APAD

Algorithmique et programmation 2

–

Journée 1

Récursivité, type abstrait de données et modules

Xavier Crégut

<cregut@enseeiht.fr>

Septembre 2003

1 Sous-programme récursif

1.1 Définition

Définition : Un sous-programme récusif est un sous-programme qui se rappelle (son corps contient un appel à lui-même).

Exemple : Fonction récursive de calcul de la factorielle :

```
Fonction factorielle(n: in Entier): Entier Est
        -- Factorielle de n
        -- Nécessite :
        --      n >= 0
    Début
        Si n <= 1 Alors
            Résultat ← 1
        Sinon
            Résultat ← n * factorielle(n-1)
        FinSi
    Fin
```

Remarque : `factorielle(n-1)` est l'appel récursif.

1.2 Fondement mathématique

Le corps de la fonction `factorielle` précédente correspond à la définition mathématique de la factorielle donnée sous forme de récurrence :

$$n! = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Remarque : On pourrait mathématiquement utiliser la notation suivante pour définir la factorielle :

$$n! = \prod_{i=1}^{i=n} i = 1 \times 2 \times 3 \ldots \times (n-1) \times n$$

Elle conduirait à un sous-programme non récursif utilisant une boucle **`Pour`**.

Remarque : On peut souvent (mais pas toujours !) supprimer la récursivité.

1.3 Terminaison

Danger : Il faut s'assurer que les appels récursifs s'arrêtent pour garantir la terminaison du programme.

Régle : Un sous-programme récursif doit toujours faire apparaître deux éléments :

1. le cas terminal où l'on sait écrire le code sans nouvel appel récursif ;
2. le cas général dans le quel on fait des appels récursifs sur un problème de taille strictement inférieure (garantie de terminaison).

Exemple : Dans le cas de la factorielle :

– on définit la taille du problème de `factorielle(n)` comme étant n ;
– le cas terminal correspond à $n \leq 1$ ($n = 0$ ou 1) où la factorielle est 1 ;
– dans le cas général ($n > 1$), on utilise l'appel récursif `factorielle(n-1)` de taille strictement inférieure ($n - 1 < n$).

1.4 Exercice

Exercice 1 : Les tours de Hanoï

C'est Lucas, mathématicien français, qui sous le pseudonyme de Claus a inventé ce jeu.

Il se présente sous la forme d'un support en bois sur lequel sont plantés trois tiges A, B et C. Sur ces trois tiges peuvent être enfilés des disques de diamètres différents (8 dans la version originale, mais N de manière générale). Dans la configuration initiale (figure 1), les disques sont empilés par ordre de taille décroissante sur la tige A.

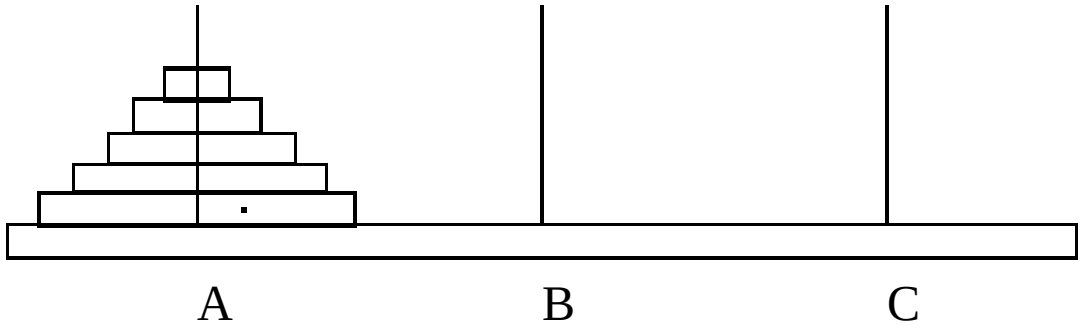

FIG. 1 – Configuration initiale du jeu des tours de Hanoï

Écrire un programme qui donne la solution de ce jeu (c'est-à-dire, la liste des coups à jouer). Pour cela, on remarquera qu'un coup est complètement déterminé par la donnée de la tige de départ et de la tige d'arrivée, car on ne peut déplacer qu'un disque à la fois (celui qui se trouve en haut de la tige).

2 Types abstraits de données

2.1 Motivation

Définir un type seul n'a que peu d'intérêt car les données de ce type devront être manipulées. Il est donc généralement utile de fournir les opérations (les sous-programmes) de « haut » niveau sur ces données.

Exemple : Le type `Entier` définit les entiers relatifs (0, 1, -1, 2,-2, 3, -3, etc). Mais quelle serait l'utilité des entiers sans les opérations usuelles (incrémentation, addition, soustraction, multiplication, division, comparaison) ou plus « évoluées » (puissance, pgcd, factorielle, etc.).

Important : Un type n'a d'intérêt que par les opérations qu'il propose !

2.2 Exemple : le type Date

Consisérons le type Date correspondant à l'identification d'un jour dans une année.

Quelles sont les opérations dont nous aimerions disposer si nous devons écrire une application manipulant des dates (mesure de temps d'exécution, agenda...) ?

Réponse : Voici quelque-unes des opérations que l'on peut souhaiter faire :

– connaître l'année, le mois et le numéro du jour dans le mois ;
– initialiser une date ;
– connaître le jour de la semaine d'une date (lundi, mardi, etc.) ;
– comparer des dates (égales, antérieure, postérieure) ;
– calculer la durée qui sépare deux dates ;
– augmenter/diminuer une date d'une certaine durée ;
– ...

Attention : Pour définir un TAD, on n'a pas besoin de connaître la structure du type ni le code des opérations, d'où le terme **abstrait**, au sens indépendant d'une implantation particulière.

Remarque : Donner le nom et la description d'une opération est insuffisant pour bien la comprendre. Les TAD proposent une **partie axiomatique** qui permet de décrire les propriétés des opérations (non présentée ici). Ces propriétés peuvent partiellement s'exprimer en utilisant la programmation par contrat.

2.3 Réalisation d'un type abstrait de données

Par définition, un type abstrait de données est un ensemble de types et de sous-programmes abstraits. Les choix de représentation et d'implantation ne sont pas encore faits.

Pour réaliser un TAD, il faut faire ces choix puis implémenter les opérations.

Comme un TAD peut être réutilisé dans différents contextes, il est préférable de le définir sous la forme d'un **module**.

3 Modules

3.1 Motivation

Le but des modules est de pouvoir regrouper au sein d'une même unité syntaxique (le module) des informations (constantes, types, sous-programmes, etc.) qui pourront être utilisées (et réutilisées) dans plusieurs programmes.

C'est la notion d'**encapsulation**.

Exemples : Voici quelques exemples de modules possibles :

– un module mathématique regroupant les opérations mathématiques telles que la racine carrée, la puissance, les fonctions trigonométriques, etc ;
– un module définissant un type Date et les opérations associées.

Synonymes : Paquetage, unité... classe (une classe est plus qu'un module).

3.2 Structure d'un module

Un module est composé de deux parties :

- la **spécification** qui déclare les informations qui sont fournies par le module et donc utilisables par les autres modules et programmes :
  - des constantes ;
  - des types ;
  - des sous-programmes.
- une **implémentation** qui définit le corps des sous-programmes.

Remarque : L'implémentation peut définir de nouveaux types, constantes et sous-programmes mais ils ne pourront pas être utilisés à l'extérieur du module. Ce sont des éléments seulement nécessaires à l'implémentation du module (écriture et structuration du corps des sous-programmes *exportés* par le module). C'est le **masquage d'information**.

Danger : Même si c'est possible, il est interdit de déclarer une variable dans la spécification d'un module car ce serait une variable globale !

3.3 Exemple

Écrire un module qui définit le type Date et les opérations associées que nous limiterons à :

– initialiser une date à partir d'un jour, d'un mois et d'une année ;
– afficher une date sous la forme jj/mm/aaaa.

Remarque : Ce module est rudimentaire. Il pourrait être compléter par d'autres opérations utiles sur les dates (comparaison, nom du jour dans la semaine, vérification de validité d'une date, incrémentation, décrémentation...)

Intérêt : Si nous avons besoin de manipuler une date, il suffit d'utiliser le module et les opérations qu'il fournit.

3.3.1 Spécification du module Date

```
Module Date Spécification Est

Type
    Date =
        Enregistrement
            jour: 1..31          -- Le numéro du jour
            mois: 1..12          -- le numéro du mois : 1 janvier...
            année: Entier        -- année > 0
        FinEnregistrement

Procédure set_date(une_date: out Date; jj, mm, aaaa: in Entier)
    -- Initialiser une_date à partir de jj/mm/aaaa.
    --
    -- Nécessite :
    --          est_date_valide(jj, mm, aaaa)

Procédure afficher_date(une_date: in Date)
    -- Afficher une_date au format jj/mm/aaaa.

Fin
```

3.3.2 Implémentation du module Date

```
Module Date Implémentation Est

Procédure set_date(une_date: out Date; jj, mm, aaaa: in Entier)
    Début
        une_date.jour ⟵ jj
        une_date.mois ⟵ mm
        une_date.année ⟵ aaaa
    Fin

Procédure afficher_sur_deux_chiffres(un_entier: in Entier)
        -- Afficher un_entier sur au moins deux chiffres en ajoutant
        -- éventuellement un zéro en tête.
    Début
        Si un_entier < 10 Alors
            Écrire('0')
        FinSi
        Écrire(un_entier)
    Fin

Procédure afficher_date(une_date: in Date)
    Début
        -- Afficher le jour
        afficher_sur_deux_chiffres(une_date.jour)
```

```
        Écrire('/')

        -- Afficher le mois
        afficher_sur_deux_chiffres(une_date.mois)
        Écrire('/')

        -- Afficher l'année
        Écrire(une_date.année)
    Fin

Fin
```

Exercice 2 Que faudrait-il changer pour contrôler la validité de l'initialisation de la date par une précondition ?

3.4 Utilisation d'un module dans un programme

Lorsque l'on veut utiliser un module, il suffit de le dire explicitement (**Utilise**).

```
Algorithme TestDate Est

Utilise Date

Variable
    d: Date
Début
    set_date(d, 31, 6, 2000)
    afficher_date(d)
Fin
```

Remarque : **Utilise** Date donne accès à toutes les informations décrites dans la spécification du module Date. Elles peuvent donc être utilisées dans le programme.

3.5 Utilisation d'un module dans un autre module

Si un module veut utiliser un autre module, il suffit de l'indiquer par un **`Utilise`** qui peut être mis soit dans sa spécification, soit dans son implémentation.

Remarque : Dans l'implémentation d'un module on peut directement utiliser (sans préciser **`Utilise`**) les modules listés dans la clause **`Utilise`** de la spécification.

Règle : On mettra toujours le **`Utilise`** dans l'implémentation, sauf s'il est nécessaire de le mettre dans la spécification.

Justification : La dépendance est moins forte si **`Utilise`** est mis dans l'implémentation.

3.6 Exercice

Exercice 3 Comment définir un module qui définit un type RendezVous étant donné qu'un rendez-vous est caractérisé par une heure de début, une heure de fin et un intitulé décrivant la nature du rendez-vous ?

3.7 Taxonomie des modules

Un module est un regroupement d'information sans sémantique imposée. Cependant, on peut identifier deux catégories de modules.

- Les **modules utilitaires** qui sont un regroupement de sous-programmes liés à un domaine :
  - module mathématique ;
  - module de gestion de l'affichage...

  Les sous-programmes sont reliés par un thème commun.
- Les **modules issus d'un type abstrait de données**. Ils encapsulent un type et les opérations associées :
  - un module date ;
  - un module fraction...

3.8 Intérêt des modules

Les modules ont plusieurs intérêts :

– **structuration** de l'application à un niveau supplémentaire par rapport aux sous-programmes ;
– **réutilisation** de code entre applications ;
– **amélioration de la maintenance** (une évolution dans l'implémentation d'un module n'a pas d'impact sur les autres modules d'une application) ;
– **amélioration du temps de compilation** grâce à la compilation séparée.

4 Les modules en C

Attention : Aucune structure syntaxique du langage C ne correspond à la notion de module. Pour les mettre en œuvre, il faut utiliser plusieurs fichiers et le préprocesseur. **C'est au programmeur de le faire.**

$\Longrightarrow$ Le programmeur doit donc s'astreindre à une discipline :

- écrire un fichier .h pour la spécification du module (les déclarations) ;
- écrire un fichier .c pour l'implémentation du module (utiliser **`static`** pour définir une entité locale au module) ;
- utiliser le préprocesseur :
  - **#include** pour inclure la déclaration d'un module ;
  - **#ifndef**, **#define** et **#endif** pour protéger les déclarations contre le risque d'inclusions multiples.

Remarque : Le compilateur C n'apporte (presqu') aucune aide dans la vérification des modules. La plupart des erreurs sont détectées par l'éditeur de liens !

4.1 Spécification du module Date

```
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Définition d'un TAD Date simplifié.
 *****************************************************************/

#ifndef TAD_DATE_SIMPLE__H
#define TAD_DATE_SIMPLE__H

struct Date {
    int jour;   /* le numéro du jour dans le mois */
    int mois;   /* le numéro du mois dans l'année */
    int annee;  /* le numéro de l'année */
};
typedef struct Date Date;

void set_date(Date *une_date, int jj, int mm, int aaaa);
        /* Initialiser une_date à partir de jj/mm/aaaa.  */

void afficher_date(Date une_date);
        /* Afficher une_date au forme jj/mm/aaaa. */

#endif
```

4.2 Spécification du module Date

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Implémentation du TAD Date.
 *****************************************************************/

#include <stdio.h>

#include "tad-date-simple.h"

void set_date(Date *une_date, int jj, int mm, int aaaa)
{
    une_date->jour = jj;
    une_date->mois = mm;
    une_date->annee = aaaa;
}

static void afficher_sur_deux_chiffres(int un_entier)
        /* Afficher un_entier sur au moins deux chiffres en ajoutant
         * éventuellement un zéro en tête.
         */
{
    if (un_entier < 10) {
```

```
        printf("0");
     }
    printf("%d", un_entier);
}

void afficher_date(Date une_date)
{
    /* Afficher le jour */
    afficher_sur_deux_chiffres(une_date.jour);
    printf("/");

    /* Afficher le mois */
    afficher_sur_deux_chiffres(une_date.mois);
    printf("/");

    /* Afficher l'année */
    printf("%d", une_date.annee);
}
```

4.3 Spécification du module Date

```
/*************************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Programme de test du TAD Date
 ************************************************************************/

#include <stdlib.h>

#include "tad-date-simple.h"

int main()
{
    Date d;

    set_date(&d, 31, 6, 2003);
    afficher_date(d);
    return EXIT_SUCCESS;
}
```

4.4 Questions de synthèse

- Lors de l'écriture d'un module où faut-il mettre les commentaires ? Dans le fichier de spécification, le fichier d'implémentation, les deux ou aucun ?
- Un module peut-il définir la fonction `main` ? Pourquoi ?
- Comment sait-on quelles directives **`#include`** mettre dans le fichier de spécification ? Et dans le fichier d'implémentation ?
- Que signifie **`static`** et pourquoi le mettre ? Que pourrait se passer si on oublie de mettre **`static`** ?
- Pourquoi utiliser l'extension `.h` pour la spécification d'un « module » ?
- Pourquoi doit-on utiliser **`#ifndef`**`...` dans un fichier d'entête (`.h`) ?
- Que se passe-t-il si on oublie dans le fichier `.c` de donner le code d'une fonction déclarée dans le fichier `.h` ?

Table des matières