Algorithmique et programmation 2 : les modules

Résumé

Ce document décrit la notion de module. Ce document décrit l'écriture dans le langage C des éléments vus en algorithmique.

Table des matières

1 Motivation **2**

2 Structure d'un module **2**

3 Exemple **3**
3.1 Spécification du module Date 3
3.2 Utilisation d'un module dans un programme 3
3.3 Implémentation du module Date 4

4 Utilisation d'un module dans un autre module **5**

5 Intérêt des modules **5**

6 Taxonomie des modules **5**

7 Modules en C **6**
7.1 Spécification du module 6
7.2 Implantation du module 7
7.3 Utilisation d'un module 8

Liste des exercices

Exercice 1 : Définir un module Date *3*
Exercice 2 : Validité d'une date *4*

1 Motivation

Le but des modules est de pouvoir regrouper au sein d'une même unité syntaxique (le *module*) des informations (constantes, types, SP...) 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).

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 l'implémentation des sous-programmes.

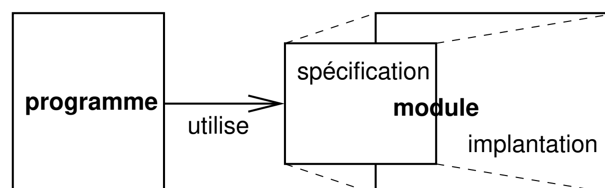
Propriété : Seule la spécification est connue de l'utilisateur, l'implémentation lui est cachée (**masquage d'information**).

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 !

Règle : L'implémentation d'un module doit au moins définir le corps des sous-programmes déclarés dans la spécification de ce module.

Remarque : L'implémentation contient généralement plus d'informations que la spécification... mais elles ne sont pas accessibles de l'extérieur.

Justification : Dans l'implémentation du module, pour définir le corps des SP, on peut définir de nouveaux types, constantes et SP mais ils seront locaux au module.



3 Exemple

Exercice 1 : Définir un module Date
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 complété par d'autres opérations utiles sur les dates (comparaison, nom du jour dans la semaine, vérification de la 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.

3.1 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
            année: Entier        -- année > 0
        FinEnregistrement

Procédure initialiser_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
```

Remarque : La fonction `est_date_valide` devrait être définie !

3.2 Utilisation d'un module dans un programme

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

```
Algorithme Application Est

Utilise
    Date

Variable
    d1, d2: Date
Début
    initialiser_date(d1, 1, 1, 2000)
    afficher_date(d1)
    initialiser_date(d2, 31, 12, 2000)
```

```
12     afficher_date(d2)
13 Fin
```

Remarque : `Utilise Date` donne accès à toutes les informations déclarées dans la spécification du module Date. Elles peuvent donc être utilisées dans le programme.
Remarque : Connaître l'implémentation du module n'est pas nécessaire !

3.3 Implémentation du module Date

```
1  Module Date Implémentation Est
2
3  Procédure initialiser_date(une_date: out Date
4                             jj, mm, aaaa: in Entier)
5      Début
6          une_date.jour <- jj
7          une_date.mois <- mm
8          une_date.année <- aaaa
9      Fin
10
11 Procédure afficher_sur_deux_chiffres(un_entier: in Entier)
12         -- Afficher un_entier sur au moins deux chiffres en
13         -- ajoutant éventuellement un zéro en tête.
14         --
15         -- Nécessite
16         --      (un_entier >= 0) Et (un_entier <= 99)
17     Début
18         Si un_entier < 10 Alors
19             Écrire('0')
20         FinSi
21         Écrire(un_entier)
22     Fin
23
24 Procédure afficher_date(une_date: in Date)
25     Début
26         -- Afficher le jour
27         afficher_sur_deux_chiffres(une_date.jour)
28         Écrire('/')
29
30         -- Afficher le mois
31         afficher_sur_deux_chiffres(une_date.mois)
32         Écrire('/')
33
34         -- Afficher l'année
35         Écrire(une_date.année)
36     Fin
37 Fin
```

- Le SP `afficher_sur_deux_chiffres` est local au module.
- Seule est donnée la sémantique des SP locaux aux modules.
- Dans l'implémentation d'un module, il est possible de mettre un bloc d'instructions pour initialiser le module (si c'est nécessaire).
  Attention, il ne s'agit en aucun cas d'un programme principal !

Exercice 2 Que faudrait-il changer pour contrôler la validité des dates ?

4 Utilisation d'un module dans un autre module

Si un module veut utiliser une information d'un autre module, il suffit de l'indiquer par un `Utilise` qui peut être placé :

— soit dans sa spécification,
— soit dans son implémentation.

Attention : L'information **doit** être dans la spécification de l'autre module !

Remarque : Les modules « utilisés » dans la spécification le sont également pour l'implémentation (inutile de remettre un `Utilise`).

Règle : Toujours mettre le `Utilise` dans l'implémentation, sauf s'il est nécessaire de le mettre dans la spécification. En d'autres termes, on ne mettra un `Utilise` dans la spécification si et seulement si un élément du module importé est utilisé dans la spécification (en général, une définition de type).

Justification : La dépendance est moins forte si `Utilise` est mis dans l'implémentation. Le programme utilisant du module A ne sait pas quels autres modules ont été utilisés pour l'écrire (par exemple le module B). Ainsi, on pourra changer l'implantation du module A et remplacer le module B par un autre module C sans que celà n'interfère avec le programme utilisateur.

Attention : Deux modules A et B ne peuvent pas s'utiliser mutuellement dans la spécification (mais ils le peuvent dans l'implémentation).

5 Intérêt des modules

Les modules ont plusieurs intérêts :

— **structuration** de l'application à un niveau supplémentaire par rapport aux sous-programmes (conception globale du modèle en V) ;
— **factorisation/réutilisation** de code (SP) 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 de chaque module.

6 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...

7 Modules en C

La notion de module n'existe pas en C. Cependant, le programmeur peut arriver à un résultat similaire en utilisant plusieurs fichiers et la directive **`#include`** du préprocesseur.

7.1 Spécification du module

Le principe est donc de faire un fichier avec l'extension `.h` (comme *header*, entête en anglais) dans lequel apparaît la spécification des des sous-programmes qui doivent faire partie de la spécification du module. Par exemple, on peut faire un fichier `date_simple.h` pour la spécification du module « `date_simple` ».

```
1  /****************************************************************
2   *  Auteur    : Xavier Crégut <cregut@enseeiht.fr>
3   *  Version   : Revision
4   *  Objectif : Spécification du module Date.
5   ****************************************************************/
6
7  #ifndef date_simple__H
8  #define date_simple__H
9
10 struct Date {
11     int jour;    /* compris entre 1 et 31 */
12     int mois;    /* compris entre 1 et 12 */
13     int annee;   /* strictement positive */
14 };
15
16 typedef struct Date Date;
17
18
19 /* Initialiser une_date à partir de jj/mm/aaaa.
20  *
21  * Nécessite :
22  *             est_date_valide(jj, mm, aaaa);
23  */
24 void initialiser_date(Date *une_date, int jj, int mm, int aaaa);
25
26 /* Afficher une_date au format jj/mm/aaaa. */
27 void afficher_date(Date une_date);
28
29 #endif
```

Notons les directives qui permettent d'éviter que la double inclusion d'un même fichier d'entête provoque une erreur de programmation.

```
1  #ifndef date_simple__H
2  #define date_simple__H
3
4  ...
```

```
5
6 #endif
```

Le nom `date_simple__H` doit être unique au sein de l'application. En général, on le construit en prenant le nom du module suivi de `__H`, les deux soulignés permettent d'éviter un conflit avec un nom de sous-programme ou variable utilisé dans le reste du programme.

7.2 Implantation du module

Le fichier d'implantation est un fichier `.c`, par exemple `date_simple.c` qui contient l'implantation du module.

```c
/****************************************************************
 *  Auteur    : Xavier Crégut <cregut@enseeiht.fr>
 *  Version   : Revision
 *  Objectif : Implantation du module Date.
 ****************************************************************/

#include "date_simple.h"

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

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

/* Afficher un_entier sur au moins deux chiffres en
 * ajoutant éventuellement un zéro en tête.
 *
 * Nécessite
 *       (un_entier >= 0) Et (un_entier <= 99)
 */
static void afficher_sur_deux_chiffres(int un_entier) {
    assert(un_entier >= 0 && un_entier <= 99);

    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);
}
```

Notons que l'on commence ce fichier par inclure le fichier de spécification pour avoir accès aux définitions de la spécification du module.

Avec **`#include`**, on utiliser les guillemets pour inclure un fichier qui est dans le même répertoire et les <> pour inclure un fichier qui est dans la bibliothèque standard (la recherche ne considère pas le répertoire courant).

Notons également que l'inclusion des modules « stdio » et « assert » n'a lieu que dans l'implémentation et non dans la spécification. Les informations fournies par ces modules ne sont utilisées que dans l'implémentation du module « `date_simple` » et non dans sa spécication.

Enfin, un **`static`** a été mis devant le sous-programme « `afficher_sur_deux_chiffres` » puisqu'elle est locale au module (elle apparaît dans son implantation et non dans sa spécification).

7.3 Utilisation d'un module

Enfin, dans le fichier `test_date_simple.c` qui correspond au programme principal, on inclut (avec **`#include`**) le fichier qui contient la spécification du module « Date ». Ceci permet au compilateur de connaître la définition du type Date et de vérifier la bonne utilisation des opérations associées.

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

#include "date_simple.h"

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

int main()
{
    Date d1, d2;

    initialiser_date(&d1, 1, 1, 2000);
    afficher_date(d1);
    printf("\n");

    initialiser_date(&d2, 31, 12, 2000);
    afficher_date(d2);
    printf("\n");

    return EXIT_SUCCESS;
}
```