Cours “Algorithmique et Structure des Données”

Licence L2 Informatique, semestre S3

Année 2019-2022

Analyse descendante

Rappel :

Algorithme
$=$
expression d'une méthode de résolution par décomposition du traitement à effectuer en actions élémentaires, en précisant l'ordre de leur exécution.

Analyse descendante : on décompose le problème en actions que l'on décomposera ensuite en sous-actions, puis en *actions élémentaires*.
Il est intéressant de considérer certaines actions comme des **modules**.

Par définition, un module désigne une entité de données et d'instructions qui fournissent une solution à une partie bien définie d'un problème plus complexe.
N.B. Dans une terminologie plus ancienne, on parlait de sous-programmes.

Avantages des modules (début)

- **Meilleure lisibilité** de la méthode de résolution du problème.
- **Suppression des codes dupliqués** : une même action qui doit être exécutée à plusieurs endroits ne sera écrite qu'une seule fois.
- **Possibilité de tests sélectifs** d'un module, indépendamment du problème complet $\Rightarrow$ plus facile de cerner puis corriger une erreur.
- **Dissimulation des méthodes** : pour utiliser un module, il suffit de connaître son effet, sans s'occuper des détails de sa réalisation.
- **Réutilisation de modules existants** : il est facile d'utiliser des modules qu'on a écrits soi-même ou qui ont été développés par d'autres personnes (*exemple :* les bibliothèques ou "libraries").

Exemples de modules "recyclables" :

affichage des valeurs d'un tableau ;

saisie d'une valeur numérique, avec boucle pour interdire des valeurs hors limites.

Avantages des modules (fin)

- **Simplicité de la mise à jour** : un module peut être modifié ou remplacé indépendamment des autres modules du programme.
- **Favorisation du travail en équipe** : un algorithme peut être développé en équipe, en déléguant la conception des modules à différentes personnes. Une fois développés, les modules peuvent constituer une base de travail commune.
- **Hiérarchisation des modules** : un algorithme peut d'abord être décrit globalement au niveau du module principal. Les détails peuvent être reportés à des modules sous-ordonnés qui peuvent eux aussi être subdivisés en sous-modules et ainsi de suite.

Les modules communiquent entre eux, par le biais des appels ; un module peut faire appel à plusieurs *sous-modules* pour sa résolution.

Deux familles de modules

On peut distinguer deux familles de modules :

- ceux qui **réalisent des actions** : on les appelera des **procédures**
- ceux qui **calculent des résultats** : on les appellera des **fonctions**

N.B. La notion algorithmique de fonction généralise la notion de fonction mathématique (ex. $\cos(\pi)$ fournit le résultat -1), mais le résultat d'une fonction peut être une structure de données plus élaborée [voir suite du cours], pas seulement une valeur numérique.

Déclaration, définition et appel d'une procédure

Une procédure constitue un bloc dans l'algorithme. Elle doit être déclarée, donc recevoir un identificateur (i.e. un nom).

Ce nom est suivi d'une liste de **paramètres** placée entre des parenthèses : ce sont les **arguments** sur lesquels la procédure effectuera ses traitements.

La définition de la procédure est un code qui contient :

- une (éventuelle) liste de variables locales
- une liste d'instructions

L'appel de la procédure apparaît, dans le code qui est extérieur à son bloc, sous la forme d'une **instruction** à part entière.

Déclaration, définition et appel d'une fonction

Au même titre qu'une variable, une fonction est déclarée à l'aide d'un identificateur et **un type** lui est attribué : celui du résultat qu'elle fournira.

Ce nom est suivi d'une liste de **paramètres** placée entre des parenthèses : ce sont les **arguments** sur lesquels la fonction effectuera ses traitements.

La définition de la fonction est un code qui contient :

- une (éventuelle) liste de variables locales
- une liste d'instructions
- une instruction finale **retourne** suivie du résultat qu'elle renvoie

L'appel de la fonction apparaît, dans le code qui lui est extérieur, sous la forme d'une **variable** qui est utilisée dans l'évaluation d'une expression, ou dans un affichage, etc. . . .

Exemple de procédure vs fonction

Déclarations et définitions :

Exemple 1 : une procédure qui affiche un texte à partir de trois arguments concernant un employé dans une entreprise :

```
Proc Affiche(Chaine nom, Entier annee, Réel salaire)
    Ecrire(nom+" est né en "+annee+" et il gagne "+salaire+" Euros.")
FinProc
```

Exemple 2 : une fonction qui calcule son âge à partir de son année de naissance :

```
Fonc Entier Age(Entier annee)
    Entier now=2017;          // une variable locale
    retourne(now-annee)
FinFonc
```

Les appels, par le module principal :

```
Chaine SonNom=‘‘Zébulon’’;   Entier NeEn=1972;   Réel Gagne=3456;
Affiche(SonNom,NeEn,Gagne)
Ecrire(‘‘Cet employé a ’’+Age(NeEn)+‘‘ ans.’’)
```

Signature d'une procédure ou d'une fonction

On appelle signature d'une procédure ou d'une fonction la liste de ses paramètres d'appel :

leur nombre, leur ordre et le type de chaque paramètre.

La signature doit être la même lors de la déclaration et lors des appels.

En revanche, les **noms** des variables n'ont aucune raison d'être les mêmes lors de la déclaration et lors des appels.

(cf. les deux exemples ci-dessus)

Variables globales vs locales

Une variable globale est déclarée dans le module principal.

Sa **durée de vie** sera toute la durée de l'exécution.
Elle sera **connue** par toutes les instructions du module principal, ainsi que par toutes celles de toutes les procédures et fonctions.

Une variable locale est déclarée à l'intérieur d'une procédure (ou fonction).

Sa **durée de vie** sera limitée à la durée de l'exécution du module (procédure ou fonction), lors d'un appel. Si le module est appelé une seconde fois, ses variables locales sont à nouveau créées.
Elles sont **connues** par les instructions du module et par toutes ses éventuelles sous-procédures ou sous-fonctions.

Les paramètres

Les variables dont les noms figurent dans la liste des paramètres à la déclaration d'une procédure ou d'une fonction existent **uniquement pendant** que ce module est exécuté (donc suite à un appel).

Par défaut, la correspondance entre les paramètres et les noms des variables passées en argument lors d'un appel se fait **par valeur**,
$\Rightarrow$ on peut considérer les paramètres *comme des variables locales* dont la valeur initiale est celle reçue lors d'un appel.

Par conséquent, les éventuelles modifications effectuées sur ces variables-là, à l'intérieur du code du module, n'ont pas d'effet sur les variables du module appelant.

Passage par adresse

Si l'on souhaite néanmoins qu'un module puisse modifier les valeurs de certaines variables du module qui l'appelle, il faut alors passer ces variables **par adresse**.

Pour cela, il faudra transmettre comme paramètre l'**adresse** d'une variable et non plus sa valeur. On pourra réaliser cela à l'aide de *pointeurs* [voir suite du cours].

Visibilité des modules

WARNING

En suivant la même logique que celle expliquée pour les variables, si un module (procédure ou fonction) est déclaré **à l'intérieur** d'un autre module (i.e. comme sous-module), alors son identificateur ne sera pas connu par les autres modules, indépendants de celui qui l'a déclaré.

En effet, il est considéré comme utilitaire local. Ceci est très pratique lorsqu'on décompose une action, a priori complexe, en sous-actions de plus en plus spécifiques, jusqu'à l'écriture d'actions élémentaires dans les codes des modules de plus bas niveau dans la hiérarchie des appels.

Conseil pratique

Conseil pratique : **EVITER** les variables **globales** !

En effet :
on risque de perdre le contrôle des valeurs d'une variable si elle peut être modifiée aussi bien par un sous-module (lors d'un ou... plusieurs appels) que par le module principal.

$\Rightarrow$ utiliser au maximum des variables locales et des paramètres.

Une fonction peut s'appeler elle-même

Un outil très puissant :

une fonction peut s'appeler elle-même : on parle alors d'appels récursifs.

La notion de **récursivité** est directement liée à la notion mathématique de **récurrence** :

- Une démonstration par récurrence s'appuie sur une preuve au rang n pour en déduire la preuve au rang $n+1$
- Une définition récursive s'appuie sur une formule au rang n pour construire la formule au rang $n+1$

WARNING : ne pas oublier d'établir une preuve ou de définir une formule, de manière directe, à un **rang initial**.

Exemple : puissance n-ième

Il y a deux manières de définir a^n (où a est un réel et n un entier positif).

Définition 1 :

$$a^n = a \times a \times a \times \ldots \times a$$

où le facteur a figure n fois.

Pour programmer cette formule, il faut écrire une boucle. On dit alors que l'on construit le résultat de manière itérative.

Définition 2 :

$$a^n = a \times a^{n-1} \quad \text{en partant de} \quad a^0 = 1 \quad (\text{on suppose } a \neq 0)$$

Pour programmer cette formule, il suffit d'écrire une fonction qui s'appelle elle-même. On dit alors que l'on construit le résultat de manière récursive.

Solution itérative

On écrit l'algorithme qui permet de calculer la formule de la définition 1 :

$$a^n = a \times a \times a \times \ldots \times a$$

```
Réel x=1.2        // Math. : il ne faut pas que x et n soient tous deux nuls
Entier i, n=4

Fonc Réel puiss_iter(Réel a, Entier n)
   Réel prod=1          // initialisation d'un produit (variable locale)
   Pour i de 1 à n
      prod ← prod * a
   Retourne prod
Fin Fonc

Début
   Ecrire("La puissance "+n+" de "+x+" vaut : "+puiss_iter(x,n))
Fin
```

Le produit 2.0736 est construit itérativement au moyen d'une boucle Pour.

Solution récursive

On écrit l'algorithme qui permet de calculer la formule de la définition 2 :

$$a^n = a \times a^{n-1} \quad \text{en partant de} \quad a^0 = 1 \quad (\text{on suppose } a \neq 0)$$

```
Réel x=1.2        // Math. : il ne faut pas que x et n soient tous deux nuls
Entier n=4

Fonc Réel puiss_recu(Réel a, Entier n)
  Si (n=0)        // pas besoin de boucle explicite dans le code de la fonction
    Retourne 1
  Sinon
    Retourne (a * puiss_recu(a, n-1))        // l'appel récursif génère une
boucle Fin Fonc

Début
  Ecrire("La puissance "+n+" de "+x+" vaut : "+puiss_recu(x,n))
Fin
```

La fonction s'appelle elle-même pour calculer récursivement le 2.0736.

Calculer une factorielle

Exercice :

CAHIER DES CHARGES :

Ecrire une fonction itérative, puis une fonction récursive, pour calculer la factorielle d'un entier postitif n :

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

CORRIGE : ..... À SUIVRE .....

Les "fonctions" en langage C

Le langage C ne fait pas de distinction clairement explicite entre une procédure et une fonction.

- Les deux types de modules se déclarent de la même manière. La seule différence est que le **type** d'une procédure sera **void**, c'est-à-dire *vide*.
- Au niveau de la définition, une fonction devra nécessairement avoir comme dernière instruction

```
return (variable ou expression);
```

Et, bien sûr, l'appel d'une procédure se comporte comme une instruction, tandis que celui d'une fonction s'utilise comme une variable.

L'exemple précédent, en C

```c
#include <stdio.h>

void Affiche(char *nom, int annee, double salaire)
{
  printf("%s est né en %d et il gagne %7.2f Euros.\n",nom,annee,salaire);
}

int Age(int annee)
{
  int now=2017;
  return (now-annee);
}

void main()
{
  char SonNom[8]="Zébulon";   // chaîne de caractères... => passée par pointeur
  int NeEn=1972;
  double Gagne=3456;

  Affiche(SonNom,NeEn,Gagne);
  printf("Cet employé a %d ans.\n",Age(NeEn));
}
```

Transmission d'une chaîne ou d'un tableau

Comme on le voit dans l'exemple ci-dessus, une chaîne de caractères doit être transmise, comme paramètre d'une procédure, **via un pointeur**.

En effet, la variable "nom" est créée lorsque la procédure est appelée. Or il est impossible de savoir, quand on définit la procédure, quelle sera la taille de la chaîne (qui, pour le langage C, est un tableau de caractères).

Plus généralement, n'importe quel tableau devra être transmis comme paramètre en utilisant un pointeur.

Passage par adresse

Prenons l'exemple d'une procédure qui échange deux variables. Il faut passer les paramètres par adresse si l'on veut que la modification effectuée par la procédure ait effet sur les variables du module appelant.

Déclaration :

```
void Permute(int *a, int *b)
{
  int aux;
  aux=*a;
  *a=*b;
  *b=aux;
}
```

Appel :

```
Permute(&x,&y);      // on transmet les adresses [comme dans un scanf()]
```

WARNING : l'appel est différent pour le passage d'un tableau [voir suite du cours]