels a et b, le nombre complexe $a + bi$.

L'addition, la soustraction, la multiplication, la division sont des lois de compositions internes : ce sont des transformateurs car ces opérations prennent deux complexes en arguments pour produire un nouveau complexe. Le complexe conjugué est aussi un transformateur mais n'a qu'un seul argument. La prise de partie réelle, imaginaire et le calcul du module sont évidemment des observateurs qui retournent des valeurs réelles. Le TAD `Complex` utilise donc le type `Real`.

Suite de l'exercice

On a donc bien 6 opérations internes et 3 observateurs. Prévoyez donc, avant de passer à la sémantique, votre tableau à double entrée pour ne pas oublier d'axiomes :

		opérations internes					
		complex	add	sub	mul	div	conj
observateurs	real						
	imag						
	modulus						

Passez donc à la section suivante (5.2.3.2).

5.2.3.2 Semantique

Préconditions

Préconditions:
$\forall\ x, y\ \in\ Complex$
div(x,y) is defined iff $y\ \neq$ create(0,0)

Axiomes

On va suivre la méthode recommandée qui consiste à combiner les observateurs avec les opérations internes. On devrait donc avoir :

3 Observateurs $\times$ 6 Opérations Internes = 18 Axiomes

On devrait avoir, au minimum, 18 axiomes. C'est relativement beaucoup. On peut essayer de voir si un observateur (resp. un transformateur) peut s'exprimer comme la combinaison d'autres observateurs (resp. transformateurs). C'est le cas du module qui peut s'exprimer comme la longueur de l'hypoténuse du triangle rectangle dont les autres côtés sont les parties réelle et imaginaire. On aura donc moins d'axiomes :

2 Observateurs Restants $\times$ 6 Opérations Internes + 1 Axiome = 13 Axiomes

On réduit donc de 5 axiomes, le gain est appréciable !

Axiomes:
$\forall\ a, b\ \in\ Real,\ \forall\ c, d\ \in\ Complex$

// Combinaison d'observateurs :
modulus(c) = $\sqrt{\texttt{real(c)}^2\ +\ \texttt{imag(c)}^2}$

// Obervateur « real» :
real(complex(a,b)) = a
real(add(c,d)) = real(c) + real(d)
real(sub(c,d)) = real(c) - real(d)
real(mul(c,d)) = real(c) × real(d) - imag(c) × imag(d)
real(div(c,d)) = (real(c)×real(d) + imag(c)×imag(d))/(real(d)^2 + imag(d)^2)
real(conj(c)) = real(c)

// Obervateur « imag »
imag(complex(a,b)) = b
imag(add(c,d)) = imag(c) + imag(d)
imag(sub(c,d)) = imag(c) - imag(d)
imag(mul(c,d)) = real(c) × imag(d) + imag(c) × real(d)
imag(div(c,d)) = (imag(c)×real(d) - real(c)×imag(d))/(real(d)^2 + imag(d)^2)
imag(conj(c)) = −imag(c)

On peut utiliser les symbole $\sqrt{\cdot}$ et $\square^2$ parce que `Complex` utilise le type `Real`.

La majorité des axiomes est bien récursive : les observateurs sont à la fois dans les termes de gauche et dans les termes de droite des axiomes. Dans les « appels récursifs », on converge bien vers le cas de base : si on développe tous les axiomes, pour toute combinaison d'opérations, on arrivera à des applications des observateurs sur le constructeur *complex*.

Si vous avez quelque chose de sensiblement différent Partagez votre solution sur eCampus !

Suite de l'exercice

Proposez maintenant une structure de données pour représenter le type abstrait *Complex*.

Une solution et son commentaire est disponible au point 5.3.

5.3 Structure de Données

```
typedef struct complex_t{
   double real;
   double imag;
} Complex;
```

Structure classique pour un nombre complexe : on retient la partie réelle et la partie imaginaire chacune dans un double.

Remarque 1 : Il n'est pas obligatoire de stocker la partie réelle et imaginaire. En effet, on peut retenir le couple argument/module. Cela simplifie certaines opérations (`modulus` devient triviale, la multiplication devient plus simple, ...).

Remarque 2 : le standard C99 introduit le mot réservé « `_Complex` ». Les fonctions de manipulation des nombres complexes sont rassemblées dans l'en-tête `complex.h`. Le `_Complex` s'emploie comme un type primitif et a donc tous ses avantages (pas besoin d'un type opaque en l'occurrence).

Suite de l'exercice

Passons maintenant à l'implémentation, en particulier l'interface. Une proposition de solution est donnée à la Sec. 5.4.

5.4 Implémentation de l'Interface

Il faut maintenant définir l'interface (*header*) pour la structure de données.

Si vous ne vous souvenez plus de ce qu'il faut faire, voyez la section 5.4.1
Le corrigé est disponible à la section 5.4.2

5.4.1 Rappels sur l'Implémentation

L'implémentation du TAD correspond à la structure de données concrète. On va considérer la programmation modulaire, i.e., la division du code en header et module.

5.4.1.1 Le Header

Le header contient l'interface du TAD et se présente dans un fichier dont, typiquement, le nom est `tad.h` (où `tad` est le nom du TAD). En particulier, le header est structuré comme suit :

Include guards Afin d'éviter les problèmes d'inclusions multiples, on veillera à emballer le header avec les *include guards* (cfr. INFO0030, Partie 1, Chapitre 1, Slides 20 → 34). Par convention, on procédera de la sorte :

```
#ifndef __TAD__
#define __TAD__

// Contenu du header

#endif
```

où `TAD` est cohérent avec le nom du TAD et le nom du header.

Type opaque La structure de données concrètes implémenant le TAD sera présentée comme un *type opaque* (cfr. INFO0030, Partie 1, Chapitre 1, Slide 11). Par définition, la structure de données sera manipulée, dans l'interface, via un pointeur. Il est impossible de procéder autrement, la structure étant incomplètement définie. La forme générale est

```
typedef struct tad_t TAD;
```

Prototypes Il s'agit des *signatures* (type de retour, identifiant, liste des paramètres formels – cfr. INFO0946, Chapitre 6, Slides →) des fonctions/ procédures correspondant aux opérations du TAD. Attention, il faut être cohérent (même identifiant, même liste d'arguments) par rapport à la définition des opérations dans la syntaxe du TAD. Attention, si le TAD utilise un type `Element`, il sera remplacé par `void *`.

Documentation . Idéalement, l'interface sera documentée (header complet, structure, fonctions) avec l'aide de l'outil Doxygen (cfr. INFO0030, Partie 2, Chapitre 4).

5.4.1.2 Le Module

Le module contient l'implémentation proprement dite du TAD et se présente dans un fichier dont, typiquement, le nom est `tad.c` (où `tad` est le nom du TAD). En particulier, le module est structuré comme suit :

Inclusion Il s'agit d'inclure le header correspondant.

```
1 #include "tad.h"
```

Défintion du Type Il s'agit de compléter la définition du type. À savoir :

```
1 struct tad_t{
    //les différents champs
3 };
```

Implémentation des Opérations . L'ordre importe peu. Attention, si jamais on désire « externaliser » une fonctionnalité dans une fonction/procédure particulière, on veillera à la définir comme `static` pour éviter son export et son utilisation en dehors du module.

Attention, pour une même interface (`tad.h`), il peut y avoir plusieurs implémentations (modules) possibles.

Suite de l'exercice

À vous ! Passez à l'implémentation du header et voyez ensuite la section 5.4.2.

5.4.2 Header

```c
#include<math.h>
#ifndef __COMPLEX__
#define __COMPLEX__

struct complex_t;
typedef struct complex_t Complex;

// Create a new complex
// @Pre: /
// @Post: return a pointer to the Complex representing (a+bi), NULL if error
Complex *complex(double a, double b);

// Create a new zero complex (equivalent to complex(0.0, 0.0) )
Complex *zero(void);

// Complex destructor
// @Pre: *c init
// @Post: *c_0 freed ∧ *c == NULL
void free_complex(Complex **c)

// Addition
// @Pre: x init ∧ y init ∧ z init
// @Post: *z = *add = *x + *y
Complex *add(Complex *x, Complex *y, Complex *z);

// Substraction
// @Pre: x init ∧ y init ∧ z init
// @Post: *z = *sub = *x - *y
Complex *sub(Complex *x, Complex *y, Complex *z);

// Multiplication
// @Pre: x init ∧ y init ∧ z init
// @Post: *z = *mul = *x × *y
Complex *mul(Complex *x, Complex *y, Complex *z);

// Division
// @Pre: x init ∧ y init ∧ z init ∧*y != 0+0i
// @Post: *z = *div = *x / *y
Complex *div(Complex *x, Complex *y, Complex *z);

// Conjugate
// @Pre: x != NULL
// @Post: *z = *conj = $\overline{*x}$
Complex *conj(Complex *x, Complex *z);

// Imaginary part
// @Pre: x != NULL
// @Post: imag = $\Im(*x)$
double imag(Complex *x);

// Real part
// @Pre: x != NULL
// @Post: real = $\Re(*x)$
double real(Complex *x);

// Modulus
// @Pre: x != NULL
// @Post: modulus = $|*x|$
double modulus(Complex *x);
#endif
```

complex.h

Le contenu du header est, somme toute, assez classique. On remarquera que les fonctions transformateurs prennent trois arguments : les 2 opérandes et le résultat. En fait, on va faire en sorte de laisser l'entièreté de la gestion de la mémoire à l'utilisateur. C'est lui qui décide exactement quand créer un `Complex` et quand le détruire. On aurait pu faire une implémentation dans laquelle `add()`, `sub')`,... créent d'abord un nouveau `Complex` avant de calculer le résultat de l'opération et de le stocker dans le nouveau `Complex` créé. Mais cela demande de faire des allocations de mémoires dans plus ou moins tous les modules et il faut correctement le documenter parce que l'utilisateur doit dans ce cas être au courant qu'il alloue un `Complex` dès qu'il lance un calcul. Ce n'est donc pas le cas ici.

On inclut `math.h` qui sera nécessaire pour le calcul du module (utilisation de `sqrt`).

Le destructeur `free_complex()` prend un pointeur de pointeur vers un `Complex`, donc une adresse d'adresse de `Complex`. En déréférençant cette adresse, on peut libérer l'espace alloué pour le `Complex`. On peut ensuite mettre cette adresse de `Complex` à `NULL` pour plus de sécurité et l'utilisateur n'a pas besoin de s'en préoccuper. C'est donc une bonne pratique **à adopter** [8].

Alerte : Pointeur pendouillant

Traduction libre de l'anglais *dangling pointer*, le pointeur pendouillant est un pointeur qui ne pointe pas vers une zone allouée. C'est par exemple le cas des pointeurs que l'on vient tout juste de libérer. Il ne faut absolument pas les déréférencer dans la suite du code. De plus, il n'est pas certain qu'un déréférencement déclenche une **erreur de segmentation** !

Le meilleur remède reste avant tout la prévention. Il suffit, après avoir libéré le pointeur, de le mettre à la valeur `NULL`. Il sera alors facile de déboguer un code qui le déréférence puisque ce dernier échouera dans tous les cas sur une erreur de segmentation. Pratique !

Suite de l'exercice

Passons maintenant à l'implémentation, en particulier le module. Une proposition de solution est donnée à la Sec. 5.5.

8. Si vous ne l'avez pas fait dans vos premiers projets d'INFO0030 et qu'on ne vous a pas retiré la moitié des points du projet, remerciez le ciel : tout le monde n'est pas aussi bâtard que Simon. Méfiez vous pour INFO0947...

5.5 Implémentation du Module

Il faut maintenant définir le module pour la structure de données.

Si vous ne vous souvenez plus de ce qu'il faut faire, voyez la section 5.4.1.2
Comparez ensuite votre propre code à la section 5.5.1

5.5.1 Fichier `complex.c`

```c
#include"complex.h"

typedef struct complex_t{
   double real;
   double imag;
} Complex;

Complex *create(double a, double b){
   Complex *temp = malloc(sizeof Complex);
   if (temp == NULL);
      return NULL;

   temp->real = a;
   temp->imag = b;
   return temp;
} // end create()

Complex *zero(void){
   return create(0.0, 0.0);
} // end zero()

Complex *add(Complex *x, Complex *y, Complex *z){
   assert(x != NULL && y != NULL && z != NULL);

   z->real = x->real + y->real;
   z->imag = x->imag + y->imag;
   return z;
} // end add()

Complex *sub(Complex *x, Complex *y, Complex *z){
   assert(x != NULL && y != NULL && z != NULL);

   z->real = x->real - y->real;
   z->imag = x->imag - y->imag;
   return z;
} // end sub()

Complex *mul(Complex *x, Complex *y, Complex *z){
   assert(x != NULL && y != NULL && z != NULL);

   double a_x = x->real, b_x = x->imag, a_y = y->real, b_y = y->imag;

   z->real = a_x * a_y - b_x * b_y;
   z->imag = a_x * b_y + b_x * a_y;
   return z;
} // end mul()

Complex *div(Complex *x, Complex *y, Complex *z){
   assert(x != NULL && y != NULL && z != NULL && y.real != 0 && y.imag != 0);

   double a_x = x->real, b_x = x->imag, a_y = y->real, b_y = y->imag;
   double denom = a_y * a_y + b_y * b_y;

   z->real = (a_x * a_y + b_x * b_y) / denom;
   z->imag = (-a_x * b_y + b_x * a_y) / denom;
   return z;
} // end div()

Complex *conj(Complex *x, Complex *z){
   assert(x != NULL && z != NULL);

   z->real = x->real;
```

```
   z->imag = -(x->imag);
   return z;
} // end conj()

double imag(Complex *x){
   assert(x != NULL);
   return x->imag;
} // end imag()

double real(Complex *x){
   assert(x != NULL);
   return x->real;
} // end real()

void erase(Complex **c){
   assert(c);
   free(*c);
   *c = NULL;
}// end erase()

double modulus(Complex *x){
   return sqrt(x->real * x->real + x->imag * x->imag);
}// end modulus
```

complex.c

Quelques remarques :

- On n'oublie pas la programmation défensive ;
- Ni d'inclure le header `complex.h`
- Dans `mul()` et `div()`, on utilise des variables temporaires parce que rien n'interdit à l'utilisateur de donner le même pointeur de `Complex` comme argument `x` et `z`. Il pourrait vouloir faire quelque chose du style `x += y`. C'est possible avec notre implémentation.
- On aurait pu rajouter des assertions intermédiaires mais elles sont, pour une fois, vraiment triviales. C'est laissé au lecteur à titre d'exercice.
- Si on veut ajouter des assertions intermédiaires, il faut réécrire des Préconditions et Postconditions concrètes (qui tiennent compte de la représentation du TAD). Ce n'est pas le cas de celles qui sont dans le `.h` qui utilisent les notations bien connues des nombres complexes.
- Une constante `zero()` a été rajoutée pour créer rapidement un `Complex` égal à zéro. Il n'a pas d'argument donc il faut écrire le mot réservé `void` comme paramètre formel. C'est le standard du langage qui le requiert.

Suite de l'exercice

Passons maintenant à l'implémentation, en particulier un programme utilisant les fonctionnalités décrites dans `Complex`. Une proposition de solution est donnée à la Sec. 5.6.

5.6 Programme Utilisant le TAD

Ce n'est pas bien compliqué à rédiger. On va faire un petit `main()` qui utilise `add()`. Vous ferez bien le reste en exercices à la maison.

```c
#include<stdio.h>
#include"complex.h"

int main(void){

   Complex *x = complex(18. ,25.);
   Complex *x = complex(17. ,42.);
   Complex *z;

   z = add(x, y, zero());

   printf("Resultat = %f + %fi", real(z), imag(z));

   // On n'oublie pas le nettoyage :
   free_complex(&x);
   free_complex(&y);
   free_complex(&z);

   return 0;
}
```

C'est un programme qui utilise le TAD. Il n'est marqué nulle part que c'est un programme de test. Proposer ce programme comme test, c'est pour se voir attribuer une cote générée par le module `zero()`. Pour bien tester, il faut évidemment utiliser une solution de test unitaire comme, par exemple, Seatest et tester tous les modules, dans tous les cas possibles (voir INFO0030, Partie 2, Chapitre 3, Slides 8 → 29). A faire à titre d'exercice à la maison (n'hésitez pas à soumettre vos tests unitaires sur eCampus)